Real-Time Systems, 29, 59–89, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels

TULLIO VARDANEGA tullio.vardanega@math.unipd.it
Department of Pure and Applied Mathematics, University of Padua, Italy

JUAN ZAMORANO jzamora@fi.upm.es
Department of Computer Architecture, Technical University of Madrid, Spain

JUAN ANTONIO DE LA PUENTE jpuente@dit.upm.es
Department of Telematics Engineering, Technical University of Madrid, Spain

Abstract. Mature research advances in scheduling theory show that carefully-crafted concurrent computational models permit static analysis of real-time behavior. This evidence enables designers to consider using suitable forms of explicit concurrency to model the inherent concurrency of real-time systems. The Ravenscar Profile, a specifically tailored subset of the Ada 95 tasking model, defines a compact and efficient concurrent computational model, especially suited for the development of high integrity, high efficiency real-time systems.

Ravenscar runtimes can be implemented by small, efficient, reliable and certifiable kernels. At least two such implementations already exist and are being industrially deployed. The simplicity and intrinsic determinism of Ravenscar kernels facilitate the definition of metrics that cater for very accurate characterization of the dynamic behavior of the runtime and of the execution time of its primitives. Accurate runtime metrics enable forms of response time analysis that minimize the pessimism in the prediction of the runtime influence on the application. This is especially useful for concurrent systems that exhibit significant dependency on runtime support services. This paper recalls the motivations of the Ravenscar Profile, outlines the definition of it and formulates a precise characterisation of the associated runtime metrics.

Keywords: Real-time systems, concurrent programming, tasking restrictions, Ravenscar Profile, static timing analysis, response time analysis

1. Introduction

Real-time systems are inherently concurrent in that they model and confront with real-world entities that possess multiple loci of control. The actual extent of concurrency varies with the type of application, which determines the range of control and service actions to execute, and with the environment in which the execution is to occur. The more composite the environment, the higher the degree of inherent concurrency. The richer the range of actions, the more valuable concurrency as a modeling aid.

This evidence notwithstanding, the prime character of real-time systems is that their correctness must be proven in the time domain as well as in the value domain. This notion raises the question of how best to represent the extent of concurrency of the system while being able to assure the correctness of its timing behavior. Two opposing views clash in this regard: the push for an architecture that would not undermine the cohesiveness and coupling warranted by the nature of the problem; and the pull for a problem representation chiefly designed to simplify verification, even at the cost of some distortion.

As the level of criticality of the system rises, as defined by sector-specific standards (e.g., RTCA, 1992), the traditional designer dispenses entirely with any representation of concurrency that entails multiple threads of control. Traditional approaches model each activity as a distinct procedure and use an application-level executive to invoke them cyclically. This strategy trades rigidity for full determinism and ease of analysis. Yet, it is well known that the traditional approach is unable to attain a faithful representation of the problem at hand and entails the poor engineering practice of artificially constructing tiny procedures to fit residual schedule frames. Locke (1992) has also shown how difficult the cyclic scheme becomes to design for systems of more than just modest complexity, how inflexible to change, how inadequate for applications where aperiodic activity may occur and where error recovery is important. For the comparatively small proportion of real-time systems that range the highest levels of criticality (e.g., aeronautics, nuclear, defense), however, the cost of formal verification and certification is so high that the designer understandably strives to simplify the design to the maximum possible extent, thus taking the traditional approach for the perceived lack of better alternatives.

For an increasing proportion of other real-time systems, the prime concern is the assurance of predictable execution behavior. Failure to meet this requirement may incur consequences of varying criticality, which rises as those systems increasingly pervade our daily life. Lower-criticality systems may attain the required level of assurance much less invasively than for certification, hence, more permissively on their architecture of choice. Deprived of the taming effect of certification, however the design of these systems pulls away from the traditional approach, yet at the risk of flexibility undermining verifiability.

Mature research in scheduling theory has proven that a careful choice of scheduling (dispatching) method, together with suitable restrictions on the allowed interactions between processes renders static analysis of real-time behaviour possible (Audsley et al., 1995).

Preemptive fixed priority scheduling (Burns, 1991, 1994) is a well known scheduling method. Typically it is used with the priority ceiling protocol (Goodenough and Sha, 1988; Sha et al., 1990) to optimally bound priority inversion and avoid deadlocks. Rate monotonic analysis (Klein et al., 1993) and response time analysis (Joseph and Pandya, 1986) are equally known examples of static analysis schemes. The notions underlying these schemes cater for computational models suitable for the analysis of concurrent real-time systems and also scalable to programs for distributed systems. Those models support periodic and aperiodic processes, hard, soft, firm, and non-critical components, and controlled interprocess communication and synchronization.

These notions hold a significant promise, which is of value for the spectrum range of real-time systems, across all integrity requirements. Two essential concerns, however, critically rate their actual effectiveness:

1. whether the allowed computational model is expressive enough to adequately represent increasingly sophisticated real-time systems and yet simple enough to stay amenable to static analysis;

2. whether the corresponding static analysis technique can be made accurate enough to permit high levels of useful processor utilisation, otherwise denied by too rigid or too permissive architectures.

Issue 1 has been positively responded by a steady flow of experience reports, (cf. e.g., Bailey et al., 1993; Dobbing and Romanski, 1999; Vardanega and Caspersen, 2001), which mostly refer to one particular instance of computational model, known as the Ravenscar Profile (Baker and Vardanega, 1997; Burns et al., 1998; Burns, 1999; Wellings, 2001).

Issue 2, instead, is still open and crucial for off-line scheduling analysis to raise its rank among other static analysis techniques. In this paper we address issue 2 specifically, and take the Ravenscar Profile as our computational model (Burns et al., 2003) and response time analysis as the preferred form of analysis. The paper expands on earlier work presented in Vardanega (1999) and in Zamorano and de la Puente (2002).

2. Computational model

2.1. Scheduling Model

At any moment in time, some processes may be ready to run, as they are able to execute instructions if processor time is made available. Other processes are suspended, as they have given up execution until some event occurs. Others still are blocked, as, while ready, they are unable to proceed (e.g., typically because they await access to a shared resource currently exclusively owned by another process). Suspended processes may become ready synchronously, as a result of an action taken by a currently running process, or asynchronously, as a result of an external event, such as an interrupt or timeout, which is not directly stimulated by the running process.

With priority-based preemptive scheduling on a single processor, a priority is assigned to each process and the scheduler ensures that the highest priority ready process is always executing. Contention for mutually-exclusive access to shared resources may incur violation to this rule.

As a process with a priority higher than the running process becomes ready and its execution can resume, the scheduler performs an immediate context switch to it. This event is said to be preemptive as it is not invoked by the running process.

Processes may interact upon contention for shared resources, exchange of data, and the need for explicit synchronisation. Uncontrolled interaction can lead to a number of well-known and studied problems (e.g., deadlock, livelock, missed deadline as well as unbounded priority inversion Cornhill and Sha, 1987).

The use of preemptive priority-based dispatching defines a mechanism for scheduling. The corresponding policy is set by the mapping of processes to priority values. With the priority ceiling protocol, processes possess two priorities: the base priority, which is the one initially assigned to them; and the active priority, which the process acquires as a result of seizing or relinquishing a shared resource. Priority-based scheduling uses the latter value.

2.2. Outline of Ravenscar Restrictions

The restricted scheduling model that is defined by the Ravenscar Profile is designed to minimize the upper bound on blocking time, to prevent deadlocks, and, by tool support, to

verify that there is sufficient processing power available to ensure that all critical processes meet their deadlines. In this model, processes do not interact directly, but do so via special resources known as protected objects, which provide mutually-exclusive access to shared data.

Each protected object typically provides either a resource access control function, including a repository for the private data to manage and implement the resource, or a synchronisation function, or a combination of both (as for data-oriented synchronisation).

A protected object that is used for synchronisation provides a facility that processes can use to signal and/or wait on events. In the Ravenscar Profile, the use of protected objects for synchronisation by the critical tasks is constrained so that at most one task can wait on each protected object, i.e., on the corresponding signal event.

The Profile definition assures absence of deadlocks by requiring use of the priority ceiling protocol (Goodenough and Sha, 1988; Sha et al., 1990), which an Ada program complies with by selecting the default locking policy named `Ceiling_Locking`. This policy requires a priority to be statically assigned to each protected object that is at least as great as the highest priority of all its calling processes, and results in the raising of the priority of the process that is using the protected object to this ceiling priority value. The process then reverts to its previous priority as soon as it leaves the object.

The use of the `Ceiling_Locking` protocol provides an optimal bound for the worst-case duration of priority inversion (Sha et al., 1990). The prescription that, during the period that a process has possession of the object, it must not perform any operation that could result in it becoming suspended, provides a further containing effect.

For any process in the system, the worst-case time bound on this form of priority inversion is calculated as the maximum time that a protected object with higher-priority ceiling may be in use by lower-priority processes. A further form of priority inversion occurs as the runtime disables interrupts to protect the execution of its own critical sections, with the effect of possibly deferring the preemptive arrival of process(es) released by interrupt. Off-line scheduling analysis needs to account for the longest duration of priority-inversion intervals of either kind.

Ravenscar tasks only experience one of these two forms of blocking per single activation, and only prior to dispatching. Figure 1 shows the scheduling states that can be assumed by Ravenscar processes and the transitions that can occur between them.

2.3. The Ravenscar Profile

2.3.1. Justification

Not all computational models are equally amenable to static analysis. In order to allow off-line scheduling analysis, the computational model must exhibit certain characteristics and prohibit others. This is especially the case for concurrent computational models, whose range of concurrency features may decisively facilitate or else obstruct analysis.

A good deal of the progress achieved by scheduling theory over the last decade has been especially directed at shedding restrictions that most reduced the expressive power of the allowable computational model. Think for example of the clash between the requirement

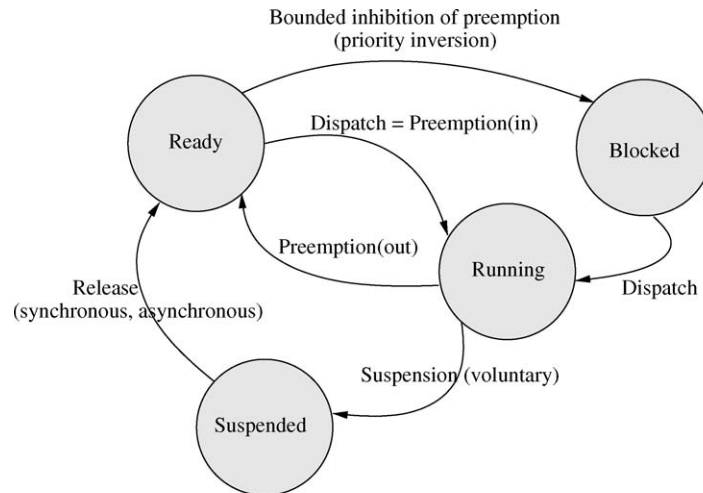

Figure 1. Scheduling states and state transitions of Ravenscar tasks. The Profile reduces the overall number of states, lessens the number of events that can trigger transitions to and from the Suspended state, and bounds the duration of the process stay in the Blocked state. Ravenscar tasks enter the Blocked state at most once per activation, and only prior to dispatching.

for process independence assumed by the initial version of rate monotonic analysis (Liu and Layland, 1973) and the producer-consumer (or signaler-waiter) relationship exhibited by most real-time systems.

The static analysis of an Ada application (and of any other systems programming language alike) which makes unrestricted use of the language standard runtime features is currently not feasible. Moreover, the potentially unbounded behavior of several tasking and other runtime calls (e.g., the dynamic binding of object-oriented programming) may make it impossible to provide acceptable bounds on execution time. Coding style rules and subset restrictions must thus be followed to ensure that all code within critical tasks be statically time-bounded, and that the execution of the tasks can be defined in terms of arrival times, deadlines, blocking times and response times.

In the following three subsections we briefly discuss some of the key restrictions that make up the Ravenscar Profile. We refer the reader to Burns et al. (2003) for the thorough presentation of the Profile.

2.3.2. Decomposition into Single-Threaded Processes

The application must be decomposed into a number of separate processes, each with a single thread of control, with clear identification of all interactions between them. Each process is implemented as a distinct Ada task with a single invocation event, followed by the operation of the process in response to that event.

The task set must be static in composition, with all tasks in the program created at the library level and typically non-terminating, thereby excluding dynamic task creation and

the formation of task hierarchies. This requirement matches current practice, as most real-time systems with integrity requirements are in effect comprised of a flat collection of non-terminating, cooperating, processes (Liu, 2000).

Ravenscar tasks are categorized as time-triggered, when they execute in response to a time event, or event-triggered, when they execute in response to other types of event, whether synchronous (i.e., generated by a running task) or asynchronous (i.e., caused by an external stimulus). If a time-triggered task receives a regular invocation event with a statically determinable rate, the task is termed periodic or cyclic. Event-triggered tasks whose activation event follows a defined arrival law are termed sporadic.

Task suspension may be either relative or absolute. Relative suspension is exposed to non-deterministic delay expiration because the delaying task may be preempted after calculating the required relative delay but before actual suspension occurs. Relative suspension is thus unable to ensure deterministic time of activation for time-triggered tasks. The Ravenscar Profile wants time-triggered tasks to use absolute suspension only, and with high-accuracy time values.

2.3.3. Restrictions on Protected Objects

The Ravenscar Profile requires that no more than one task at any one time be suspended on a closed entry barrier for each protected object used as a task synchronisation primitive. (protected object entries implement monitor procedures guarded by condition variables, with callers queuing up on them until the condition holds, akin to Dijkstra's guarded commands Dijkstra, 1975). The restriction prevents the formation of queues of tasks on an entry, with the consequent nondeterminacy of the waiting time in the queue.

The Profile allows at most one entry per protected object, to prevent multiple barriers from becoming open simultaneously as the result of a protected action and to avoid the consequent nondeterminacy of selecting which entry to service first.

In order to attain deterministic execution of task synchronisation, the evaluation of entry barriers must be free of side effects. To this end, the profile requires the barrier value to either be static or be read directly, without computation, from one of the protected object components. Applications that require composite barrier expressions can simply declare an additional Boolean value within the protected data and assign to it the result of the composite expression whenever the evaluation result may change. (This restriction forces deliberate side effects to be programmed explicitly.)

The profile, finally, disallows the current caller of a closed protected entry to be dynamically transferred to another entry queue, by which Ada's most powerful requeue statement supports condition synchronization.

These prohibitions collectively warrant deterministic task release from protected entry queues.

2.3.4. Restrictions on Dispatching

The Profile assumes use of the standard Ada `FIFO_Within_Priorities` task dispatching policy. With this policy in effect on the general language model, modifications to the ready queues occur as follows:

- When a suspended task becomes ready, it is added at the tail of the ready queue for its active priority.

- When the active priority of a ready task that is not running changes or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except when the active priority is lowered due to the loss of inherited priority, whereby the task is added at the head of the ready queue for its new active priority.[1] Under the profile restrictions, the active priority of a task changes only upon inheriting or relinquishing a ceiling priority, as the profile excludes the use of dynamic task priorities and allows it to change in no other circumstance.

- When the setting of the base priority of a running task takes effect, the task is added at the tail of the ready queue for its active priority. Under the profile restrictions, this event only occurs at task creation.

- When a task executes a delay statement that does not result in suspension, the task is added at the tail of the ready queue for its active priority.
  Under the Profile restrictions, this event would only occur upon a `delay until` statement issued with an absolute time parameter in the past, as the result of either a system or a programming error.

Each of these events is a task dispatching point, i.e., a point in time at which the runtime selects among tasks ready for execution. When a running task is preempted, the task is added at the head of the ready queue for its active priority.

2.4. Ravenscar Runtime Implementation

Figure 2 sketches the interaction between primitives and data structures in a Ravenscar runtime. The accurate characterization of the execution time of these notional primitives that we seek is greatly facilitated by the simplicity of their semantics and is crucial to precise response time analysis.

2.4.1. Time-Triggered Tasks

Time-triggered tasks call the language-level statement `delay until` to command the time of their next activation. The wake-up system may use an `Interval Timer` in place of the usual periodic clock. With the interval timer model, a down-counting timer is primed with the exact number of ticks until the expiry of the suspension interval commanded by the task at the head of the time-ordered Delay queue. When the interval timer ticks down to zero, an interrupt is generated, which results in direct or indirect transfer of control to the Delay queue processing code.

Primitive `Delay_until (Enter)` denotes the suspension service. If the required suspension interval is shorter than the time value at the head of the Delay queue, the

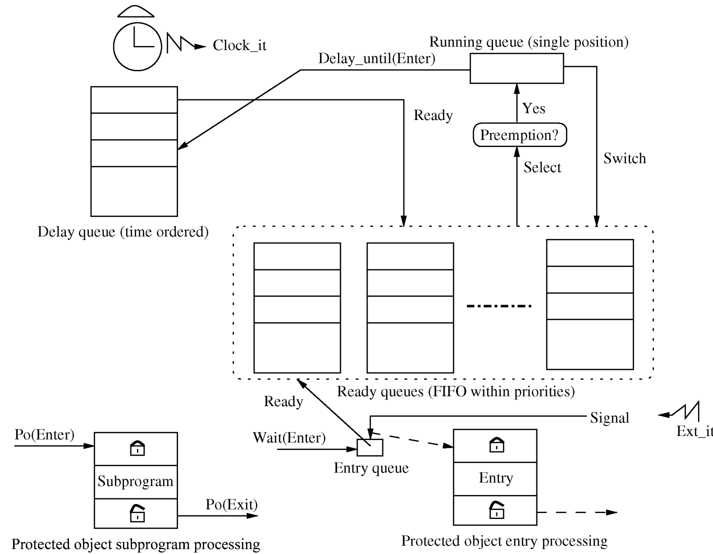

Figure 2. Interaction between Ravenscar runtime primitives and data structures. The execution of primitives with bold-faced names gives rise to task dispatching points.

primitive involves setting the down counter; otherwise, it involves traversing the time-ordered queue until the right position, for which the worst-case execution time is a function of the total number of time-triggered tasks in the system. The dual action of returning from the suspensive call is executed by the task at the time of the next activation.

The task suspension is a task dispatching point, which entails execution of primitive `Select` to dispatch (i.e., to determine the best runnable task out). If preemptive switch to a new running task is required, this is performed by primitive `Switch`.

Interrupts off the clock are serviced by primitive `Clock_it`, which determines the tasks that are due for release off the Delay queue and passes this information on to primitive `Ready`, which changes the status of the corresponding tasks to ready and places them in the appropriate queues. (This action subsumes the invocation of primitive `Delay_until(Exit)` for the readied tasks.) This event is a task dispatching point, which is treated in the same way as upon task suspension. We will refine this notion of clock handling in Section 3.3.

2.4.2. *Event-Triggered Tasks*

Event-triggered tasks make a suspending call to a protected entry with a (closed) barrier to await their next activation event. In the general case, this call results in the invocation of primitive `Wait`. In keeping with the Profile restriction, one protected object with its one protected entry is generally dedicated to each event-triggered task.

Under normal conditions, the barrier is closed at the time of invocation and the caller is enqueued on the Entry queue. If that was not the case, the event would indicate a timing or execution error in the system.

The releasing call that opens the barrier may originate, asynchronously, from an external interrupt or, synchronously, from the running task. Either event results in the caller's invocation of primitive `Signal`. Upon an asynchronous releasing event, the invocation occurs within the handler of the interrupt attached to the corresponding entry, which is located by primitive `Ext_it`. For Ravenscar applications, the immediate interrupt handler sequence, which delivers the release event, limits itself to opening the barrier, so as to bound the interference effect incurred from executing at hardware priority. The application-level treatment of the interrupt is then deferred to the execution of the event-triggered task, which normally runs at software priority. Primitive `Ext_it` includes all the housekeeping actions that the runtime requires to precede and follow execution at interrupt level.

The protected object model of Ada features two hierarchical levels of protection, in what is known as the Eggshell model (Barnes, 1998). In this model, tasks can be in one of three states in relation to a protected object: (i) outside, waiting to gain access to the object; (ii) enqueued, inside the shell, on an entry queue; and, (iii) at most one, inside the object, executing the code of an entry or of a subprogram. An outward order of service is placed on task calls to the protected object: existing calls waiting on an entry queue take precedence over new calls; any new call cannot even evaluate the entry barrier until any call currently executing within the protected object completes, along with any other enqueued calls that can subsequently be processed.

The strength of the Eggshell model is that it guarantees that no change may occur to the barrier condition after a task has gained access to the protected object. The model is crucial for determinism when the full language is in use, but it makes static analysis of the entry queues processing prohibitive.

In the Eggshell model, the entry queue of a protected object is notionally represented as two priority-ordered queues: the Wait queue and the Signaled queue. When a caller to a protected object entry is suspended on a closed barrier, primitive Wait is called, which enqueues the caller on the Wait queue. Subsequently, the runtime transfers ownership of the protected object lock to the task at the head of the Signaled queue (if any). Transfer is achieved by raising this task priority to the protected ceiling value and performing a direct context switch to it. When this task gets control, it re-evaluates the barrier (which may have changed in the meanwhile) and, if open, it continues to execute the corresponding entry. Else, as the barrier expression evaluates to closed, the task enqueues itself again on the Wait queue. In the case the Signaled queue was empty, the caller simply releases the lock performing the same unlock epilogue code as for a protected object with no barrier (cf. Section 2.4.3).

In the general language model, the state of all barriers in a protected object with entries needs to be re-evaluated, and the entry queues serviced, after every execution of a protected subprogram or entry. The runtime achieves this by invoking primitive `Signal` at the end of each entry service and protected procedure. Primitive `Signal` transfers the current queue of waiters from the Wait queue, excluding the head element, to the Signaled queue and performs a direct context switch to the task at the head of the Wait queue. This task then behaves exactly as the head task released off the Signaled queue.

The Ravenscar restrictions cater for drastic simplification of the entry queue implementation model. The Profile requires that no more than one event-triggered task can be on any protected object entry queue. The time bound to the entry queue service will then be

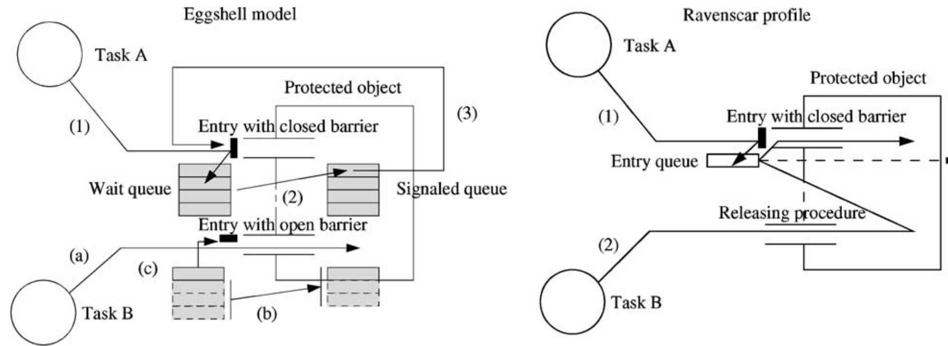

Figure 3. Breakdown of the *Eggshell* model implementation of entry queue servicing in contrast with the Ravenscar simplified implementation. For Eggshell model: (1) task A invokes an entry call with a closed barrier and queues on the Wait queue; (2) control transfers to the task at the head of the Signaled queue; (3) The head task evaluates the barrier; (a) Task B executes an entry with an open barrier; (b) The Wait queue, minus the head, is moved to the Signaled queue; (c) The head task evaluates the barrier. for Ravenscar Profile: (1) Task A invokes an entry call with a closed barrier and queues on the Entry queue (task dispatching point); (2) Task B executes the releasing procedure with the Proxy Model, while task A is moved to the Ready queue (task dispatching point)

only that of inserting the task at the head of the Wait queue, denoted by `Wait(Enter)`. Furthermore, as the Signaled queue will always be empty, `Wait(Enter)` gives rise to a task dispatching point, always incurring the selection of a new task from the Ready queues (`Select`), and the context switch to it (`Switch`). The single-waiter restriction also facilitates the adoption of the Proxy Model, by which the `Signal` caller evaluates the barrier and (if open) executes the protected entry on behalf of the waiting task, while the runtime calls primitive `Ready` to place the waiting task on the Ready queue.

This strategy spares the waiting task the execution of `Wait(Exit)`. It also saves one context switch from the signaling task (were it to relinquish the ceiling lock upon opening the barrier and leaving the protected object) to the waiting task (were it to acquire ceiling priority upon leaving the Wait queue and being transferred ownership of the protected lock).

There occurs a task dispatching point as both the signaling and the waiting tasks leave the protected object.

Figure 3 contrasts the general *Eggshell* implementation of entry queue servicing with the simplified version permitted by the Ravenscar Profile.

2.4.3. Other Runtime Primitives

Primitives `Po(Enter)` and `Po(Exit)` control respectively the entrance into and the departure of any tasks from protected operations that do not involve entry queuing, and include the raising, respectively the lowering, of the calling task active priority. The latter event is a task dispatching point.

The Ravenscar profile adopts the priority ceiling protocol (Goodenough and Sha, 1988; Sha et al., 1990). Each protected object is thus statically assigned a ceiling priority at least as great as the highest priority of all its calling tasks. Calling tasks acquire the ceiling priority for the duration of their execution within the object.

Table 1. Required bounds for Ravenscar runtime primitives.

Primitive	Invoked by	With effect on
`Delay_until(Enter)`	Time-triggered tasks	Self
`Clock_it`	Runtime	Time-triggered tasks
`Ready`	Runntime	Any tasks
`Select`	Runtime	Any tasks
`Switch`	Runtime	Any tasks
`Po(Enter)`	Any task	Self
`Po(Exit)`	Any task	Self
`Wait(Enter)`	Event-triggered tasks	Self
`Signal`	Any task and protected interrupt handler	Event-triggered tasks
`Ext_it`	Runtime	Event-triggered tasks
`Defer_Preemption`	Runtime	Any tasks

The implementation of this protocol is straightforward. All it takes is to raise the active priority of the task enabled to execute a protected operation to the ceiling priority of the object, and to restore the old priority value on exit from the object. Accordingly, primitive `Po(Enter)` moves the task from the Ready queue for its current active priority to the head of the Ready queue for its new active priority, which equals the ceiling priority of the target protected object. Primitive `Po(Exit)` does the converse.

2.4.4. Summary

Table 1 lists all primitives that a Ravenscar runtime must implement and whose precise worst-case execution time bounds are required for off-line scheduling analysis. For every primitive, the table indicates the nature of the caller (task, runtime) and the task on which the primitive takes effect.

The notional primitive `Defer_Preemption` listed at the bottom of the table characterizes the time interval during which the runtime may disable interrupts, thereby potentially deferring preemption.

In the following section we discuss the factors that need to be accounted for, and be suitably documented, in the determination of the execution-time bounds for these primitives.

3. Precise Response Time Analysis

3.1. Foundations

Response time analysis stipulates that the worst-case response time of a process be defined as the longest elapsed time it takes for that process to complete its most demanding set of activities in response to a single activation event occurring under maximum contention from the rest of the system. The worst-case response time of any task τ_i does, thus, result from the summation of three additive components:

1. The worst-case execution time of task τ_i, C_i, which is defined as the total time cost of all τ_i's sequential blocks of execution that lay in the statically-determined worst-case path enclosed within the task's execution profile, including the time cost of the runtime services required for the support of that execution (Puschner and Burns, 2000).

2. The interference incurred by τ_i, I_i, caused by the occurrence of preemptive execution of higher-priority tasks and higher-priority runtime services (such as the serving of higher-priority interrupts) incurred during τ_i's ready period.

3. The *blocking* experienced by τ_i, B_i, which occurs as a due release of τ_i is delayed by priority inversion effects, whether upon temporary inhibition of interrupts or by effect of `Ceiling_Locking`. The worst-case blocking is determined as the largest possible deferral effect incurred from any of the two sources. (We shall discuss this further in section 3.5.)

Component C_i and B_i are analysed reflecting on the call sequence made by tasks in the application. Component I_i, instead, is analysed assuming the execution of τ_i to occur under the notional concept of critical instant (Liu and Layland, 1973), which requires that:

- all time-triggered tasks be disjointly released at time $t_0 = 0$,
- all external interrupts be disjointly raised at time $t_0 = 0$ and arrive at their maximum frequency,
- all event-triggered tasks be disjointly released off their entry queue at time $t_0 = 0$.

Care must be taken to avoid incurring excessive pessimism in the determination of these terms, as this may obliterate the usefulness of the analysis. The objective of this paper serves this goal well because the higher the accuracy of the accounted runtime overhead components, the lower the pessimism and the greater the margin for useful application-level processing.

The response time equation is based on recurrence relations in which task τ_i's response time, R_i, is expressed as a monotonically increasing summation term. The recurrence converges as long as the overall processor utilisation is not greater than 1. The schedule of the task set is feasible as long as R_i falls within the deadline for every task τ_i in the system.

$$R_i = R_i^n = B_i + C_i + \mathrm{I}_i^{R_i^{n-1}} \quad (n > 1) \tag{1}$$
$$R_i^1 = B_i + C_i$$

The R_i value so obtained represents the worst-case time of completion of an activation of τ_i. The following section discusses which runtime overhead components contribute to the individual terms of the equation.

3.2. Runtime Metrics

A very attractive feature of the Ravenscar Profile is that its runtime can be implemented by a small, efficient, reliable and certifiable kernel, by which term we mean the target-dependent part of a Ravenscar runtime.

Table 1 shows that, overall, the runtime implementation needs about 10 (notional) primitives. To date, at least two such kernels have been developed and industrially deployed: the Aonix ObjectAda/Raven technology (Dobbing and Romanski, 1999); and the Technical University of Madrid Open Ravenscar Kernel (ORK) (de la Puente et al., 2000a).

Accurate analysis of the timing behaviour of Ravenscar applications requires a set of metrics that precisely characterize the runtime overhead that may be incurred on execution. This characterization is greatly facilitated by the simplicity and intrinsic determinism of Ravenscar kernels.

Annex D of the Ada language specification (Ada, 95, 2000) lists a number of metrics optionally required of implementations. Such metrics are a useful initial basis. Yet, systematic approaches to scheduling analysis for time-critical systems, for example, Vardanega (1999), have shown that modern analysis methods require finer-grained information about the runtime behavior (Burns and Wellings, 2001). In this section we explore the issue further and lay down requirements and put forward definitions for a set of metrics that enable precise response time analysis. The specification of the resulting definitions is the central contribution of this paper. We shall return to this point in Section 4.

The metrics we discuss concern the execution of time-triggered and event-triggered tasks, protected procedures used in the way of interrupt handlers and timers for both periodic clocks and absolute delays. The corresponding set of primitives covers all of the concurrency constructs allowed by the Ravenscar Profile, which, as we asserted in Section 1, user experience has shown to be a sufficiently powerful computational model for most real-time applications.

3.2.1. Modeling Time-Triggered Tasks

In order for time-triggered tasks to be statically analysable, they have to belong in the category of periodic, i.e., they must receive a regular activation event with a statically assigned rate. A number of significant events contribute runtime overhead to the execution of these tasks. Figure 4 illustrates such events, which collectively constitute the set of factors to include in the resolution of response time equation (1). The time interval denoted $R - I$

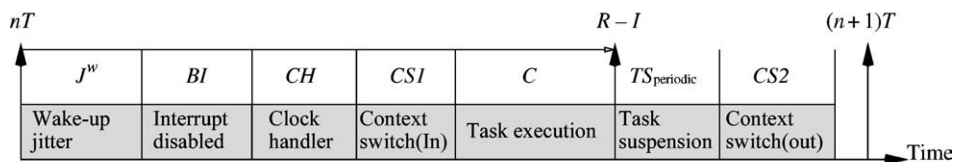

Figure 4. Events occurring in one time-triggered task execution.

in Figure 4 represents the task response time minus the interference effect incurred from events independent from the sequence of interest. We shall progressively characterize the missing interference component in the remainder of the discussion.

3.2.1.1. Wake-up jitter. Time-Triggered Ravenscar tasks use absolute time values in the `delay until` statement to program their next wake-up time. The actual alarm setting may attain differing degrees of timing accuracy. A coarse clock tick is the prime source of inaccuracy, which may result from coarse software representation of the hardware clock, but also from relative programming of fine-grained hardware timers (Zamorano et al., 2001). In the latter case, the absolute time parameter of the `delay until` statement translates into a relative delay (e.g., in the number of clock ticks that best approximates the required length of suspension), which may cause jitter on the actual time of activation. The worst-case delay jitter then equals the difference between the upper bound and lower bound of the suspension interval calculation error.

In some hardware architectures, the wake-up time calculation may involve variable-duration operations, such as divide and multiply, whose execution time may be difficult to bound (INTEL, 1989; TEMIC, 1996). Processors with hardware timers implemented on wide registers (64-bit or wider) can always use absolute time and therefore avoid incurring clock inaccuracies (Zamorano et al., 2001).

Required metric: Where the runtime does not directly support absolute wait time, a bound must be provided on the wake-up time calculation error.

Figure 4 denotes this overhead component as J^W, which may well depend on the value of the demanded suspension interval.

3.2.1.2. Interrupt disabled. The kernel may protect its own execution of critical sections by temporarily disabling the acknowledgment of interrupts from the clock and all other external sources. The elapsed time that the kernel may execute with the acknowledgment of interrupts disabled is one of the two contributing factors to the blocking experienced by Ravenscar tasks, the other arising from the use of the priority ceiling protocol (cf. Section 3.5).

Required metric: The runtime implementor shall document the maximum duration of the elapsed time of kernel execution with interrupts disabled.

Figure 4 denotes this overhead component as BI.

3.2.1.3. Clock handler. The clock handler, which we have associated to primitive `Clock_it` (cf. table 1), executes when the clock interrupt is acknowledged by the processor. Modeling the clock handler overhead depends on the approach adopted for the implementation of suspension intervals as well as for the support of the fine-grained time reference that the Ada specification of function `Ada.Real_Time.Clock` requires.

Required metric (preliminary): The runtime implementor shall document the maximum execution time of the primitive associated with the handling of the clock, which Table 1 and Figure 2 denote as `Clock_it`.

With reference to Figure 4 we may, for the moment, stipulate:

$$CH = \texttt{Clock_it} \tag{2}$$

We shall refine the definition of this metric in Section 3.3.

Since the runtime may need to handle more than one type of clock interrupt, we cannot tell *a priori* whether the handling also included the readying of a time-triggered task from the Delay queue. The corresponding overhead term (i.e., `Ready`) will therefore be accounted for in the component denoted $CS1$ in Figure 4.

3.2.1.4. Context switch(In). The clock handler moves the time-triggered task off the Delay queue to the Ready queue, thus giving rise to a task dispatching point. From the standpoint of Figure 4, we assume that this results in the switch to the time-triggered task, because equation (1) accounts as interference all other events in which the task does not have the highest priority among those runnable.

Required metric: The runtime implementor shall document the maximum execution time of the three primitives associated with the handling of a task dispatching point, which Table 1 and Figure 2 denote as `Ready`, `Select` (which may be linear in the number of Ready queues and positions within them) and `Switch`.

With reference to Figure 4 we may therefore stipulate:

$$CS1 = \texttt{Ready} + \texttt{Select} + \texttt{Switch} \tag{3}$$

3.2.1.5. Task execution. The response time equations require the task execution component proper to be determined as the time cost of all the sequential blocks of execution that lay in the worst-case execution profile of the task, in addition to the time cost of the runtime services required for the support of that particular execution (which effectively amounts to the cost of invoking protected subprograms that do not involve queuing). This term does not include the interference incurred from preemption. The worst-case execution time of the task is just one of the three additive components that determine its response time.

Tools and methods exist that assist the developer in determining the worst-case execution time of Ravenscar tasks. (Cf. e.g., Puschner and Burns, 2000).

3.2.1.6. Task suspension. Upon completion of the current activation, the task suspends itself until the time of the next activation. At some point during its execution, the task computes the value of the next activation time. As we have seen earlier in Section 2.4, the execution of program statement `delay until` results in the invocation of primitive `Delay_until(Enter)`. That primitive moves the task off the running state and places it in the Delay queue (cf. Figure 2). The time required by the execution of this primitive depends on the position where the task must be inserted in the (time-ordered) Delay queue. The worst case insertion time depends on the maximum queue length, which is equal to the number of time-triggered tasks in the application.

Required metric: The runtime implementor shall document the time bound for the execution of primitive `Delay_until(Enter)`, possibly making it parametric to the length of the queue.

With reference to Figure 4 we may therefore stipulate:

$$TS_{\text{periodic}} = \texttt{Delay_until(Enter)} \tag{4}$$

Figure 4 shows that we account this term as interference from the outgoing task on the response time of the next task to run.

3.2.1.7. Context switch(Out). Upon termination of the current activation of the task and its placement in the Delay queue, a task dispatching point occurs, whereby the runtime must select a new runnable task (which may also be an idle task if no application-level task were ready to execute) and perform a switch to it.

This sequence of action, which contributes to the interference term in the response time equation of the subsequent task to run, encompasses execution of primitives `Select` and `Switch`. The corresponding metric is subsumed by the one required for the dual event: `Context swith(In).`

With reference to Figure 4 we may therefore stipulate:

$$CS2 = \texttt{Select} + \texttt{Switch} \tag{5}$$

3.2.1.8. Response time for time-triggered tasks. Assuming that all terms from Figure 4 are significant, the response time equation for a time-triggered task is as follows, where notation $j \in hp(i)$ denotes that task τ_j's priority is greater than τ_i's:

$$\begin{aligned} R_i^n &= B_i + CS1 + C_i \\ &\quad + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{n-1} + J_j^A}{T_j} \right\rceil (CS1 + C_j + TS + CS2) + I_{\text{clock}}^{R_i^{n-1}} + I_{\text{extint}}^{R_i^{n-1}} \\ R_i^1 &= B_i + CS1 + C_i \\ R_i &= R_i^n + J_i^W \quad (i \in \text{periodic}) \end{aligned} \tag{6}$$

Note the refinements we have introduced with respect to equation 1:

- term $CS1$ has been added to the execution time component of every task in the system (i.e., C_i and $C_j \forall j \in hp(i)$).

- term $I_i^{R_i^n}$ has been substituted by the more elaborate equation:

$$I_i^{R_i} = \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j^A}{T_j} \right\rceil (CS1 + C_j + TS + CS2) + I_{\text{clock}}^{R_i} + I_{\text{extint}}^{R_i}$$

where T_j is the period or the minimum inter-arrival time of task τ_j, as applicable, and J_j^A denotes the variability in the arrival of the activation event (called activation jitter) that might occur if τ_j was sporadic and the activation event was synchronous; J^A is an application-dependent value, on which the runtime has no direct impact

- components *TS* and *CS*2, which for now we have only characterized for time-triggered tasks but has an equivalent characterization for event-triggered tasks, accounts the interference overhead incurred on the suspension of higher-priority tasks.

- term $I_{\text{clock}}^{R_i}$, which we shall discuss in Section 3.3, represents the general clock handling overhead as it builds up from the elementary *CH* component.

- term $I_{\text{extint}}^{R_i}$, which we shall discuss in Section 3.4, denotes the interference effect incurred from the handling of external interrupts other than the clock.

- term J_i^W, finally, was defined in Section 3.2.1.1 as wake-up jitter.

Overhead component *BI*, which denotes the maximum duration of the elapsed time of kernel execution with interrupts disabled, is accounted for the determination of term B_i, which we shall discuss in Section 3.5. As the Ravenscar Profile prescribes the use the `Ceiling_Locking` protocol, the blocking effect arising from BI and that from priority inversion are not cumulative.

3.2.2. Modeling Event-Triggered Tasks

In order for event-triggered tasks to be statically analysable they must belong in the category of sporadic. Their activation event must thus follow a defined arrival law. These tasks await their next activation event by making a call to the closed barrier of a protected object entry. The activation event takes the form of a call to a protected procedure that opens the barrier. The call can be made synchronously by a running task or asynchronously by the runtime-level handler of the interrupt attached by the application to the protected procedure.

The Proxy Model discussed in Section 2.4 allows the task that opens the barrier to execute the protected entry on behalf of the waiting event-triggered task. The Ravenscar runtime implementation in the GNAT compiler (Ada Core Technologies, 2000), which is integrated with ORK (de la Puente et al., 2000b), uses the Proxy Model. With this approach, once the releasing protected procedure is executed, the value of the barrier is set to open and the barrier is re-evaluated accordingly, the event-triggered task is moved off the entry queue onto the Ready queue and the code of the protected entry is executed in the context of the interrupt handler or task delivering the releasing event.

Figure 5 assumes the scenario in which the releasing call is made asynchronously and depicts the corresponding sequence of events. The time interval denoted $R - I$ in the figure represents the task response time minus the interference effect incurred from events independent from the sequence of interest.

The events that concur to the execution of an event-triggered task have the same interpretation for response time analysis as the ones we encountered in the execution of as

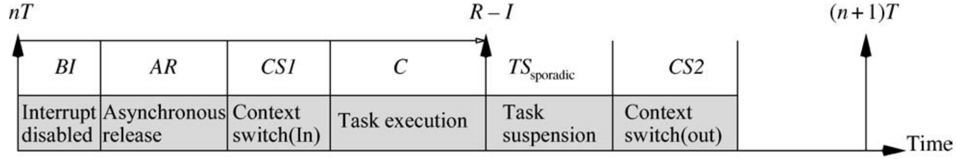

Figure 5. Events occurring in one event-triggered task execution with asynchronous release.

time-triggered task, with minor adaptations that reflect the different nature of the task.

3.2.2.1. Interrupt handler (asynchronous release). The interrupt handler, the initiation of which we have associated to primitive `Ext_it` (cf. Table 1), executes when an unmasked external interrupt is acknowledged by the processor. The `Ext_it` primitive locates the protected procedure that the application has attached to the interrupt, and passes this information on to the subsequent invocation of primitive `Signal`.

With the Ravenscar restrictions in effect, the cost of `Signal` includes: the execution of the preamble that seizes the lock of the protected object by using primitive `Po(Enter)`; the code of the protected procedure itself, which usually just opens the barrier of the relevant entry; and the epilogue that releases the lock by using primitive `Po(Exit)`.

Required metric: The runtime implementor shall document the maximum execution time of the two primitives associated with the invocation of protected subprograms, which table 1 designates as `Po(Enter)` and `Po(Exit)`, possibly making it parametric to the range of supported priorities.

With this provision, we may therefore stipulate:

$$\texttt{Signal} = \texttt{Po(Enter)} + \mathit{wcet}(\text{procedure}) + \texttt{Po(Exit)} \tag{7}$$

The Proxy Model actually causes additional operations to be performed in the context of the releasing call: evaluating the barrier, executing the code of the protected entry and moving the event-triggered task from the entry queue to the Ready queue. This chain of events gives rise to a task dispatching point, with the selection of a task from those ready. For the purpose of this discussion, we assume that the event-triggered task in question is actually selected, while we account as interference the execution of all higher-priority ready tasks that may occur before it.

Required metric: The runtime implementor shall document the time bound for the execution of all actions involved in the execution of an external interrupt handler under the Proxy Model.

With reference to Figure 5 we may therefore stipulate:

$$AR = \texttt{Ext_It} + \texttt{Signal} + \mathit{wcet}(\text{entry}) \tag{8}$$

where component $wcet$(entry) bounds the barrier evaluation and the execution of the entry service under the Proxy Model. As term $CS1$ already includes the overhead incurred from readying the task, we do not include component `Ready` in the resolution of term AR.

We shall discuss the attribution of this runtime overhead component in Section 3.4.

3.2.2.2. Synchronous release. The sole aspect that differentiates the synchronous release of an event-triggered task from the asynchronous release we have just analysed resides in the nature of the releasing call. The corresponding overhead factor, which we denote SR, can be easily obtained from equation 8 by dropping component `Ext_It` from the right-hand side of the equation:

$$SR = \texttt{Signal} + wcet(\text{entry}) \tag{9}$$

Component SR may either account as interference from higher-priority tasks, where it would be collated in term C_j of response time equations (6) and (11), or else as blocking from priority inversion, which we discuss in Section 3.5. In both cases, the overall term shall also include the cost of executing the application-level code of the protected entry on which the event-triggered task was enqueued.

3.2.2.3. Task suspension. Similarly to time-triggered tasks, event-triggered tasks suspend themselves voluntarily until the next activation. Upon the call to the closed barrier of the protected entry, the runtime moves the event-triggered task off the running state to the entry queue attached to the protected object. In Section 2.4, we have used primitive `Wait(Enter)` to capture this event. The restrictions enforced by the Ravenscar Profile prevent entry calls made by distinct tasks from queuing up in the same queue. This reduces the runtime overhead of the suspension operation, while adding to the determinism of insertion and release.

Required metric: The runtime implementor shall document the time bound for the execution of primitive `Wait(Enter)`.

With reference to Figure 5 we may therefore stipulate:

$$TS_{\text{sporadic}} = \texttt{Wait(Enter)} \tag{10}$$

3.2.2.4. Response time for event-triggered tasks. Assuming that all terms from Figure 5 are significant, the response time equation for event-triggered tasks is as follows:

$$\begin{aligned} R_i^n &= B_i + CS1 + C_i \\ &\quad + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{n-1} + J_j^A}{T_j} \right\rceil (CS1 + C_j + TS + CS2) + I_{\text{clock}}^{R_i^{n-1}} + I_{\text{extint}}^{R_i^{n-1}} \\ R_i^1 &= B_i + CS1 + C_i \end{aligned} \tag{11}$$

Component BI from figure 5 is taken into account in the determination of term B_i of the equation. Term TS resolves into either TS_{periodic} or TS_{sporadic} according to the nature of task τ_j.

As the event-triggered task τ_i experiences no wake-up jitter, no term J_i^W has to add to R_i^n, whereas activation jitter still affects the calculation of the interference effect from higher-priority event-triggered tasks with synchronous activation. If component J^W in Figure 4 is negligible (as the runtime offers an accurate wake-up service) then response time equation (6) for time-triggered tasks and equation (11) for event-triggered tasks are identical.

3.3. Modeling Clock Interrupts

Ravenscar kernels must support two sources of clock interrupts: those associated to the release of time-triggered tasks and those needed to maintain the language-level time reference returned by function `Ada.Real_Time.Clock`.

The determination of the Ravenscar runtime overhead must therefore account for: demanded interrupts, which arise from the programming of the interval timer; and periodic interrupts, which incur from maintaining the software representation of the clock register.

Notation T_{periodic} denotes the fixed rate of the clock. The width of the down counting register of the hardware timer and the corresponding frequency of update, which is part of the hardware setting, provide an upper bound for T_{periodic}.

Equation (12) provides the worst-case bound on the number of periodic interrupts that may cause interference on any task's response time:

$$I_{\text{periodic}}^{R_i} = \left\lceil \frac{R_i}{T_{\text{periodic}}} \right\rceil CH_{\text{periodic}} \tag{12}$$

where CH_{periodic} is the cost of handling a single interrupt off the periodic clock.

The worst-case bound on the number of demanded interrupts that may cause interference on the same task ready period results from the summation of two terms: the disjoint activation of all lower-priority time-triggered tasks that occurs at the critical instant; and the raising of as many interrupts off the interval timer as activations of higher-priority time-triggered tasks throughout the task ready period. Equation (13) provides this bound:

$$\begin{aligned} I_{\text{demanded}}^{R_i} = & \sum_{j \in hp_{\text{periodic}}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times CH_{\text{demanded}} \\ & + \sum_{t \in lp_{\text{periodic}}(i)} CH_{\text{demanded}} \end{aligned} \tag{13}$$

where CH_{demanded} is the cost of handling a single interrupt off the interval timer and notation $j \in lp(i)$ denotes that task τ_j's priority is lower than τ_i's.

Equation (13) captures precisely the level of pessimism embodied in the notion of critical instant defined in Section 3.1. Gentler definitions exist, which would require adaptations to the equation: for example, one in which all time-triggered tasks were simultaneously, as opposed to disjointly, released from the Delay queue at the critical instant.

On reflection of this discussion, we may now refine equation (2), which stipulates the required runtime metric for clock handling overhead, to:

Required metric: The runtime implementor shall document the maximum execution time of the primitive(s) associated with the handling of the internal clock(s), which raise interrupts that we have categorised as demanded and periodic:

$$\begin{aligned} \text{CH}_{\text{demanded}} &= \texttt{Clock_it}_{\text{demanded}} \\ \text{CH}_{\text{periodic}} &= \texttt{Clock_it}_{\text{periodic}} \end{aligned}$$

In force of equations (12) and (13), we may thus stipulate:

$$I^{R_i}_{\text{clock}} = I^{R_i}_{\text{periodic}} + I^{R_i}_{\text{demanded}} \tag{14}$$

Possible optimisations to the implementation entailed by this double-clock model exist. For example, if there are enough demanded interrupts, the kernel may do away with the periodic ones and yet be able to maintain an accurate clock value. In this case, the analysis model should try to bound the actual number of periodic interrupts required within the interval of observation.

ORK for ERC32 targets (de la Puente et al., 2000a; ATMEL, 2003), deliberately uses two distinct hardware timers for the two sources of interrupt, because of the difficulty of keeping the drift-less monotonic clock required by the Ravenscar Profile with just one interval timer (Zamorano et al., 2001).

3.4. Modeling Other Interrupts

Program-level interrupt handlers execute at a priority level not inferior to the hardware interrupt priority. Task τ_i may therefore incur interference as event-triggered tasks are released upon the arrival of external interrupts.

`System.Interrupt_Priority` specifies the range of hardware priority level. Ordinary tasks execute at software priority levels, within the `System.Priority` range. The two ranges are contiguous, with the former at the high end.

The release of a low-priority event-triggered task off its entry queue may cause an interference effect on higher-priority tasks. With the Proxy Model in effect, the task being released neither inherits the ceiling priority of the protected object nor executes the entry on which it was enqueued. Hence, this situation configures as classical interference instead of as priority inversion. Equation (15) models this effect as interference from higher-priority interrupt handlers:

$$I^{R_i}_{\text{extint}} = \sum_{k \in hp_{\text{interrupt}}(i)} \left\lceil \frac{R_i}{T_k} \right\rceil \times AR \tag{15}$$

where AR is the term we have bounded with Equation (8) and T_k is the minimum inter-arrival time of the event-triggered task released by the corresponding interrupt.

As the base priority of task τ_i may itself be in the Interrupt_Priority range, some hardware interrupts could effectively have a lower priority than it. In such a case, the lower-priority hardware interrupts would be masked and will be not acknowledged during the execution of the task.

3.5. Evaluating Blocking

Raising the active priority of tasks with low base priority may make them execute in preference to higher-priority tasks. This event is known as priority inversion (Cornhill and Sha, 1987; Locke et al., 1988). The priority ceiling protocol optimally bounds the duration of these situations, which response time analysis models as blocking with the B component of equation (1) and its derivatives (6) and (11).

Blocking may also occur as the runtime temporarily disables interrupts to protect the execution of its own critical sections. In Ravenscar applications, the triggering of an interrupt represents an activation event for an event-triggered or a time-triggered task: the time interval during which interrupts are disabled defers the task dispatching point that follows the readying of the task. If that task has higher priority than that of the currently running task, the deferral is a case of priority inversion.

These two forms of priority inversion are not cumulative: they both defer the very first preemption from the higher-priority task; as soon as that preemption has taken place, no other circumstance will ever lead the higher-priority task to give way to a lower-priority task.

The longest time that task τ_i may experience blocking from priority inversion is thus the maximum between the longest elapsed time of kernel execution with disabled interrupts (cf. term BI and its required metric in Section 3.2.1.1) and the longest protected operation invoked by a lower-priority task on a protected object with higher ceiling priority, which, for task τ_i, we denote B_{pc_i}. Hence, we have:

$$B_i = \max_i\{BI, B_{pc_i}\} \tag{16}$$

In order to determine the value to assign to term B_{pc_i} we must characterize the runtime overhead incurred on execution of protected operations. The application-level cost of the protected operation, which adds to that overhead, will be determined by normal timing analysis techniques (Puschner and Burns, 2000).

Two kinds of protected objects exist in the Ravenscar Profile: entry-less protected objects and single-entry protected objects.

3.5.1. Entry-Less Protected Object

Protected objects of this kind implement mutually-exclusive access to shared data. The worst-case execution time of their protected subprograms results from the summation of the the worst-case execution time of the protected operation and the overhead from primitives Po(Enter) and Po(Exit) listed in Table 1.

The blocking potential of entry-less protected object subprograms may thus be expressed as:

$$B_{\text{elps}} = \texttt{Po(Enter)} + wcet(\text{subprogram}) + \texttt{Po(Exit)} \tag{17}$$

where the $wcet$(subprogram) component bounds the worst-case execution time of the application-level code of the subprogram.

3.5.2. *Single-Entry Protected Object*

Protected objects of this kind support data-oriented synchronisation between tasks. As the name tells, these objects provide a single entry in addition to protected subprograms. Event-triggered tasks invoke this entry to program their next activation event.

3.5.2.1. Protected entry. Under the Proxy Model, the execution of the entry cannot give rise to blocking, other than if the barrier were open, which would only occur upon a timing or execution error of the application. In the nominal situation, event-triggered tasks suspend themselves on a closed barrier by entering the queue associated to the entry. With the Proxy Model in effect, the code of the entry will then be executed by the task that opens the barrier. Irrespective of whether the opening of the barrier occurs synchronously or asynchronously, the code of the entry is thus executed without the priority of the calling task being raised to the ceiling of the object.

3.5.2.2. Protected procedures. The execution of protected procedures incurs blocking on tasks with priority higher than the calling task but lower than the ceiling priority of the protected object.

In Ravenscar applications, the protected procedure(s) of this object typically open(s) the barrier associated with the entry. The Proxy Model causes the calling task to execute the code of the entry on behalf of the event-triggered task waiting on the entry queue. The blocking potential of a releasing protected procedure is thus bounded by the SR component described in equation (9), plus the cost of readying the task:

$$B_{\text{rp}} = SR + \texttt{Ready} \tag{18}$$

This kind of protected object can also provide mutually-exclusive access to shared data. It may thus provide protected subprograms that do not explicitly open the barrier. In this case, the $wcet$(entry) term of the SR defining equation must not include the execution of the entry service, but only the evaluation of the barrier, which follows every execution of protected procedures of protected objects with entry. The blocking potential of these procedures may thus be expressed as:

$$B_{\text{nrp}} = B_{\text{elps}} + wcet(\text{barrier_evaluation}) \tag{19}$$

The case may also occur whereby tasks, which may invoke operations of protected objects with procedures designated as interrupt handlers, have base priority in the `Interrupt_Priority` range. The ceiling priority of the protected object may in this case turn out to be higher than the hardware priority of the corresponding interrupt. As a consequence, the execution of the protected interrupt handler would cause blocking on tasks with priority higher than the hardware interrupt but lower than the ceiling of the object. The blocking potential of the protected interrupt handler is bounded by the *AR* component resolved in equation (8) plus the cost of readying the event-triggered task:

$$B_{\mathrm{pih}} = AR + \texttt{Ready} \tag{20}$$

3.5.2.3. Protected functions. Protected functions cannot change the state of protected objects, which includes the value of the barrier. Hence, the barrier need not be evaluated after the execution of protected functions. It follows that the runtime operations involved in the execution of protected functions belonging to protected objects with one entry are exactly the same as those of subprograms of entry-less protected objects, which we bounded with equation (17).

3.5.3. Bounding Priority Ceiling Blocking

Equations (17)–(20) supply all the information we need to resolve the B_{pc_i} component of Equation (16). The worst-case priority-ceiling blocking potential incurred by task τ_i is bounded by the maximum of all protected subprograms of protected objects with ceiling $k \in hp(i)$ invoked by tasks with base priority $j \in lp(i)$:

$$B_{pc_i} = \max_{\{j \in lp(i) \wedge k \in hp(i)\}} \left\{ B_{elps_{k_j}}, B_{rp_{k_j}}, B_{nrp_{k_j}}, B_{pih_{k_j}} \right\} \tag{21}$$

Table 2 provides a synoptic view of all the execution components that contribute priority-ceiling blocking potential to the response time of Ravenscar tasks. These components

Table 2. Protected operations that contribute blocking potential to Ravenscar tasks.

Execution component		Defined by		
Symbol	Meaning	Metric	Equation	Used in equation
B_{elps}	Any subprogram	3.5.1	(17)	(21)
B_{rp}	Releasing procedure	3.5.2	(18)	(21)
B_{nrp}	Non-releasing procedure	3.5.2	(19)	(21)
B_{pih}	Interrupt handler	3.5.2	(20)	(21)
AR	Asynchronous release	3.2.2.1	(8)	(20)
SR	Synchronous release	3.2.2.2	(9)	(18)

Table 3. Synopsis of all runtime overhead components that contribute to the response time of Ravenscar tasks.

Runtime overhead component		Defined by		
Symbol	Meaning	Metric	Equation	Used in equation
B	Blocking	3.5	(16)	(6), (11)
BI	Disabled interrupts	3.2.1.2		(16)
J^W	Wake-up jitter	3.2.1.1		(6)
$CS1$	Context switch(in)	3.2.1.4	(3)	(6), (11)
$CS2$	Context switch(out)	3.2.1.7	(5)	(6), (11)
TS	Time-triggered task suspension	3.2.1.6	(4)	(6), (11)
	Event-triggered task suspension	3.2.2.3	(10)	
I_{clock}	Clock handling	3.3	(14)	(6), (11)
I_{extint}	Other interrupt handling	3.4	(15)	(6), (11)

originate synchronously within the execution of the running task; therefore, they also contribute to the worst-case execution time bound (term C) of the calling task.

3.6. Putting it all Together

Table 3 summarises the whole set of runtime factors that contribute timing overhead to the execution of Ravenscar tasks, and which we have used in constructing the corresponding response time equations (6) and (11).

4. Evaluation

The Ravenscar Profile restrictions dramatically simplify the implementation of the supporting kernel. An immediate benefit of this simplification should be a tangible improvement in execution performance, chiefly in terms of ample reduction in runtime overheads.

In order to prove this point, we related the runtime overheads incurred on two comparable kernels: a Ravenscar-specific one, i.e., ORK for ERC32 targets (de la Puente et al., 2000a; ATMEL, 2003), and a general-purpose real-time alternative, that is, the RTEMS release for the same target (On-Line Applications Research, 2003). Both kernels are open-source products that can be used with the GNAT Ada compiler (Ada Core Technologies, 2000), the same version of which (v3.13p) was used for our measurements.

As the RTEMS-based platform was not Ravenscar compliant, we could not directly apply our fine-grained metrics to it. As that platform supported the full Ada model, however, we were able to depart from the coarse-grained metrics defined by the annexes C and D of the current Ada language specification (Ada, 95, 2000). Of them, we only chose those that had a parallel under the Ravenscar Profile, so that we could express them in terms of the metrics described in this paper (cf. Table 3).

We also added two further common metrics that account for context switch time and interrupt lateness, the latter of which reduces to protected interrupt handler lateness under the Ravenscar Profile.

Where applicable, the code attached to each base comparison metric denotes annex, chapter and clause of the corresponding definition in the Ada language specification. For each base metric we have provided the corresponding breakdown factors expressed in terms of our proposed metric set.

C.3.1(15) Protected interrupt handler overhead. The execution time that cannot be directly attributed to the protected handler procedure.

This metric fully corresponds to our `Ext_it` term in Table 1, which denotes the runtime primitive that locates the protected handler attached to the asserted interrupt and the housekeeping actions that precede and follow execution occurring at interrupt level, which are known as "prologue" and "epilogue", respectively. Prologue and epilogue actions may involve installing and removing the interrupt stack, if any.

Protected interrupt handler lateness (PIHL). The execution time incurred from the time that the interrupt was asserted until the first user statement of the attached protected interrupt handler.

This metric corresponds to the sum of our `Ext_it` and `Po(Enter)` notional primitives, the latter term denoting the runtime overhead of seizing the lock to the protected object hosting the designated handler, detracted by the cost of the epilogue actions, which only occur upon leaving the object. (The latter set of actions may be particularly burdensome for register-window based architecture like the one we used for our measurements.)

This term is an indicator of runtime performance, but it is too coarse grained to be directly used for off-line scheduling analysis, which arguably requires finer-grained metrics like ours.

D.12(11) Mutual exclusive access. The time overhead incurred to seize mutually exclusive access to an entry-less protected object.

This metric encompasses the sum of our `Po(Enter)` and `Po(Exit)` notional primitives.

D.9(13) Delay until lateness. The difference between the requested time of delay expiration and the resumption time actually attained by a task following an absolute time suspension.

This metric encompasses the whole set of overhead factors that precede the execution of a time-triggered task, which Figure 4 denotes as $J^W + BI + CH + CS1$.

As the cost of the `Ready` and `Select` component of term *CS1* is often linear in the number of tasks becoming ready (cf. Equation 3), we took two measurements, for the case of 1 and $(1 + N)$ tasks becoming ready.

This term is a general indicator of runtime performance, but it is too coarse to be directly used for off-line scheduling analysis and also hard to empirically measure in the worst case. Once again, break-down factors like ours are better suited for this purpose.

Context switch: The elapsed time between the last statement executed at a task dispatching point until the first statement in the selected running task.

Our metric set captures this duration as the sum of the two overhead factors *CS1* (cf. Section 3.2.1.4) and *CS2* (cf. Section 3.2.1.7). The coarse metric is a general indicator

Table 4. Comparative values, expressed in processor clock cycles, for standard metrics obtained on kernel configurations using GNAT run-time libraries.

Coarse metric	Our metric	ORK	RTEMS
C.3.1(15)	Ext_it	1028	6829
PIHL	Ext_it + Po(Enter) −epilogue	893	5002
D.12(11)	Po(Enter) + Po(Exit)	534	2467
D.9(13), 1 task	$J^W + BI + CH + CS1_{(1)}$	2835	7840
D.9(13), $1 + N$ tasks	$CS1_{(+N)}$	$N \times 140$	$N \times 2240$
Context switch	$CS1 + CS2$	431	928

of runtime performance, but accurate off-line scheduling analysis definitely needs better breakdown factors like ours.

Table 4 lists the measurement values obtained for the metrics of interest. The values are given in processor clock cycles. The measurements were taken from executions on TSIM (Gaisler Research, 2003), a clock cycle true simulator of ERC32 targets, with both kernels configured to run at 14 MHz.

Most of the values could be obtained exactly by direct use of performance measurement features of the simulator. A few others required use of the kernel clock, which exposed them to the potential inaccuracy of the specific clock handling mechanisms. Unlike ORK, which uses the interval time model, RTEMS uses the traditional periodic clock model in a manner whereby the clock resolution equals the periodic interrupt period. We set the RTEMS timer interrupt period to 1 ms so as to attain sufficient accuracy while incurring an acceptable overhead. Slightly better performance figures could be obtained should a coarser clock resolution be used, but that would not compare fairly with the high clock accuracy sought by the Ravenscar Profile.

As Table 4 shows, the Ravenscar-specific kernel performs significantly better than its general-purpose counterpart, while our metrics are considerably finer-grained than standard metrics used for the full runtime (and, by extension, for general-purpose real-time kernels). This proves the point we made throughout this paper.

5. Conclusions

Real-time systems are inherently concurrent. Yet, when static analysis is important, designers refrain from making this character explicit because of the perceived conflict between ease of verification and direct expression of concurrency. In contrast with this perception, the Ravenscar Profile is a concurrent computational model, which facilitates direct expression of predictable concurrency in a form easily amenable to static analysis. Response time analysis is a handy and informative form of static analysis. It determines the longest elapsed time it takes for a process to complete its most demanding instance of activation under maximum contention from the rest of the system. The values that feed this analysis describe the time cost of the application code as well as that of the runtime services involved with that execution. The determination of these values is exposed to the detrimental effect of

excessive pessimism that may arise from inaccurate or coarse characterisation of the execution behaviour of the system. The simplicity and intrinsic determinism of the Ravenscar Profile help contain this effect, for they permit a very accurate characterisation of all the primitive services supported by the runtime. In this paper we have provided an accurate account of the dynamic semantics of those runtime primitives and formulated metrics for an equally accurate characterisation of their timing behaviour. To further prove the attractiveness of the Ravenscar Profile we have also shown how its implementation allows the compliant runtime to greatly reduce its overhead in comparison with more general-purpose real-time kernels.

Acknowledgments

We gratefully thank the rest of the ORK development team: José Ruiz, Rodrigo García, Ramón Fernández, and Miguel Múñoz. We also acknowledge the useful comments and recommendations by the anonymous reviewers.

Note

1. The latter clause is the only exception to FIFO treatment of tasks within priority queues, in the interest of sparing context switches.

References

Ada 95. 2000. Consolidated Ada Reference Manual, International Standard ISO/IEC-8652:1995(E) with Technical Corrigendum 1. Springer-Verlag, LNCS 2219.

Ada Core Technologies. 2000. GNAT Reference Manual. Version 3.13. Ada Core Technologies.

ATMEL. 2003. TSC695F SPARC single chip processor—User Manual. http://www.atmel.com (search = "ERC32").

Audsley, N., Burns, A., Davis R., Tindell, K. and Wellings, A. J. 1995. Fixed priority preemptive scheduling: an historical perspective. *Real-Time Systems* 8(3): 173–198.

Bailey, C., Fyfe, E., Vardanega, T., and Wellings, A. 1993. The use of preemptive priority-based scheduling for space applications. In *Proceedings of the Real-Time Systems Symposium*, Vol. 14. Raleigh-Durham, NC (USA), pp. 253–257, IEEE.

Baker, T., and Vardanega, T. 1997. Session summary: tasking profiles. In *Proceedings of the 8th International Ada Real-Time Workshop*. Ada LettersVol XVII(5): 5–7.

Barnes, J. 1998. *Programming in Ada 95*. 2nd edn. Addison-Wesley Publishing Co. ISBN: 0201342936.

Burns, A. 1991. Scheduling hard real-time systems: a review. *Software Engineering Journal* 6(3): 116–128.

Burns, A. 1994. Preemptive priority based scheduling: an appropriate engineering approach. In S. Son (ed.), *Advances in Real-Time Systems*. Prentice-Hall.

Burns, A. 1999. The Ravenscar Profile. *Ada Letters* XIX(4): 49–52.

Burns, A., Dobbing, B., and Romanski, G. 1998. The Ravenscar Profile for high integrity real-time programs. In L. Asplund (ed.), *Reliable Software Technologies—Ada-Europe'98*. Springer-Verlag.

Burns, A., Dobbing, B., and Vardanega, T. 2003. Guide to the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York (UK). http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf.

Burns, A., and Wellings, A. J. 2001. *Real-Time Systems and Programming Languages*, 3rd edn. Addison-Wesley.

Cornhill, D., and Sha, L. 1987. Priority inversion in Ada or what should be the priority of an Ada server task?. *Ada Letters* 7(7).

de la Puente, J., Ruiz, J., and Zamorano, J. 2000a. An open Ravenscar real-time kernel for GNAT. In H. B. Keller and E. Ploedereder (eds.), *Reliable Software Technologies—Ada-Europe 2000* Springer-Verlag, pp. 5–15.

de la Puente, J., Ruiz, J., Zamorano, J., García, R., and Fernández-Marina, R. 2000b. ORK: An open source real-time kernel for on-board software systems. In *DASIA 2000—Data Systems in Aerospace*. Montreal, Canada.

Dijkstra, E. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *CACM* 18(8): 453–457.

Dobbing, B., and Romanski, G. 1999. The Ravenscar Profile: experience report. In *Proceedings of the 9th International Real-Time Ada Workshop. Ada Letters* XIX(2): 28–32.

Gaisler Research. 2003. The TSIM home page. http://www.gailser.com → TSIM.

Goodenough, J., and Sha, L. 1988. The priority ceiling protocol: a method for minimizing the blocking of high priority Ada Tasks. Technical Report SEI-SSR-4, Software Engineering Institute, Pittsburgh, Pennsylvania.

INTEL. 1989. i486 User's Manual. Intel.

Joseph, M., and Pandya, P. 1986. Finding response times in real-time systems. *BCS Computer Journal* 29(5): 390–395.

Klein, M. H., Ralya, T., Pollack, B., Obenza, R., and González-Harbour, M. 1993. *A Practitioner's Handbook for Real-Time Analysis. Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer Academic Publishers.

Liu, C., and Layland, J. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20(1).

Liu, J. W. S. 2000. *Real-Time Systems*. Prentice-Hall.

Locke, C. D. 1992. Software architectures for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems* 4(1): 37–53.

Locke, C. D., Sha, L., Rajkumar, R., Lehoczky, J., and Burns, G. 1988. Priority inversion and its control: an experimental investigation. In *Proceedings of the International Workshop on Real-Time Ada Issues*, ACM SIGAda. *Ada Letters* VIII(7).

On-Line Applications Research. 2003. The on-line applications research RTEMS home page. http://www.oarcorp.com/RTEMS/rtems.html.

Puschner, P., and Burns, A. 2000. A review of worst-case execution time analysis. *Real-Time Systems* 18(2/3): 115–128.

RTCA: 1992, Software considerations in airborne systems and equipment ceritication. Requirements and Technical Concepts for Aviation. RTCA SC167/DO-178B. Issued in Europe as EUROCAE document ED-12B.

Sha, L., Rajkumar, R., and Lehoczky, J. P. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Tr. on Computers* 39(9): 1175–1185.

TEMIC. 1996. SPARC V7 Instruction Set Manual. TEMIC.

Vardanega, T. 1999. Development of on-board embedded real-time systems: an engineering approach. Technical Report ESA STR-260, European Space Agency. ISBN 90-9092-334-2.

Vardanega, T., and Caspersen G. 2001. Using the Ravenscar Profile for Space Applications: The OBOSS case. In M. González-Harbour (ed.), *Proceedings of the International Workshop on Real-Time Ada Issues*. *Ada Letters*, XXI:96–104.

Wellings, A. 2001. 10th International Real-Time Ada Workshop—Session Summary: Status and Future of the Ravenscar Profile. *Ada Letters* XXI(1).

Zamorano, J., and de la Puente, J. 2002. Precise Response Time Analysis for Ravenscar Kernels. In J. Tokar (ed.), *Proceedings of the International Workshop on Real-Time Ada Issues*, *Ada Letters* XXII.

Zamorano, J., Ruiz, J., and de la Puente, J. 2001. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In A. Strohmeier and D. Craeynest (eds.): *Reliable Software Technologies—Ada-Europe 2001*. Springer-Verlag, pp. 317–327.

Tullio Vardanega graduated in computer science at the University of Pisa, Italy. Subsequently, he received his PhD in computer science from the Technical University of Delft, The Netherlands, while being staff member of the European Space Research and Technology Centre (ESTEC) at Noordwijk in the Netherlands. At ESTEC, Dr Vardanega held responsibilities for research and technology transfer activities in the area of on-board embedded real-time software. In January 2002, he was appointed lecturer in Computer Science, Faculty of Science, at the University of Padua, Italy, where he took on teaching and research responsibilities in the areas of high-integrity real-time systems, quality of service under real-time constraints and software engineering methods and processes for such environments. He has authored numerous papers and technical reports on these subjects.

Juan Zamorano is an Associate Professor in the Department of Computer Architecture and Technology (DATSI) at the School of Computer Science (FI) of the Technical University of Madrid (UPM). He teaches Computer Architecture and Real-Time Systems. He is a member of the Real-Time Systems Group at DIT-UPM, and has collaborated in a number of national and international research projects. His research interests are in real-time systems design and implementation, including design methods, software and hardware architectures, and real-time operating systems and languages. Juan Zamorano has authored or co-authored more than 20 technical papers and reports, most of them in the fields of Ada and real-time systems.

Juan Antonio de la Puente is a Professor in the Department of Telematics (DIT) at the School of Telecommunication Engineering (ETSIT) of the Technical University of Madrid (UPM). He has held teaching and research positions at the Technical Universities of Madrid and Valencia (UPV). He leads the Real-Time Systems Group at DIT-UPM, and has been responsible for a number of national and international research projects. His research interests are in real-time systems design and implementation, including design methods,

software architectures, and real-time operating systems and languages. Professor de la Puente has authored orco-authored more than 60 technical papers and reports, and has edited several conference proceeding volumes, most of them in the fields of Ada and real-time systems.