On the Effect of Protected Entry Servicing Policies on the Response Time of Ada Tasks

Jorge Garrido(✉), Juan Zamorano, Alejandro Alonso, and Juan A. de la Puente

Sistemas de Tiempo Real e Ingeniería de Servicios Telemáticos (STRAST), Information Processing and Telecommunications Centre, Universidad Politécnica de Madrid (UPM), Madrid, Spain
{jgarrido,jzamorano,aalonso,jpuente}@dit.upm.es

Abstract. Real-time multiprocessor systems are being used extensively in industrial applications. Ada provides ample support for such systems, including a complete tasking model providing time predictability, especially when restricted by the Ravenscar profile. A fundamental element of this tasking model is inter-task communication by means of protected objects. The definition of resource locking policies with bounded priority inversion is a fundamental aspect of protected objects, which has received considerable attention, with some interesting results that can be used in multiprocessor real-time systems. However, there is another important subject, the service policy for protected entries, that has received less attention in the research community and is also important in order to guarantee a predictable time behaviour. The impact of the service model on the response time analysis of multiprocessor real-time systems is evaluated in the paper for the self-service model and the proxy model, and their relation to the MSRP and the MrsP locking policies is discussed. Extensions to response time analysis for the proxy model with both locking policies are also contributed.

Keywords: Real-time systems · Multiprocessor systems
Compiler implementation · Ada Ravenscar profile
Schedulability analysis

1 Introduction

Ada support for multiprocessors allows developers to build real-time embedded applications with enhanced performance and full control over the execution of applications on the available processor cores.

This work has been partially funded by the Spanish National R&D&I plan (project M2C2, TIN2014-56158-C4-3-P).

Real-time systems require temporal properties of the tasks to be ensured by the implementation. For hard real-time systems, task deadlines must be guaranteed in all cases, including the worst possible conditions. This usually requires using scheduling methods with a predictable temporal behaviour [10], as well as analysis methods that enable developers to verify that the temporal behaviour of the system meets the requirements (see e.g. [7] for a comprehensive presentation of the topic).

The Ada Real-Time Annex [1, annex D] provides a flexible priority-based dispatching model that allows developers to use some common scheduling policies. Dynamic priorities provide additional flexibility. Dispatching domains can be assigned to the tasks in order to specify the processor or processors on which they may execute. The flexibility of the full Ada tasking model, however, makes temporal analysis difficult or even impossible. The Ravenscar profile [1, D.13] was defined in order to provide a limited tasking model that ensures that static temporal analysis techniques can be applied to real-time systems. In the following, a multiprocessor system model based on fully-partitioned fixed-priority scheduling with inter-task communication based on shared protected objects (PO) will be assumed. This model is compatible with the Ravenscar Profile, although not necessarily limited to it.

Two important aspects of the implementation of protected objects are the mechanisms used for servicing blocked PO entry queues, and the locking policies that must be used for minimising the effects of priority inversion [18]. A number of multiprocessor locking protocols have been proposed [16], among which MSRP [11] and MrsP [8] have received widest attention.

While the use of resource locking protocols, both in general Ada and the Ravenscar profile, has been analysed in detail [12], the impact of entry queue servicing policies on the temporal behaviour of multiprocessor real-time systems has not been discussed in detail. The aim of this paper is to contribute to such analysis, with focus on the so-called proxy model. The contents are organised as follows: Sect. 2 describes the details of entry servicing in Ada protected objects. Section 3 summarises the main results on real-time analysis of the MSRP and MrsP protocols. The main contribution is the impact of service models on response time analysis, which is discussed in Sect. 4. Finally, the conclusions of the study and suggestions for future work are presented in Sect. 5.

2 Protected Objects in Ada

2.1 Protected Objects and Protected Operations

Protected objects are the preferred mechanism for inter-task communication in Ada. A protected object consists of one or more private data fields, together with a set of operations that can be carried out on the data. Protected operations can be of three kinds: functions, procedures, and entries, and are executed in mutual exclusion. Only procedures and entries can change the protected data, and therefore multiple calls to protected functions may be executed concurrently if the implementation chooses to do so.

2.2 Protected Entries

Protected entries have a Boolean barrier. When a task issues a call to an entry, the barrier is evaluated and the call is accepted if the barrier is true. Otherwise, the calling task is suspended on a queue associated to the entry. Barriers are reevaluated at the end of every execution of a protected procedure or entry. If there are any pending calls on entries with open barriers, one of the calls is selected to be serviced, i.e. removed from the queue and executed.

The service order for pending entries can be specified with a pragma Queuing_Policy. The default policy is FIFO. Alternatively, priority order or some other implementation-defined policy can be defined. The default policy does not specify which entry queue is to be served first if there are more than one queue with open barriers.

Pending queued entries take precedence over new calls trying to access the protected object. In this way, a task that was waiting for change in the state of the protected object can resume its execution with the guarantee that the state has not changed again, thus avoiding possible race conditions and starvation [4]. This rule is commonly known as the "eggshell model".

The evaluation of barriers and the execution of the entry body that is selected to be serviced at the end of a protected operation can be executed in different ways, as the standard does not specify which task should serve the entry. Two possible approaches are the *self-service model* and the *proxy model* [14,17].

2.3 Self-service Model

Under this approach, when a task ends a protected operation and an entry with an open barrier is selected for execution, the task that has called the entry (the *caller task*) is resumed, and executes the entry body. This is a simple approach that allows for parallel execution of both tasks, and may thus be preferable for multiprocessor implementations. However, execution on monoprocessors may be less efficient, as it requires more context switches, and may be difficult to implement on some real-time kernels [15].

2.4 Proxy Model

An alternative approach is the *proxy model* [14]. In this model, the task ending a protected operation acts as a *server task* that reevaluates the barriers and executes the selected entry body on behalf of the caller task. The process is repeated while there are pending entries with true barriers. This approach saves context switches and simplifies the design of the run-time system, which makes it the method of choice for implementing protected entry servicing on monoprocessors. It can also be used on multiprocessors, although the implementation may become more complex in this case, depending on whether the caller and the server task execute on the same or different processors [9].

2.5 Ravenscar Restrictions

The proxy model may make real-time analysis difficult with unrestricted Ada tasking, since the number of tasks that may be waiting on entry queues may be high, thus leading to a very long execution time for the server task. However, in the Ravenscar profile protected objects may have at most one entry, and there may be at most one waiting task on a closed entry. Therefore, the server task may have to execute at most one entry body, and its execution time stays bounded, thus making response time analysis feasible from a practical point of view.

3 Resource Sharing Protocols for Multiprocessor Systems

3.1 Resource Sharing Protocols

A fundamental issue in multiprocessor real-time systems is the definition of resource access protocols that provide for bounded task blocking. Although other approaches are possible [16], most of the published work has been aimed at adapting some well-known monoprocessor methods, such as the Priority Ceiling Protocol (PCP) [18] or the Stack Resource Policy (SRP) [2,3], to multiprocessor systems.

The default policy for Ada is Ceiling_Locking, which is based on SRP, a generalisation of PCP also valid for Earliest Deadline First (EDF) scheduling. In the following we examine in detail two multiprocessor protocols derived from SRP that have received significant attention from the research community, namely MSRP and MrsP.

3.2 Multiprocessor Stack Resource Policy

The Multiprocessor Stack Resource Policy (MSRP) [11] is an extension of SRP for multiprocessors. The MSRP system model is defined by a fully partitioned scheduling with global resources that are, in turn, not bounded to a specific processor. In this policy, unsatisfied resource accesses are serviced in FIFO order, and tasks spin-wait non-preemptively until access is granted. The following properties are inherited from SRP:

– A task can only be locally blocked before it starts executing.
– A task can only be locally blocked once per activation, bounded to one critical section length.
– It can be easily implemented on a multiprocessor Ravenscar-compliant kernel by assigning all global resources a ceiling priority equal to the highest priority in the system.
– The access cost to a shared resource. i.e. the longest time a task can be blocked awaiting is bounded.

The access cost to a shared resource is bounded as requests are serviced in FIFO order and at most one request per processor can be issued at a time, because shared resource accesses are not preemptable. Therefore, the maximum time a task τ_i running on processor P_m can be spinning waiting to access a resource r^k can be expressed as:

$$\text{spin}(r^k, P_m) = \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k \tag{1}$$

where w_x^k is the worst-case access time to resource r^k which a task τ_x executing on a remote processor may experience. The spin time calculated as above is to be added, for each access, to the task worst-case execution time when carrying out the schedulability analysis. This result can be improved, as shown by Brandenburg and Wieder [5,19], by using holistic analysis and mixed-integer linear programming techniques to safely reduce the pessimism in the number of times a remote processor can cause spin delay on each task activation.

Using MSRP with fixed-priority scheduling, as in the Ravenscar profile, has a major drawback. Since waiting for access to a shared resource is not preemptable, the blocking incurred by a high-priority task does not depend on its use of shared resources, but on that of lower-priority tasks, even though the ceiling priorities of the shared resources are lower than the priority of the high-priority task. Since high-priority tasks often have short deadlines, especially if priorities are assigned in deadline-monotonic order, they can be expected to be most affected by priority inversion.

3.3 Multiprocessor Resource Sharing Protocol

The Multiprocessor resource sharing Protocol (MrsP) [8] was devised to address the above described drawback in MSRP. This protocol also relies on the properties of PCP and SRP: resources are assigned a ceiling priority, and all access requests are performed at that priority. As in MSRP, access requests are dealt with in FIFO order, and the tasks waiting for access to a resource spin-wait until they are granted access. However, the spin-wait and access itself are done at the ceiling priority of the resource, and thus calling tasks can be preempted. Therefore, tasks with priorities higher than the ceiling priority of the resource are not blocked by lower-priority tasks accessing the resource.

Another benefit of waiting at ceiling priorities is that, as in MSRP, at most one task per processor can be trying to access the resource at a time. As a result, and like in MSRP, the length of the resource FIFO queue is bounded by the number of processors from where the resource is accessed. A desirable access cost to a resource would then be the sum of worst access times of each remote processor, plus the cost of the access to be performed [12], as in Eq. (1) above. However, since all the shared resource activity is executed at its ceiling priority, accesses are not guaranteed to be completed without being preempted by a higher-priority task.

In order to achieve the same access cost with MrsP as with MSRP, it must be made sure that a task executing an operation on a shared resource makes progress while other tasks are spin-waiting for the resource. To this end, spin-waiting tasks must be capable of undertaking the access operation of a locally preempted task holding the resource lock. This cooperation must respect the FIFO order [8]. This can be accomplished by delegating the execution on the waiting task, or by migrating the preempted task to a processor where a task is spin-waiting for the resource.

The first method is not practical. This approach would require accesses to shared objects to be atomic, without side effects and potentially being executed in parallel by waiting task, accepting only one final commit. The second approach can, on the contrary, be easily implemented since task migration mechanisms are integrated in most run-times of multiprocessor operating systems. In particular, an Ada implementation based on a smart modification of the affinities of the involved entities which may enable this kind of migration is outlined in [6]. The main drawback of this approach is the overhead caused by such migrations. A way to account for this overhead and an evaluation of its influence is presented in [20], where MrsP is shown to provide better schedulability than MSRP, even including this overhead.

4 Impact of Service Modes in Response Time Analysis

An aspect that has not received enough attention to date is the impact of the entry queue service models on the response time of multiprocessor real-time systems. The next paragraphs discuss the main issues related to using the proxy and the self-service models with MSRP and MrsP.

4.1 Entry Servicing in MSRP

MSRP is compatible with the Ada definition of protected objects, as long as the ceiling priority of all protected objects is assigned a non-preemptable value, i.e. one which is higher than any task priority.

The current GNAT implementation of protected objects on multiprocessors follows the proxy model. As previously explained, a task calling an entry with a closed barrier is suspended. When a call to another protected procedure or entry completes, barriers are reevaluated, and pending calls to entries with newly opened barriers are executed by the calling task, which acts as a server task. Since the server task executes non-preemptively, all pending entry bodies are executed until no remaining task is enqueued on an open entry.

The eggshell model implies that the resource is busy while pending entries are executed by the server task, and thus further calls to protected operations are postponed until the enqueued requests have been served. This makes the maximum number of calls to be executed by the server task to be bounded by the maximum number of tasks that can issue entry calls on the resource.[1] Let $G_e(r^k)$ be the set of tasks calling entries in resource r^k. If $|G_e(r^k)|$ is the

[1] For this bound to be effective it must be assumed that the task set is static, or at least that there is a bound on the number of caller tasks for the resource.

size of this set, the worst-case overhead incurred in each access to a protected subprogram operation in r^k is:

$$\text{overhead}(r^k, P_m) = |G_e(r^k)| \times C_e^k + \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k + |G_e(r^k)| \times C_e^k \quad (2)$$

where C_e^k is the maximum cost of servicing an entry request in resource r^k, P is the set of all processors, and P_m is the processor on which the server task runs, as in Eq. (1).

With the Ravenscar profile, the analysis is simplified because there may be at most one task blocked on an entry, and a protected object can have at most one entry. The worst-case overhead is then:

$$\text{overhead}(r^k, P_m) = C_e^k + \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k + C_e^k \quad (3)$$

4.2 Entry Servicing in MrsP

MrsP does not have a defined behaviour for the Ada entry model. The Ada implementation proposed in [6] gives no hint on how enqueued entries should be serviced and analysed under MrsP. In order to complete the protocol definition, some alternatives for such implementation are explored in the following paragraphs.

Self-service Model. This approach is widely accepted as best suited for multiprocessor systems, since it is supposed to better support parallelism. However, in practice it presents some drawbacks, particularly a loss of efficiency when implemented on top of POSIX threads, even on monoprocessors [17].

The expected benefit of using self-service in multiprocessor systems comes from the fact that the task waiting on a closed entry and the task opening the barrier can be allocated to different processors. However, this can lead to unexpected blocking in the execution of higher-priority tasks, contrary to the monoprocessor case, where a task can only be blocked once per activation by a lower-priority task, and only before the higher-priority task starts executing.

Consider the example depicted in Fig. 1 including tasks τ_1 and τ_2 with priorities $p_1 = 1$ and $p_2 = 2$, respectively.[2] Let τ_1 be blocked on an entry belonging to a protected object r with a ceiling priority $p_r = 3$, and then τ_2 is released and executes non-protected code on the same processor. If the barrier is opened after a protected operation executing on some other processor, τ_1 becomes runnable again with an active priority of 3, thus preempting τ_2.

To prevent such a case, one possibility would be to decrease the active priority of a task waiting on a closed entry barrier. A similar scheme is used in GNAT for the proxy model, where the caller task, after locking the protected object, reverts to its previous active priority, to be rescheduled at that priority once

[2] Following Ada convention, a higher value indicates a higher priority.

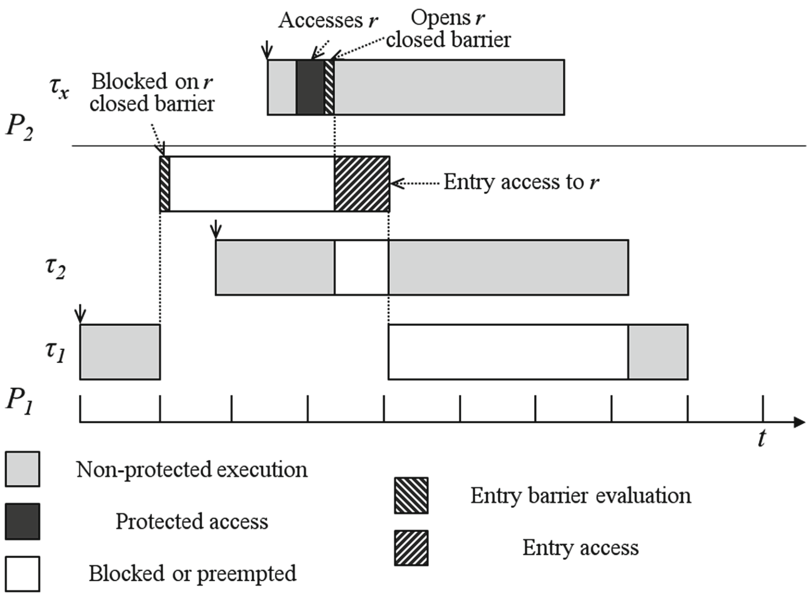

Fig. 1. Potential delayed priority inversion under self-service model.

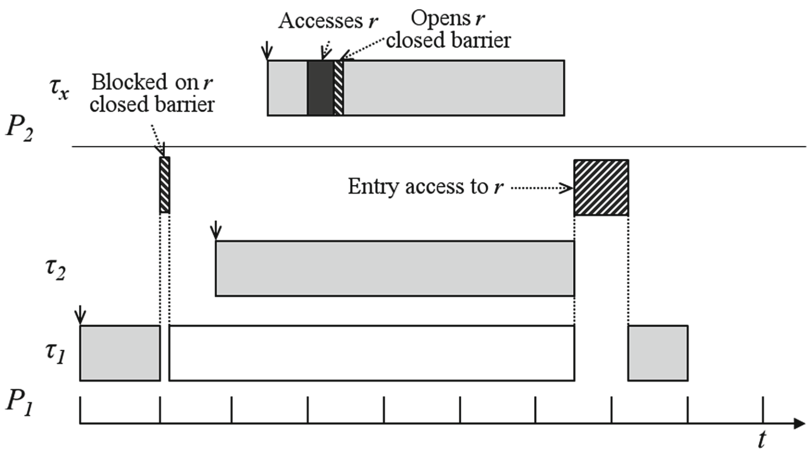

Fig. 2. Tasks blocked on closed entry barriers decrease their active priority.

its entry request has been served. This approach is illustrated in Fig. 2. In this example, τ_1 can perform its entry access to the resource (raising again its active priority to 3) when τ_2 execution is completed.

This approach, however, can lead to further issues. A task that is spin-waiting on an entry call may not be able to access the resource even after the barrier has been opened by another task, because the priority of the waiting task is not

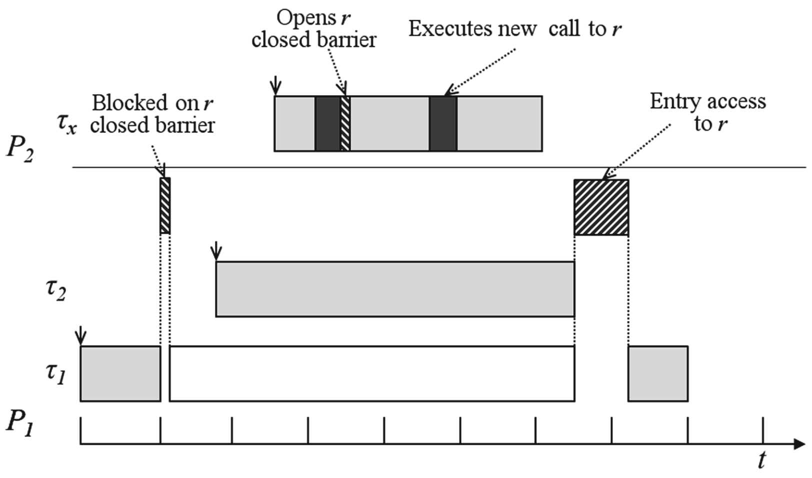

Fig. 3. Approach breaking the eggshell model.

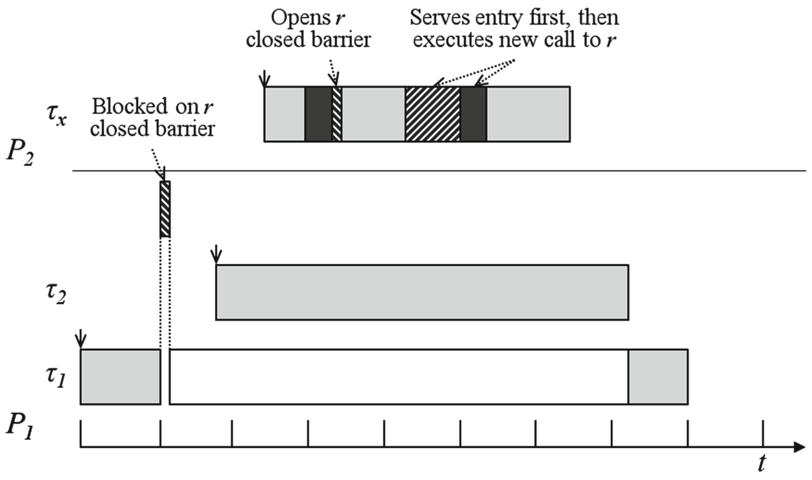

Fig. 4. Approach implementing a delayed proxy model.

high enough, as it has been reverted to its basic priority. If there is a new call to a protected operation in the same object, it should not be accepted until the pending entry call has been serviced, as per the eggshell model. The second call is thus delayed even if the entry barrier is now open, thus leading to further priority inversion. Note that this can also happen when the task that makes the new call runs on the same processor as the waiting task.

Aside from being highly inefficient, the timing behaviour in such situations cannot be analysed without adding extra pessimism. Two possible ways of tackling this issue are:

– Letting the new call proceed without serving the entries, thus breaking the eggshell model (Fig. 3).
– Serving the entries and then letting the new call proceed. This would be somewhat of a 'delayed proxy model' (Fig. 4).

The former option is not compatible with the Ada standard, and therefore will not be further discussed. The latter one, on the contrary, would not be far from the current proxy GNAT implementation. Furthermore, as the entry calls are only executed by delegation, as shown in Fig. 4, when strictly required, parallelism can be improved.

Unfortunately, this solution would only make a true benefit if both the task calling the entry and that making the new call are hosted on the same processor. In any other case this solution would complicate the implementation. Executing a subprogram call (even a function call) would require to check whether there are remote entry callers queued on open barriers, then to check their scheduling state (if they are currently running or not) and finally to undertake their access if necessary to preserve the FIFO order.

In any case, this solution is actually a variant of the proxy model, and its implications can be studied along with the discussion in the following paragraphs.

Proxy Model. As previously shown, serving outstanding entry calls at the end of the call that opens the corresponding barriers improves efficiency and is consistent with the Ada semantics.

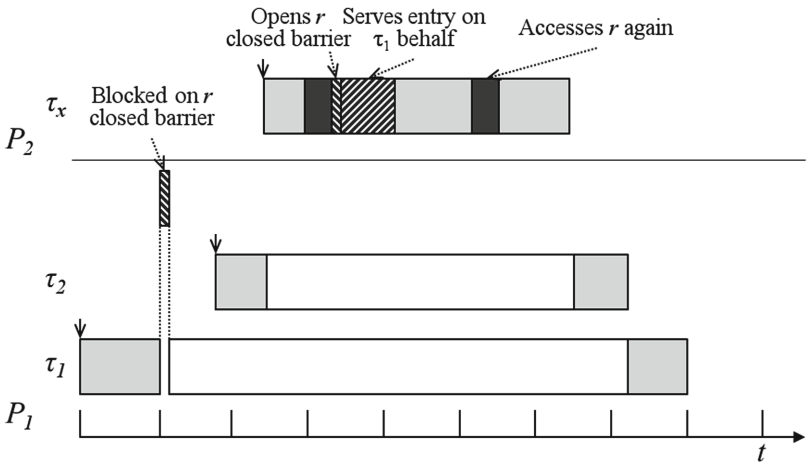

Fig. 5. Proxy model entry servicing.

When accesses to protected objects are carried out in a non-preemptive way, the entry servicing is deterministic: entries are served in FIFO order (as required by MSRP and MrsP) by the server task on its own processor, until the barrier is closed again or all queued entry calls have been served.

However, when accesses to protected objects can be preempted, a different kind of problem arises, as the servicing of pending entry calls can be delayed by a preemption of the server task, which compromises response time analysis. In order to overcome this issue and ensure progress in the server tasks, two possible solutions can be envisioned:

- To migrate the server task to a processor where the access to the resource can be completed, according to the helping mechanism presented in Sect. 3.2.
- To perform the entry servicing non-preemptively in order to avoid such situation.

In general, under MrsP a task holding a resource lock may be preempted. As explained in Sect. 3, the preempted task is migrated to a remote processor if there is a task actively waiting for the resource. This approach further increases the potential access cost to a shared resource, since the migration cost has to be included in the analysis. The worst-case number of migrations can be obtained by calculating the number of local higher-priority tasks releases during the access [20], i.e. for each valid migration target (each processor that hosts at least one task accessing the resource), calculate the number of times it can suffer interference from higher-priority tasks. This is calculated by obtaining the ceiling value of the resource access time divided by the period of each higher-priority task on the migration targets. This value, multiplied by the measured worst-case cost of a migration is to be added to the final resource access cost. By adding the time of servicing an entry to the resource access time, i.e. extending the time in which the access can be preempted and migrated again, the previously presented analysis yields a safe upper bound for the overhead induced by migrations under the proxy model.

The other possible solution is to serve the entry in a non-preemptable way. While this would limit the overhead in the server task, it might also negatively affect the response time of higher-priority tasks, even those not accessing the involved resource. MrsP was designed to avoid or at least reduce the unnecessary blocking suffered by higher-priority tasks from resources only used by lower-priority tasks. However, short non-preemptable sections can still be beneficial with this protocol, as shown in the evaluation made in [20]. Keeping non-preemptable sections short should not be specially costly in real systems, especially those implemented with the Ravenscar profile, since entries are mainly used for task synchronization purposes.

In order to consistently use MrsP with the Ada tasking model, the impact of servicing entries first, according to the eggshell rules, must be evaluated. The following equation, which follows a similar approach as (2), can be used to extend the results in [8] and [13, Eq. 5]:

$$e^k = |map(G(r^k))| \times (c^k + |G_e(r^k)| \times C_e^k) \tag{4}$$

where e^k is the cost of a single access to a resource r^k. Function $map(G(r^k))$ returns the set of processors that host at least one task accessing resource r^k

and $|map(G(r^k))|$ returns the size of that set. This safely bounds the number of access requests to r^k that can be issued at a time, since, due to the use of ceiling priorities, only one task per processor can be trying to access the resource at any given moment. This number is multiplied by the time required for an access to the resource, that is the sum of is execution time c^k, plus the time required to serve the potentially queued entry calls with barriers now open. For Ravenscar systems, this equation can be simplified as only one entry request can be queued per access:

$$e^k = |map(G(r^k))| \times (c^k + C_e^k) \tag{5}$$

As shown in [20], this way of analysing resource contention is highly pessimistic, especially when resource access request patterns are uneven among tasks and processors. Consider a task running on a processor issuing requests to a resource every few milliseconds, while there is only another task accessing the same resource from a different processor, with a rate in the order of seconds. Then it is clearly pessimistic to assume that all accesses from the first task will be delayed by accesses from the second task. Therefore, the response time analysis must provide means to reduce that pessimism based on the periodicity of the requests issued on each processor. This same reasoning can be used to reduce the pessimism on the need to service entry queue calls present in Eqs. 4 and 5. Given the semantics of entry calls their frequency may not be comparable to that of other protected actions. This is particularly true for Ravenscar systems, where entries are only meant to be used to synchronize tasks, and thus the frequency of entry requests may be expected to be clearly lower than that of other requests. If this is the case, it is clearly pessimistic to assume that each non-entry access will suffer an entry servicing overhead (Fig. 5).

A safe upper bound for the maximum entry-servicing overhead a task τ_i can incur on a single activation due to a resource r^k can be calculated as:

$$EN_i^k = \sum_{\tau_x \neq \tau_i} \left\lceil \frac{R_i}{T_x^k} \right\rceil \tag{6}$$

where T_x^k is the minimum inter-arrival time of entry requests to resource r^k by a task τ_x, and R_i is the response time of τ_i. This result can be used to reduce the pessimism of the previous equations, and therefore, the cost of all accesses to a resource r^k by τ_i during an activation can be expressed as:

$$E_i^k = EN_i^k \times C_e^k + N_i^k \times (|map(G(r^k))| \times c^k) \tag{7}$$

where N_i^k is the maximum number of times τ_i accesses r^k. Note that, while Eqs. 6 and 7 are also valid for general Ada tasking, with the Ravenscar profile, EN_i^k can be also safely bounded by $N_i^k \times |map(G(r^k))|$, since each access to the resource can only have to serve at most one entry. In consequence, the lower of both values is to be used for schedulability analysis.

5 Conclusions

Protected objects and entries are a powerful mechanism for controlling the synchronization of concurrent tasks. Nevertheless, Ada protected entries exhibit some peculiarities that have to be taken into account when analysing the temporal behaviour of real-time systems.

Among the possible implementation of protected entry servicing in multiprocessors, self-service has a potential for taking advantage of parallel execution to improve the efficiency of the mechanism. However, it cannot be used with locking policies based on PCP or SRP, such as MSRP or MrsP, without compromising the properties of these protocols or violating the eggshell definition.

On the other hand, the proxy model is simpler to implement, and can be used with MSRP and MrsP. It has also been shown to be analysable for systems using non-preemptive spin-locking. The overhead caused by the extra execution time in protected calls, as well as the impact of giving priority to entry servicing over new calls, as required by the eggshell model, on the response time analysis, have been calculated, and new response time equations have been derived.

References

1. Ada Reference Manual, ISO/IEC 8652:2012(E) with COR.1:2016 (2016). http://www.ada-auth.org/arm.html
2. Baker, T.P.: A stack-based resource allocation policy for realtime processes. In: 1990 Proceedings of the 11th Real-Time Systems Symposium, pp. 191–200, December 1990
3. Baker, T.P.: Stack-based scheduling for realtime processes. Real-Time Syst. **3**(1), 67–99 (1991)
4. Barnes, J.: Programming in Ada 2012. Cambridge University Press, Cambridge (2014)
5. Brandenburg, B.B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2011)
6. Burns, A., Wellings, A.J.: Locking policies for multiprocessor Ada. Ada Lett. **33**(2), 59–65 (2013)
7. Burns, A., Wellings, A.: Analysable Real-Time Systems: Programmed in Ada. CreateSpace Independent Publishing Platform (2016)
8. Burns, A., Wellings, A.J.: A schedulability compatible multiprocessor resource sharing protocol-MrsP. In: 2013 25th Euromicro Conference on Real-Time Systems (ECRTS), pp. 282–291. IEEE (2013)
9. Chouteau, F., Ruiz, J.F.: Design and implementation of a Ravenscar extension for multiprocessors. In: Romanovsky, A., Vardanega, T. (eds.) Ada-Europe 2011. LNCS, vol. 6652, pp. 31–45. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21338-0_3
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). http://doi.acm.org/10.1145/1978802.1978814
11. Gai, P., Lipari, G., Natale, M.D.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the 22nd IEEE Real-Time Systems Symposium. IEEE Computer Society (2001)

12. Garrido, J., Zamorano, J., Alonso, A., de la Puente, J.A.: Evaluating MSRP and MrsP with the multiprocessor Ravenscar profile. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 3–17. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_1
13. Garrido, J., Zhao, S., Burns, A., Wellings, A.: Supporting nested resources in MrsP. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 73–86. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_5
14. Giering, E.W., Baker, T.P.: The GNU Ada Runtime Library (GNARL): design and implementation. In: WADAS 1994: Proceedings of the Eleventh Annual Washington Ada Symposium & Summer ACM SIGAda Meeting on Ada, pp. 97–107. ACM Press, New York (1994)
15. Giering, E.W., Mueller, F., Baker, T.P.: Implementing ada 9x features using posix threads: design issues. In: Proceedings of the Conference on TRI-Ada 1993, TRI-Ada 1993, pp. 214–228. ACM, New York (1993). http://doi.acm.org/10.1145/170657.170736
16. Lin, S., Wellings, A.J., Burns, A.: Ada 2012: resource sharing and multiprocessors. Ada Lett. **33**(1), 32–44 (2013)
17. Miranda, J.: A detailed description of the GNU Ada run time (2003). http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/
18. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
19. Wieder, A., Brandenburg, B.B.: On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In: Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, 3–6 December 2013, pp. 45–56 (2013). https://doi.org/10.1109/RTSS.2013.13
20. Zhao, S., Garrido, J., Burns, A., Wellings, A.: New schedulability analysis for MrsP. In: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10. IEEE (2017)