Asynchronous Critical Sections in Real-Time Multiprocessor Systems

Michael Schmid(✉) and Jürgen Mottok

Laboratory for Safe and Secure Systems - LaS^3,
University of Applied Sciences Regensburg, Regensburg, Germany
{michael3.schmid,juergen.mottok}@oth-regensburg.de

Abstract. Sharing data across multiple tasks in multiprocessor systems has intensively been studied in the past decades. Various synchronization protocols, the most well-known being the Priority Inheritance Protocol or the Priority Ceiling Protocol, have been established and analyzed so that blocking times of tasks waiting to access a shared resource can be upper bounded. To the best of our knowledge, all of these protocols share one commonality: Tasks that want to enter a critical section, that is already being executed by another task, immediately get blocked. In this paper, we introduce the Asynchronous Priority Ceiling Protocol (A-PCP), which makes use of aperiodic servers to execute the critical sections asynchronously, while the calling task can continue its work on non-critical section code. For this protocol, we provide a worst-case response time analysis of the asynchronous computations, as well as necessary and sufficient conditions for a feasibility analysis of a set of periodic tasks using the proposed synchronization model on a system that preemptively schedules the tasks under the rate-monotonic priority assignment.

1 Introduction

Sharing data between various tasks plays a very important role in real-time systems. Therefore, over the last few decades, synchronization protocols have intensively been established and studied in order to provide bounded blocking times for tasks. The best known of such protocols are the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP), both covered by Sha et al. in [8]. However, all of the real-time synchronization protocols we found in the state of the art are based on mutual exclusion and thus, immediately block tasks that want to enter the critical section which is already being executed by another task. In this paper, we consider asynchronous critical sections, i.e. the execution of the critical section is relocated to an aperiodic server associated with the shared resource while the tasks waiting on the result of the asynchronous computations can continue their execution in order to carry out other work that does not access shared resources. The proposed model introduces a few benefits:

(1) The data locality is improved as the computations always take place on the same processor and thus, reduces the amount of data moved around in distributed systems and non-uniform memory access (NUMA) architectures.

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 56–67, 2019.
https://doi.org/10.1007/978-3-030-18656-2_5

(2) Tasks that make use of asynchronous critical sections continue their execution with their normal priority and thus, priority inversions can only occur on processors running aperiodic servers.
(3) Blocking times and the amount of context switches are reduced as the tasks only block and yield when the results of the asynchronous computations are not available at the time instant when they are needed.

The last item is only beneficial when the work that the task performs after raising a request for the asynchronous critical section is larger than the execution time of the critical section. As an example, we assume that the shared resource is not being accessed and a task raises a request for an asynchronous critical section. The execution of the critical section will immediately start while the task runs non-critical code in the meantime. Both threads are considered to execute without preemption. As soon as the task finishes running the non-critical code, it self-suspends to wait for the result of the asynchronous computations. This does not happen when the task tries to acquire the result of the asynchronous computations after the execution of the critical section has finished or in the case where the critical section is executed directly by the task as it is done in common synchronization protocols. However, as critical sections tend to be short compared to non-critical sections [1], tasks should rarely be blocked by asynchronous critical sections.

1.1 Contribution and Structure

Following contributions are added to the state of the art:

(1) A real-time multiprocessor synchronization protocol that allows the asynchronous execution of critical sections through the use of aperiodic servers.
(2) A model for the proposed protocol that upper bounds the worst-case response times of the asynchronous critical sections under rate-monotonic preemptive scheduling.
(3) Necessary and sufficient conditions for a feasibility analysis of a task set using the Asynchronous Priority Ceiling Protocol.

The rest of this paper is organized as follows: Sect. 1.2 presents related work on real-time synchronization. The notations used throughout this paper are presented in Sect. 2 together with the model of the A-PCP. Subsequently, in Sect. 3, the response time analysis of asynchronous critical sections on systems using rate-monotonic preemptive scheduling is conducted and followed by the feasibility analysis in Sect. 4. At last, the outcome of this paper is summarized in Sect. 5.

1.2 Related Work

Many different real-time synchronization protocols can be found in the state of the art. The two best-known are described by Sha et al. in [8], namely the Priority Inheritance Protocol and the Priority Ceiling Protocol. They derive a

set of sufficient conditions under which a set of periodic tasks can be scheduled by rate-monotonic preemptive scheduling on a single processor. Both protocols deal with uncontrolled priority inversion problems by temporarily raising the priority of the task holding the critical section. An important advantage of the PCP over PIP is that the former protocol prevents transitive blocking and deadlocks. In [5,6], Rajkumar et al. made necessary adaptions to the Priority Ceiling Protocol that allow a schedulability analysis for tasks executing in parallel on distributed (Distributed PCP, D-PCP) and shared (Multiprocessor PCP, M-PCP) memory multiprocessor systems, respectively. Both previously mentioned papers provide a pessimistic analysis of worst-case blocking times of tasks. A more detailed analysis of the Multiprocessor PCP is covered in various papers, e.g. by Lakshmanan et al. [2] and Yang et al. [10]. For a more detailed survey of real-time synchronization protocols the reader is referred to [4].

As mentioned before, all real-time resource sharing protocols known to us share the commonality that a task that wants to enter a critical section is blocked when the shared resource is already being accessed by another task. In the sector of distributed computing, the Active Object pattern [7] describes a way of providing synchronized access to a shared resource by relocating the computations to a thread of execution residing in the control of the shared resource. Thereby, the execution of the critical sections is done asynchronously, allowing the task to continue its computation on non-critical section code. To the best of our knowledge, no real-time analysis of this pattern has been conducted in order to prevent unbounded blocking times and priority inversions of tasks using this pattern. As a result, our work is the combination of the Active Object pattern and the D-PCP.

2 Notations and Model

We now present the notations used for the task model and the asynchronous critical sections, as well as the assumptions that are necessary for the response time and feasibility analysis of the A-PCP.

2.1 Asumptions and Notations

In this paper, we consider a task set Γ of n periodic tasks scheduled on a shared-memory multiprocessor with m identical processing cores $p_1, p_2, ..., p_m$. Note that we will use the words processor and cores interchangeably. Each task τ_i (with $1 \leq i \leq n$) is represented by a 2-tuple $\tau_i = (T_i, C_i)$, where T_i is the period of the task and C_i denotes the worst-case execution time (WCET). The task periodically releases a job, at multiples of T_i, which executes for C_i units of time. The l-th job of task τ_i is denoted as $J_{i,l}$ and is released at time instant $r_{i,l}$. An arbitrary job of τ_i is denoted as $J_{i,*}$ with its release time being $r_{i,*}$. Furthermore, we consider implicit deadlines, i.e. the deadline of $J_{i,l}$ is equal to the release time $r_{i,l+1}$ of the subsequent job. Each job may be preempted by higher priority jobs and resume its execution later on. The time instant at which job

$J_{i,l}$ finishes its execution is denoted as the completion time $f_{i,l}$. The worst-case response time (WCRT) of a task τ_i is defined as $R_i = \max_{\forall l}(f_{i,l} - r_{i,l})$. As in [6], we assume that tasks are statically allocated to processors and assigned a fixed priority based on the rate-monotonic algorithm. The priority P_i is shared by all jobs of task τ_i. We assume that lower indices represent a higher priority, i.e. task τ_i has a higher priority than τ_j if $i < j$. The sets of tasks with a higher and lower priority than τ_i are denoted as $hp_i(\Gamma)$ and $lp_i(\Gamma)$, respectively.

Throughout this paper, the accesses to shared resources $\varrho_1, \varrho_2, ..., \varrho_x$ are protected by aperiodic servers $\alpha_1, \alpha_2, ..., \alpha_x$ following the rules of A-PCP. Whenever a job of τ_i wants to access a shared resource, it raises a request to the corresponding aperiodic server of the shared resource. This server is responsible for executing the critical section. Each request $\mu_{i,l}$ is characterized by a 3-tuple $\mu_{i,l} = (\rho_{i,l}, \zeta_{i,l}, \gamma_{i,l})$, where l denotes the l-th request raised by an arbitrary job of task τ_i, $\rho_{i,l}$ indicates the time instant when $J_{i,*}$ raises the request, $\zeta_{i,l}$ represents the worst-case execution time of $\mu_{i,l}$ and $\gamma_{i,l}$ is the time instant when the result is needed by $J_{i,*}$ in order to continue its execution. It must be noted that the execution requirements $\zeta_{i,*}$ do not contribute to the execution time C_i of task τ_i. The completion time $\phi_{i,l}$ denotes the time instant when the aperiodic server has finished the computation of $\mu_{i,l}$ and has provided a result to the respective job. If $\phi_{i,l} > \gamma_{i,l}$, i.e. the task needs the result of an asynchronous critical section that has not yet finished its execution, the task is suspended until $\phi_{i,l}$. The worst-case response time $\sigma_{i,l}$ is defined as the maximum difference $\phi_{i,l} - \rho_{i,l}$ among all jobs of τ_i. Arbitrary requests and their properties are denoted as $\mu_{i,*}$, $\rho_{i,*}$, ..., $\sigma_{i,*}$. The set of requests raised by τ_i to an aperiodic server α_n is represented by M_i^n, in addition $M_i = \cup_{\forall n} M_i^n$ is the set of all requests raised by task τ_i. The priority of all requests in M_i is equal to the priority of τ_i. Finally, we do not allow nested asynchronous critical sections, i.e. aperiodic server α_x is not allowed to raise a request to α_y when $x \neq y$ and each task may only have one pending request, i.e. $\gamma_{i,l} < \rho_{i,l+1}$.

To clarify the notations, an example is shown in Fig. 1. Two tasks τ_x, τ_y and the aperiodic server run on three distinct processors p_1, p_2 and p_3. In the interval $[0, 3)$, the aperiodic server is executing a critical section with a computation time $\zeta_{...}$ from a request which was raised before $t = 0$. During the same interval the queue of the aperiodic server is considered empty. When task τ_y raises an arbitrary request $\mu_{y,*}$ at $t = \rho_{y,*} = 1$, the request is stored in the queue of the server for later execution. At $t = \rho_{x,*} = 3$, task τ_x also raises a request $\mu_{x,*}$ which is stored in the queue and the aperiodic server finishes the computations of $\zeta_{...}$. The server then decides which request to run next. In this case, we consider the priority of task τ_y greater than the priority of τ_x and thus, request $\mu_{y,*}$ is executed next for $\zeta_{y,*}$ units of time. At time $t = \gamma_{y,*} = 5$, task τ_y needs the results of the computations $\zeta_{y,*}$ in order to continue its execution. As the results are not available yet, the task suspends itself. At time instant $t = \phi_{y,*} = 8$, the aperiodic server finishes the execution of $\zeta_{y,*}$ and returns the result to τ_y which allows the task to continue its execution. Again, the aperiodic server picks the next request from the queue, which is request $\mu_{x,*}$, and executes it. At $t = \phi_{x,*} = 10$, the

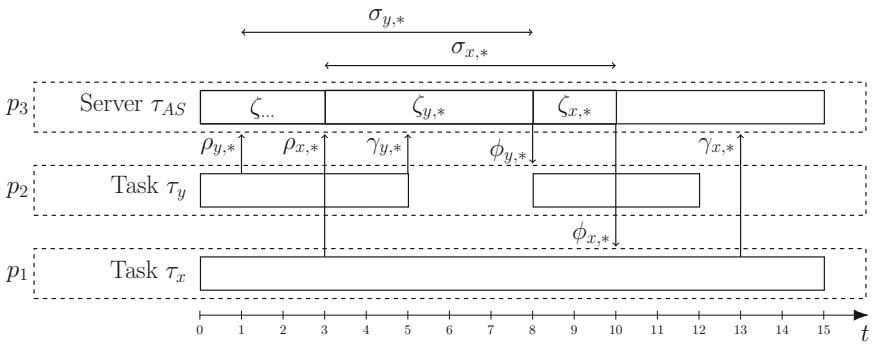

Fig. 1. Example of an asynchronous critical section

server finishes $\zeta_{x,*}$ and returns the result to τ_x. As $\gamma_{x,*} > \phi_{x,*}$, task τ_x does not need to self-suspend.

In concurrent programming languages, a way of returning the results of asynchronous computations is through the use of `future`-objects: On the function call which raises a request, tasks receive a `future`-object from the aperiodic server. When the task needs to retrieve the result, it calls the respective method of the `future` (in `C++` and `Java`, this method is called `get`), which blocks the task in case the server has not yet stored the result in the `future`. An example is shown in the `C++` code of Listing 1: In line 2, the task raises a request to the aperiodic server and receives a `future` in return. The server is responsible for executing the function `int modify_resource(int arg)` which modifies a shared resource. In the meantime, the task is able to perform some different non-critical section code. As soon as the task needs the result of the asynchronous computations, i.e. the return value of `modify_resource`, it calls the method `get` of the `future`, which either blocks when the result is not ready or returns the result otherwise.

```
1 // raise request to the aperiodic server task
2 future<int> future_obj = T_as.raise(modify_resource, 5);
3
4 // perform other work here, while the aperiodic server
      calls modify_resource with the argument '5'
5
6 int r = future_obj.get() //this line blocks if the
      result  was not stored in the future yet
```

Listing 1. Programming example of a task raising a request

2.2 Asynchronous Priority Ceiling Protocol

Due to its simplicity and its ability to suspend itself when the job queue is empty and restart once a new job is enqueued, we decided to use a deferrable server for the execution of the asynchronous critical sections. Typically, a deferrable

server is used to provide high responsiveness to aperiodic tasks. In our case, we will use the server to serialize the execution of the critical sections and thus, introduce synchronized access to a shared resource. We therefor briefly revisit the deferrable server model introduced by Strosnider et al. [9]: Each deferrable server is represented by a periodic task τ_{DS} with period T_{DS} and a capacity C_{DS}. The server is ready to execute at the beginning of its period and services aperiodic tasks until it exhausts its execution budget C_{DS}. The capacity is fully replenished at the end of each period.

Before deriving the worst-case response time and feasibility analysis, we define the properties of the A-PCP. In the A-PCP, each shared resource ρ_n is assigned a deferrable server α_n with period T^n_{DS} and execution budget C^n_{DS}. Tasks that want to access a shared resource, raise a request to the corresponding deferrable server. The request is stored in a priority ordered queue, while the server repeatedly pops the highest prioritized request from the queue and executes the respective critical section. As a shared resource is only accessed by one deferrable server and the critical sections are run to completion by the server, accesses to shared resources take place in a synchronized fashion.

Notation. We use Γ_n to denote the set of tasks that access the resource ρ_n at any given time.

Notation. We denote the set of tasks, both periodic and aperiodic server tasks, running on the same processor as the aperiodic server α_n as $sp(\alpha_n)$.

Definition. Let P_H be the priority of the highest priority task in the system. The priority ceiling Ω_n of a shared resource ρ_n is defined to be the sum of P_H and the highest priority of the tasks accessing ρ_n:

$$\Omega_n = P_H + \max_{\tau_i \in \Gamma_n} \{P_i\}$$

Every aperiodic server must run with the priority given by the priority ceiling of the corresponding shared resource. As a result of the rate-monotonic priority assignment, we need to determine a suitable period and a capacity which is large enough to handle the critical sections of all tasks accessing the shared resource. The period T^n_{DS} can be defined such that it is slightly smaller than the period of the next lesser prioritized task running on the same processor. Having the value of T^n_{DS}, the execution budget can be calculated by summing up the execution times ζ of all requests by all jobs running in the given period:

$$T^n_{DS} = \min_{\forall i} \{T_i | (\tau_i \in sp(\alpha_n)) \wedge (P_i < \Omega_n)\} - 1$$

$$C^n_{DS} = \sum_{\tau_i \in \Gamma_n} \left\lceil \frac{T_i}{T^n_{DS}} \right\rceil \sum_{\mu_{i,j} \in M^n_i} \zeta_{i,j}$$

Note that the aperiodic server tasks with smaller priority ceilings also have to be considered for the determination of T^n_{DS}.

3 Response Time Analysis

In this section, we derive upper bounds for worst-case response times of asynchronous critical sections. We start by showing that the execution of the asynchronous critical sections only depends on other asynchronous critical sections:

Lemma 1. *The response times of asynchronous critical sections is a function of other asynchronous critical sections only.*

Proof. As the deferrable server tasks are given the highest priorities on the processor, they are not subject to preemption by periodic tasks and also do not depend on the execution of tasks in other circumstances. Therefore, a deferrable server can only be preempted by a higher priority aperiodic server running on the same processor. As a result, the response times of asynchronous critical sections is only dependent on the execution of other asynchronous critical sections.

Following Lemma 1, we now consider job $J_{i,*}$ to raise an arbitrary request $\mu_{i,*}$ and derive the maximum amount of computations done by the deferrable server before it is able to execute $\mu_{i,*}$.

Lemma 2. *Each time a job $J_{i,*}$ raises a request $\mu_{i,*}$ for an asynchronous critical section to a deferrable server α_n, the execution of the critical section is delayed by lower priority critical sections running on α_n for at most*

$$d_i^l = \max\{\zeta_{j,k} | \mu_{j,k} \in \{M_j^n | \tau_j \in lp_i(\Gamma_n)\}\} - 1. \tag{1}$$

Proof. The proof follows from the fact that the execution of asynchronous critical sections on the deferrable server α_n can not be preempted by other requests to the same server. As lower priority requests will not be scheduled prior to $\mu_{i,*}$, only a request which is already being executed by the deferrable server delays the execution of $\mu_{i,*}$. The maximum delay d_i^l occurs when the longest request by lower priority tasks starts execution exactly one time instant before $\mu_{i,*}$ is raised.

Lemma 3. *Each time a job $J_{i,*}$ raises a request $\mu_{i,*}$ to a deferrable server α_n, the execution of the critical section is delayed by higher priority critical sections running on α_n for at most*

$$d_i^h(\Delta t) = \sum_{\tau_j \in hp_i(\Gamma_n)} \left\lceil \frac{\Delta t}{T_j} \right\rceil \sum_{\mu_{j,k} \in M_j^n} \zeta_{j,k} \tag{2}$$

during the interval Δt.

Proof. During the interval Δt, a higher priority task τ_j can release at most $\lceil \frac{\Delta t}{T_j} \rceil$ jobs. Every time a job $J_{j,*}$ runs, it can request $\sum_{\mu_{j,k} \in M_j^n} \zeta_{j,k}$ time units of computation from the deferrable server α_n. Summing up the amount of work imposed on the deferrable server by all higher priority jobs results in $d_i^h(\Delta t)$.

As aperiodic servers can be subject to preemption by higher prioritized servers, the execution time of the critical sections run by those servers has to be taken into account as well. The delay is accounted for in the equation shown in Lemma 4. It must be noted that servers always run with the priority ceiling of the shared resource. Due to this constraint, also a lower prioritized task τ_L can increase the response time of a request raised by a high priority task τ_H, if the aperiodic server of the request $\mu_{L,*}$ has a higher priority than the server of request $\mu_{H,*}$.

Notation. We denote the set of aperiodic server tasks that run on the same processor and have a higher priority than α_n as $hp_n(\alpha)$.

Lemma 4. *Each time a job $J_{i,*}$ raises a request $\mu_{i,*}$ to a deferrable server α_n, the execution of the critical section is delayed by higher prioritized aperiodic servers for at most*

$$d_i^{\alpha}(\Delta t) = \sum_{\alpha_m \in hp_n(\alpha)} \sum_{\substack{\tau_j \in \Gamma_m, \\ \tau_j \neq \tau_i}} \left\lceil \frac{\Delta t}{T_j} \right\rceil \sum_{\mu_{j,k} \in M_j^m} \zeta_{j,k} \tag{3}$$

during the interval Δt.

Proof. All higher prioritized servers can preempt α_n. During the time interval Δt, jobs of a task other than τ_i raising requests to higher prioritized servers can execute for at most $\lceil \frac{\Delta t}{T_j} \rceil$ times. Every time such a job $J_{j,*}$ executes, it will impose $\sum_{\mu_{j,k} \in M_j^m} \zeta_{j,k}$ time units of work to the higher prioritized server α_m. Summing up the work of all tasks imposed to higher priority aperiodic servers running on the same processor as α_n results in Eq. 3.

Notation. We use $d_i(\Delta t)$ to denote the sum of the previously derived delays of Lemmas 2, 3 and 4 and denote $e_{i,*}(\Delta t)$ as the sum of $d_i^{\alpha}(\Delta t)$ and the execution time $\zeta_{i,*}$:

$$d_i(\Delta t) = d_i^l + d_i^h(\Delta t) + d_i^{\alpha}(\Delta t) \tag{4}$$

$$e_{i,*}(\Delta t) = \zeta_{i,*} + d_i^{\alpha}(\Delta t) \tag{5}$$

Equation 4 characterizes the delay imposed on the execution of $\mu_{i,*}$ by requests that run before $\mu_{i,*}$ is scheduled on α_n, as well as higher prioritized servers. Once $\mu_{i,*}$ starts executing, only requests to higher prioritized servers can delay the response time $\sigma_{i,*}$. This is represented by Eq. 5 which accounts for the execution time of $\mu_{i,*}$ and the amount of time higher priority servers execute prior to the aperiodic server α_n. The maximum delay can be determined by finding the solutions d_i^{max} and $e_{i,*}^{max}$ of the recursive functions $d_i^{z+1}(d_i^z)$ and $e_{i,*}^{z+1}(e_{i,*}^z)$, respectively. The iteration starts with $d_i^0 = d_i^l$ and ends when $d_i^{z+1} = d_i^z$. Equivalently, $e_{i,*}^{z+1}(e_{i,*}^z)$ starts and ends with $e_{i,*}^0 = \zeta_{i,*}$ and $e_{i,*}^{z+1} = e_{i,*}^z$, respectively. Note that the above computations of d_i^{max} and $e_{i,*}^{max}$ yield a pessimistic estimation and can be reduced by considering the exact

amount of asynchronous critical sections requested to the deferrable servers in the two intervals.

Combining our previous results we can derive an upper bound for the worst-case response time $\sigma_{i,*}$ of a request $\mu_{i,*}$.

Theorem 1. *Each time a job $J_{i,*}$ raises a request $\mu_{i,*}$ the worst-case response time $\sigma_{i,*}$ of $\mu_{i,*}$ can be upper bounded by:*

$$\sigma_{i,*} = d_i^{max} + e_{i,*}^{max} \tag{6}$$

Proof. This theorem follows directly from the previous lemmas: Lemma 1 states that the computations of the deferrable server are only a function of other asynchronous critical sections, while Lemmas 2, 3 and 4 consider the amount of computations imposed by critical sections of other tasks. Finally, according to the model, $J_{i,*}$ may only have one pending request at a time and thus, only the computations $\zeta_{i,*}$ are imposed on the deferrable server by $J_{i,*}$. Before $\mu_{i,*}$ is scheduled by the server the requests considered in Lemmas 2, 3 and 4 contribute to $\sigma_{i,*}$. The maximum amount of time those requests execute prior to $\mu_{i,*}$ is accounted for in d_i^{max}. As soon as $\mu_{i,*}$ is running on the aperiodic server α_n, only higher priority servers can delay the response time $\sigma_{i,*}$ by preempting α_n. This is considered by $e_{i,*}^{max}$. Therefore, the sum of d_i^{max} and $e_{i,*}^{max}$ results in the worst-case response time.

4 Feasibility Analysis

Following the results of the previous section, we now provide sufficient conditions for a schedulability analysis. In our model, tasks can be considered to self-suspend themselves if the result of an asynchronous critical section is not available in time. We utilize this behavior to conduct the schedulability analysis conformable to [3,6], where the total amount of time a task remains suspended is added to the computation of the schedulability test. We rephrase Theorem 10 of [6] to match the wording of [3]:

Theorem 2. *A set of n periodic self-suspending tasks can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied:*

$$\forall i, 1 \leq i \leq n, \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1), \tag{7}$$

where B_i is the worst-case suspension time of task τ_i and n is the number of tasks bound to the processor under test.

Notation. We denote M_i^b as the set of requests to aperiodic servers that lead to a self-suspension of job $J_{i,*}$:

$$M_i^b = \{\mu_{i,j} | \phi_{i,j} > \gamma_{i,j}\}$$

Every processor has to be tested separately with the conditions of Eq. 7. If the processor under test does not run a deferrable server, the following portions contribute to B_i (adapted from [3]):

(1) The blocking time $b_i(np)$ of non-preemptive regions (e.g. local critical sections) of lower priority tasks on the processor that runs τ_i: Each time a job $J_{i,*}$ suspends itself it can be blocked for $b_i(np) = \max_{i+1 \leq k \leq n} \theta_k$ units of time, where θ_k denotes the worst-case execution time of non-preemptive regions on the processor. In this paper, we do not consider local critical sections, however, it is possible to run the common Priority Ceiling Protocol on local shared resources. If $|M_i^b|$ requests lead to a self-suspension of job $J_{i,*}$, then $(|M_i^b| + 1) * b_i(np)$ is added to B_i.

(2) The duration $b_i(ss^1)$ due to self-suspension of τ_i: The upper bound of the duration that a job of τ_i remains self-suspended due to $\mu_{i,j}$ can be determined by subtracting the instant when $J_{i,*}$ self-suspends from the worst-case completion time ($\phi_{i,j} = \rho_{i,j} + \sigma_{i,j}$) of the asynchronous critical section. Summing up the durations a job remains self-suspended due to all of its requests yields in $b_i(ss^1)$:

$$b_i(ss^1) = \sum_{\mu_{i,j} \in M_i^b} (\phi_{i,j} - \gamma_{i,j})$$

(3) The duration that accounts for deferred execution of higher priority self-suspending tasks on the same processor as τ_i:

$$b_i(ss^2) = \sum_{\tau_k \in hp_i(\Gamma)} \min(C_k, b_k(ss^1))$$

Notation. We use $sp_i(M_j)$ to denote the set of requests a task τ_j raises to aperiodic servers running on the same processor as task τ_i.

Since tasks allocated to a processor that runs at least one deferrable server can be blocked by every asynchronous critical section (even by their own requests and requests of lower priority tasks), B_i has to be computed differently:

$$B_i = \sum_{\tau_j \in \Gamma} \sum_{\mu_{j,k} \in sp_i(M_j)} \zeta_{j,k}$$

Theorem 2 provides sufficient conditions for a feasibility analysis. In [3], Liu derives a set of necessary and sufficient conditions based on the time-demand analysis. We can use the previously calculated values of B_i in order to determine the worst-case response time R_i of the tasks and identify the feasibility of the system:

Theorem 3. *A set of n periodic self-suspending tasks can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied:*

$$\forall i, 1 \leq i \leq n, R_i^{l+1} = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^l}{T_k} \right\rceil C_k \leq T_i, \tag{8}$$

where B_i is the worst-case suspension time of task τ_i and n is the number of tasks bound to the processor under test.

The worst-case response time of task τ_i can be determined by finding the solution to the recursive function in Eq. 8. The iteration starts with $R_i^0 = C_i + B_i$ and ends either when $R_i^{z+1} = R_i^z \leq T_i$, indicating that the task τ_i is schedulable or when $R_i^{z+1} > T_i$, which means that the task set is not feasible.

5 Conclusion and Future Work

In this paper, we introduced the asynchronous execution of critical sections through our proposed synchronization protocol named Asynchronous Priority Ceiling Protocol, which is a combination of the Distributed Priority Ceiling Protocol [6] and the Active Object pattern [7]. In the Asynchronous Priority Ceiling Protocol, each shared resource is assigned to a distinct aperiodic server that is responsible for executing the critical sections in a sequential manner. We therefor established a model and subsequently derived a worst-case response time analysis of the asynchronous computations for task sets using the proposed protocol and scheduled under rate-monotonic preemptive scheduling. The worst-case response times of the asynchronous critical sections allowed us to derive the worst-case suspension times of tasks and by making adaptions to the schedulability analysis of Rajkumar et al. [6] and Liu [3], we provided necessary and sufficient conditions that allow to determine the feasibility of a task set using the proposed synchronization protocol.

Our computation of the worst-case response times of the asynchronous critical sections yields a pessimistic bound and can be improved by considering the exact amount of requests a task raises to aperiodic servers. As a result, schedulability tests would benefit greatly from more accurate computations. Another important item of future work are evaluations and comparisons to common mutual exclusion based synchronization protocols. This can be done in terms of schedulability tests, simulations or on an actual system.

References

1. Brandenburg, B.B., Anderson, J.H.: A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 105–124. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92221-6_9
2. Lakshmanan, K., de Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: 2009 30th IEEE Real-Time Systems Symposium, pp. 469–478, December 2009
3. Liu, J.W.S.: Real-Time Systems. Prentice Hall, Upper Saddle River (2000)
4. Midonnet, S., Fauberteau, F.: Synchronizations: Shared Resource Access Protocols, pp. 149–191. Wiley, Hoboken (2014)
5. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings, 10th International Conference on Distributed Computing Systems, pp. 116–123, May 1990

6. Rajkumar, R., Sha, L., Lehoczky, J.P.: Real-time synchronization protocols for multiprocessors. In: Proceedings Real-Time Systems Symposium, pp. 259–269, December 1988
7. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, vol. 2. Wiley, Hoboken (2000)
8. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
9. Strosnider, J.K., Lehoczky, J.P., Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Trans. Comput. **44**(1), 73–91 (1995)
10. Yang, M.L., Lei, H., Liao, Y., Rabee, F.: Improved blocking time analysis and evaluation for the multiprocessor priority ceiling protocol. J. Comput. Sci. Technol. **29**(6), 1003–1013 (2014)