Fair and Starvation-Free Spinlock for Real-Time AUTOSAR Systems

Drona Nagarajan
drona.nagarajan@tugraz.at
Graz University of Technology
Graz, Austria

Tobias Scheipel
tobias.scheipel@tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

Abstract

We present in this paper, a fair and starvation-free spinlock protocol for partitioned fixed priority (P-FP) preemptive multi-core real-time operating systems (mRTOS). We discuss some strongly related works in this regard and highlight the drawbacks with respect to the AUTOSAR standards. We then proceed to define a system model and develop a mathematical framework to calculate an upper bound on the blocking time a task incurs under our protocol. We also discuss how our protocol was implemented in a state-of-the-art AUTOSAR-compliant mRTOS. Finally, through experimental evaluation, we show that our protocol performs better than the existing spinlock protocol of the mRTOS.

CCS Concepts: • **Computer systems organization → Embedded systems**; *multi-core resource sharing*.

Keywords: spinlocks, multi-core, AUTOSAR, FIFO ordering

ACM Reference Format:
Drona Nagarajan, Tobias Scheipel, and Marcel Baunach. 2024. Fair and Starvation-Free Spinlock for Real-Time AUTOSAR Systems. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605098.3635922

1 Introduction

Spinlock is an inter-core mutual exclusion mechanism wherein a task (or a thread) simply waits in a tight loop polling a lock variable until it is available. AUTOSAR [1] mandates the use of spinlocks for global resource sharing. When a task (τ_i) requests a globally shared resource that is currently occupied by another task on another core (τ_j), then τ_i busy waits on its core. AUTOSAR specifications specify that the spinning tasks are preemptible i.e., any higher-priority task released on the same core is allowed to preempt the spinning task. Such a spinlock is called a *preemptible spinlock*.

Of course, multiple tasks across cores can request a specific spinlock. AUTOSAR specifications do not mandate any specific order in which these requests are satisfied. The simplest, and often effective, way to implement preemptible spinlocks is to satisfy the requests in no particular order. Such spinlocks are called *unordered* spinlocks.

As observed by Wieder et al. in [2], the lack of ordering guarantees makes it challenging to derive non-trivial bounds on the worst-case blocking a task suffers due to other remote tasks competing for the same resource. To overcome the lack of ordering guarantees, several methods are proposed that associate a queue (First-In-First-Out (FIFO) or priority) with a globally shared resource. These queues ensure that all the requests are satisfied in a particular order.

Wieder categorizes various approaches to spinlocks in his thesis [3]. F|* category of spinlocks use a FIFO-based mechanism to manage access to globally shared resources. The "*" suggests that these spinlocks can be preemptible (F|P) or non-preemptible (F|N). Similarly, P|* category of spinlocks use a priority queue-based mechanism. Unordered preemptible spinlocks fall under the U|P category. Wieder has shown that F|* and P|* spinlocks always dominate the U|* spinlocks. Wieder defines the dominance relationship as "A spinlock of type A dominates a spinlock of type B if and only if, for any task set and for any task therein, the worst-case blocking duration under A does not exceed the worst-case blocking duration under B".

Although associating a queue with a spinlock resource guarantees an order, it also introduces problems of starvation and deadlocks as we will see in section 2. We propose an F|P type spinlock which guarantees fairness in the sense that all the requests are satisfied in a strictly FIFO order. We derive upper bounds on the blocking a task incurs under our protocol and empirically show that our approach dominates the existing implementation of a U|P spinlock in a state-of-the-art AUTOSAR-compliant OS.

2 Related work

M-PCP proposed by Rajkumar et al. associates a priority queue with every global resource [5]. In M-PCP, whenever a task requests a global resource (mutex) *M*, the task is suspended on its core if the request is not satisfied immediately. The request is then queued in a priority queue associated with the global resource. When a request reaches the head of

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635922

the priority queue, the corresponding task is resumed on its core. The priority of the task is raised to the ceiling priority of the resource which is defined as $\pi(M) = \pi_G + \pi_c$, where π_G is a global priority higher than the priority of all the tasks in the system and π_c, is the highest base priority out of all the tasks accessing M.

Due to the definition of the ceiling priority of M, a task that acquires M becomes the highest priority task on its core. Therefore, the task which acquires M can preempt any other task which has not acquired a global resource. Also, if two tasks, say τ_i and τ_j (on the same core) acquire global resources M_i and M_j respectively, such that $\pi(M_i) > \pi(M_j)$, then τ_i preempts τ_j regardless of the base priorities of the tasks when τ_i acquires M_i. The protocol also necessitates that the priorities of the tasks are comparable across cores.

M-PCP guarantees priority ordering of requests from different tasks to a global resource. However, the idea of suspending a task and then allowing the task to resume execution with a raised priority after acquiring a resource leads to multiple preemptions and, by extension, multiple priority inversions. The authors have mathematically treated multiple priority inversions in [6]. Moreover, using a priority queue may lead to the starvation of lower-priority tasks in the queue. Consider Figure 1:

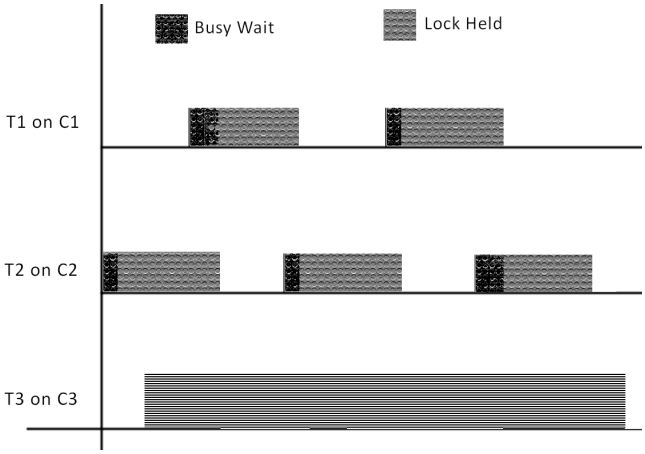


Figure 1. Starvation of a lower-priority task on a remote core.

We assume that tasks T_1, T_2, and T_3 are on different cores. The priorities of tasks T_1 and T_2 are greater than T_3. Clearly, due to the higher priority of tasks T_1 and T_2, the request from T_3 is pushed back in the priority queue and T_3 starves. Therefore, a strong progress guarantee cannot be implied.

Also, the definition of the ceiling priority of a global resource under M-PCP, requires that the base priorities of the tasks are comparable across cores. This condition is not always applicable to different specifications. In fact, a high-priority task on a core does not necessarily dominate a low-priority task on a remote core.

The Flexible Spinlock Model (FSM) proposed by Afshar et al. specifies that every globally shared resource has an associated FIFO queue [7]. Whenever a task requests a global resource, the request is queued in the associated FIFO queue if the request cannot be satisfied immediately. Whether or not the corresponding task suspends on its core is defined by a spinning priority, ρ_i^{spin}. The spinning priority can have any value in the range $0 \leq \rho_i^{spin} \leq \rho_{P_K}^{max}$. $\rho_{P_K}^{max}$ is the maximum priority of a core: $\rho_{P_K}^{max} = max\{Priority(\tau_i) \in P_K\} + 1$. Whenever a task requests a global resource that is currently occupied, the task is added to the FIFO queue and spins at spinning priority of ρ_i^{spin} on its local core. When the task acquires the resource, the priority of the task is raised to $\rho_{P_K}^{max} + \rho_i$.

The authors analyze different values of ρ_i^{spin}. If $\rho_i^{spin} = 0$, the task spins at the lowest spin level which is akin to a suspension-based approach. Whereas if $\rho_i^{spin} = \rho_{P_K}^{max}$, the task spins at the highest spin level which is akin to non-preemptible spinlock. A mathematical analysis concludes that "there is no single approach dominating the others.". This means that depending on the requirements of different critical sections and other optimizations, any spin level can be selected.

As stated in the case against M-PCP, suspending a task and then resuming execution with a raised priority leads to multiple preemptions and priority inversions. Whereas, a task spinning non-preemptively is a potential cause for starvation of other tasks on the same core. An optimal spin level is left as an open question by the authors.

Although using a FIFO queue implies a strong progress guarantee, it can also lead to a deadlock. Consider a scenario where a low-priority task τ_{lp} requests a global resource R_G. R_G is currently not available and therefore the request from τ_{lp} is queued in the FIFO queue associated with R_G and τ_{lp} starts spinning at some spinning priority $\rho_{\tau_{lp}}^{spin}$. A higher-priority task τ_{hp}, from the same core as τ_{lp}, arrives such that $\rho(\tau_{hp}) > \rho_{\tau_{lp}}^{spin}$, preempts τ_{lp} and requests the same resource R_G. This immediately causes a deadlock because the request by τ_{hp} is queued after the request by τ_{lp} (FIFO ordering) and τ_{lp} cannot execute because τ_{lp} cannot preempt τ_{hp}.

To prevent this scenario, τ_{lp} spins at the *highest local ceiling of global resources* (CP). CP is set to the highest priority of any task on that core that uses a global resource. Therefore, if τ_{hp} causes a preemption then τ_{hp} is a higher priority task not requesting global resources. Although this approach prevents a deadlock, if τ_{hp} is ***the*** highest priority task on its core accessing a global resource then, τ_{lp} spins at the highest priority and can potentially lead to starvation.

The Preemptable Waiting Locking Protocol (PWLP) proposed by Alfranseder et al. [15] also associates a FIFO queue with a global resource. PWLP draws parallels with the *short* variant of FSM developed by Block et al [16]. Unlike [16], PWLP allows preemptible spinning of tasks whose request is currently queued in the associated FIFO queue of the global

resource. Therefore, any higher-priority task on the same core can preempt the spinning task.

Under PWLP if a task is in the state *polling* (spinning) and a higher-priority task arrives, the higher-priority task preempts the current task. The *polling* task is suspended and enters a *parked* state. When a task transitions to the parked state, the corresponding resource request is deleted from the FIFO queue of the resource. When the task eventually transitions to the *running* state, it issues a new request for the required global resource. This approach avoids the deadlock scenario as described above.

Although this approach avoids deadlock due to requests for the same global resource from the same core, it may also lead to the starvation of tasks waiting for the resource. Since the task has to issue a new request, the position of the request in the FIFO queue is not deterministic. When the request is reissued, the request can be positioned at the end of the FIFO queue or at the head of the FIFO queue depending on the state of the queue at that time. Also, frequent queuing, deleting, and re-queuing requests may lead to significant OS overheads.

Apart from these strongly related works, the survey works by Brandenburg and Wieder [2] discuss more approaches using spinlocks and give a detailed blocking analysis of those approaches. Gai et al. introduced the multi-core Stack Resource Policy (MSRP) [18] for P-EDF systems which is an extension of the Stack Resource Policy developed by Baker in [17] for single-core systems with dynamic scheduling. In [14], Kontothanassis et al. propose a spin-based approach with a dequeuing policy. The policy is similar to a suspension-based approach in the sense that the task is de-queued from the resource queue if the task is preempted. The task is queued again after preemption. This leads to a higher delay for the tasks as the frequent de-queuing and re-queuing add extra overhead.

The related works discussed specify rules that may lead to starvation of tasks or do not conform to the strict priority-ordered execution of tasks. The starvation is not localized to a core but also affects completely unrelated tasks on remote cores. Apart from that, policies such as de-queuing and re-queuing add extra OS overheads.

We propose a F|P type spinlock that ensures that deadlocks are avoided. The policies also ensure that the requests are satisfied in a strict FIFO order. We also develop a mathematical framework that can be used to calculate upper bounds on the response times of the tasks sharing resources locally and/or across cores.

We have already discussed the strongly related work in Section 2. We define a system model and describe the rules and regulations of our protocol in Section 3. In Sections 3.2 and 3.3, we develop a mathematical model of the blocking a task incurs in our protocol. We discuss how our protocol was implemented in a state-of-the-art mRTOS and do an empirical evaluation of the performance of our protocol against the existing state-of-the-art implementation in the mRTOS in Section 4. Finally, Section 5 motivates some ideas for future work.

3 Multi-core Highest Locker Protocol

Before we proceed with specifying the rules of our protocol, we define a system model. We assume a P-FP preemptive system. We define a set of cores on a multicore processor as $C = \{C_1, C_2, C_3, \ldots, C_n\}$. A task set $T = \{\tau_1^x, \tau_2^x, ..., \tau_1^y, \tau_2^y, ..., \tau_1^z, \tau_2^z, ...\}$ such that each task $\tau_i^k \in C_k$, where C_k is the k^{th} core of a multi-core processor. Each task is characterized by its period ($p_{\tau_i^k}$), the execution time ($e_{\tau_i^k}$), the release time ($r_{\tau_i^k}$), and a priority ($P_{\tau_i^k}$). The tasks are arranged in descending order of their priorities ($P_{\tau_i^k} > P_{\tau_{i+1}^k}$). It is worth noting that priorities across cores are not comparable in our approach. Therefore, a task that has a high priority on its core does not dominate a lower-priority task on a different core.

AUTOSAR differentiates between locally shared and globally shared resources. Locally shared resources are managed using the Highest Locker Protocol (HLP) [12]. Globally shared resources are managed using spinlocks. We specify the following resource model for our protocol.

A set of spinlock-protected global resources is defined as $R_G = \{R_G^1, ..., R_G^p\}$. Each global resource R_G^x has an associated FIFO queue. Usually, for real-time systems, the number of tasks that will request access to a particular resource (local or global) is defined. Based on this assumption, we can define the length of the FIFO queue associated with every R_G^x as $\lambda_{R_G^x}$.

We allow for the nesting of global critical sections (*gcs*). We leverage the AUTOSAR specifications which state that the nesting order must be statically defined and must ensure freedom from deadlocks that may occur due to the wrong nesting order. This is the responsibility of the system designer and we assume that such a nesting order exists. However, in this work, we limit ourselves to non-nested spinlocks.

For multi-core resource sharing, we must ensure that gcs are atomic, i.e., if a task is executing a gcs, preemptions from local higher-priority tasks must not be allowed. As seen in the related work, this is achieved by raising the priority of the task which acquires a global resource. To guarantee that a gcs is atomic, we raise the priority of the corresponding task to the ceiling priority of the core. The ceiling priority of a core, $\overline{P}_{C_k}$, is equal to the priority of the highest-priority task on that core.

In addition to global resources, we also define a set of local resources as $R_L = \{R_L^1, ..., R_L^q\}$. Under HLP, the priority of the task, which has acquired a resource, is raised to the ceiling priority of the resource ($\overline{P}_{R_L^x}$). Preemptions can then be caused by only those tasks whose base priority is greater

than the ceiling priority of the resource that is currently occupied.

3.1 Specifications of M-HLP

We now specify the rules of our protocol:

1. Whenever a task requests R_G^x, the request is queued in the associated FIFO queue if the request cannot be satisfied immediately (FIFO queue is not empty). The corresponding task then starts spinning on its core at whatever priority it had when the request was issued. This allows other tasks on the same core which have a higher priority to preempt the spinning task and therefore maintain the strict priority-ordered scheduling mandated by AUTOSAR. The request of a preempted task is not deleted from the FIFO queue.
2. If the FIFO queue associated with R_G^x is empty (no requests from any task), then a new request is immediately satisfied. The priority of the task is immediately raised to $\overline{P}_{C_k}$ of the core to which the task belongs.
3. When a request which was queued, reaches the head of the FIFO queue, the corresponding task is *eligible* to enter its critical section. Since the corresponding task is spinning preemptively, it is possible that the task is not actively polling its position in the FIFO queue at the time when its request reached the head of the FIFO queue. When the task eventually resumes execution, it immediately acquires R_G^x, raises its priority, and enters its critical section.
4. When the task completes its gcs and releases R_G^x, the request is deleted from the FIFO queue, and the priority of the task is lowered to whatever priority it had before acquiring R_G^x. The next request in the queue is eligible to enter its critical section.

As explained in the related works section, FIFO-ordering of requests can cause a deadlock due to requests for the same global resource from tasks belonging to the same core. To tackle this problem and to maintain a strict FIFO ordering of the requests to R_G^x, we allow the higher-priority task, whose request for R_G^x is already queued, to yield the core to the lower-priority task on the same core, which is eligible to enter its critical section protected by R_G^x. The priority of the lower-priority task is raised immediately to the ceiling priority of its core and it enters its critical section.

This approach takes advantage of the fact that the higher-priority task is not doing any worthwhile execution. Therefore, allowing the higher-priority task to yield the core to a lower-priority task that is already eligible to enter its critical section allows the lower-priority task to progress.

3.2 Determining Acquisition Latency and Remote Blocking

To explain the acquisition latency, consider a set of three tasks on say core C_1. The tasks are characterized as follows $\{T = (p_{\tau_i^1}, e_{\tau_i^1}) : \tau_1^1 = (3ms, 1.4ms), \tau_2^1 = (5ms, 0.17ms), \tau_3^1 = (7ms, 2.09ms)\}$. We assume that the priorities of the tasks are defined based on the rate-monotonic priority assignment [11]. We also assume that all the tasks are ready/released at $t = 0ms$.

Now, τ_3^1 requests a global resource R_G^x at time $t = 2.4ms$. We assume that R_G^x was not available at this time therefore the request from τ_3^1 is queued in the associated FIFO queue of R_G^x. Since τ_3^1 spins preemptively, τ_1^1 and τ_2^1 can preempt τ_3^1.

Eventually, the request from τ_3^1 progresses to the head of the FIFO queue. Let's assume the request from τ_3^1 reaches the head of the FIFO queue at time $t = 3.8ms$. Figures 2a and 2b should clarify the state of the FIFO queue at different times.

Requests from other remote tasks	Requests from other remote tasks	Requests from other remote tasks	Request from T_3 on C_1 ($t = 2.4ms$)	A control task T_C

(a) $t = 2.4ms$

Request from T_3 on C_1 ($t = 3.8ms$)	A control task T_C			

(b) $t = 3.8ms$

Figure 2. State of the FIFO queue of R_G^x at different times

The Gantt chart (figure 3) shows the state of execution of the higher priority tasks in the core C_1. The red boxes correspond to task τ_1^1 and blue boxes correspond to τ_2^1:

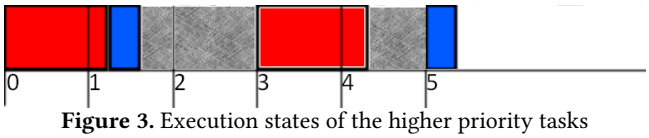


Figure 3. Execution states of the higher priority tasks

Clearly, at $t = 3.8ms$, τ_3^1 is preempted by τ_1^1. Although τ_3^1 (grey box) is *eligible* to enter its critical section, it acquires the resource after τ_1^1 completes its execution and τ_3^1 resumes execution. When τ_3^1 resumes execution, it immediately acquires R_G^x, the priority of τ_3^1 is raised to the ceiling priority of the core, and τ_3^1 begins executing its critical section. The difference between the time when a task becomes *eligible* to enter its critical section and when the task actually acquires the lock and enters its critical section is called *acquisition latency*.

In Figure 2a and 2b, this acquisition latency is propagated down the queue. From the perspective of the control task T_c, our question is: (1) How long do the higher priority tasks block a lower priority task that is eligible to enter its critical section (acquisition latency)? (2) How much blocking does the control task T_c incur due to all the requests that precede its request in the FIFO queue?

The time at which a spinning task becomes eligible is usually measured in absolute terms, i.e., from t = 0. Therefore, we need to calculate, in absolute terms, whether or not any local higher-priority tasks are already executing when a spinning task becomes eligible. If there are, then we also need to calculate, in absolute terms, how much time these

higher-priority tasks demand for completion. Assuming that the higher-priority tasks may also have to spin, waiting for a spinlock resource, the execution time of these higher-priority tasks may also be bloated.

We developed an algorithm that takes into account the *elapsed* time of the higher-priority tasks. We first use the algorithm in [10] that returns the *last idle instance* $(L_n(t), n = y-1$, where y represents the task $\tau_y^x)$ with respect to some time t. Assuming that a task τ_y^x becomes eligible at time t_{EL}, if $L_n(t_{EL}) = t_{EL}$, then τ_y^x suffers 0 acquisition latency. Otherwise, we calculate the acquisition latency. We first calculate the release times of the higher-priority tasks with respect to the last idle instance ($t = L_n(t_{EL})$) as:

$$r_j(t) = \left\lceil \frac{t - r_{\tau_j^x}}{p_{\tau_j^x}} \right\rceil \cdot p_{\tau_j^x} + r_{\tau_j^x}, \forall\ j = 1, \ldots, n \tag{1}$$

We calculate the *next computing instance* as [10]:

$$\rho_n(t) = \min_{j=1,\ldots,n} \left(\left\lceil \frac{t - r_{\tau_j^x}}{p_{\tau_j^x}} \right\rceil \cdot p_{\tau_j^x} + r_{\tau_j^x} \right) \tag{2}$$

Algorithm 1 calculates the total time elapsed after completion of the higher-priority tasks. The value $w(t)$ is the total relative time demanded for completion by the higher-priority tasks released at the last idle instance. Therefore, the sum $L_n(t_{EL}) + w(t)$ is the total time elapsed since $t = 0$.

The if-condition checks if the total time elapsed is less than the next computing instance relative to t_{EL}. If true, then the acquisition latency is the difference between the total time elapsed and t_{EL}. Else, we calculate the next computing instance relative to $L_n(t_{EL}) + w(t)$ and reiterate.

Applying the algorithm to our example where $t_{EL} = 3.8ms$ for task τ_3^1, we get $L_2(3.8) = 3ms$ (which indeed is the last idle instance as we can see in Figure 3). Also, the release times of tasks τ_1^1 and τ_2^1 can be calculated as:

$$r_1(3) = \left\lceil \frac{3-0}{3} \right\rceil \cdot 3 + 0 = 3$$

$$r_2(3) = \left\lceil \frac{3-0}{5} \right\rceil \cdot 5 + 0 = 5$$

The next computing instance with respect to t_{EL} is:

$$\rho_2(3.8) = \min_{j=1,2} \left(\left\lceil \frac{3.8-0}{3} \right\rceil \cdot 3 + 0, \left\lceil \frac{3.8-0}{5} \right\rceil \cdot 5 + 0 \right) = 5$$

Calculating $w(5)$, we get $w(5) = 1.4ms$. Finally, the if-condition evaluates to be true ($3 + 1.4 = 4.4ms < 5ms$) and we get an acquisition latency of 0.6ms.

Let the length of the gcs of task τ_3^1 associated with R_G^x be $\sigma(\tau_3^1, R_G^x)$. Going back to Figure 2b, the task τ_3^1 contributes a total blocking of $0.60ms + \sigma(\tau_3^1, R_G^x)ms$ towards the control task T_c. Task T_c is necessarily a task on a remote core $C_k \neq C_1$.

Algorithm 1 Calculate Acquisition Latency

```
procedure AcqLat(τ_y^x)(L_n(t_EL), t_EL)
    ITER ← 0
    t ← ρ_j(t_EL)
    while True do
        w(t) = Σ_{j=1}^{y-1} ⌈ (t − r_j(L_n(t_EL))) / p_{τ_j^x} ⌉ · e'_{τ_j^x}
        if L_n(t_EL) + w_j(t) < t then
            return ((L_n(t_EL) + w(t)) − t_EL)
        else
            ITER ← L_n(t_EL) + w(t)
            t ← ρ_j(ITER)
        end if
    end while
end procedure
```

Generalizing this result, a request for R_G^x from a task τ_i^k towards the tail of the associated FIFO queue incurs a worst-case blocking time of $B_{\tau_i^k}(R_G^x)$:

$$B_{\tau_i^k}(R_G^x) = max \left(\sum_{\forall \tau_y^x \rightarrow R_G^x \setminus \{\tau_i^k\}} \left(AcqLat(\tau_y^x) + \sigma(\tau_y^x, R_G^x) \right) \right) \tag{3}$$

where $\rightarrow$ signifies that the task τ_y^x accesses the global resource R_G^x. Since spinlock is a busy wait mechanism, the blocking caused by remote tasks is accounted for in the execution time of the task τ_i^k, therefore the execution time of τ_i^k is *bloated*. But, since we allow the task τ_i^k to spin preemptively on its core, the execution time of the task is not bloated for the entire duration of $B_{\tau_i^k}(R_G^x)$. Assuming that τ_i^k issues a request for R_G^x at time $req(R_G^x, \tau_i^k)$, we must subtract the preemptions caused by those higher-priority tasks that will be released in the interval $\Omega = [req(R_G^x, \tau_i^k), (req(R_G^x, \tau_i^k) + B_{\tau_i^k}(R_G^x))]$. The set of higher-priority tasks (let's say S) that are released, on core C_k, in this interval can be calculated using Equation (1) with $t = req(R_G^x, \tau_i^k)$:

$$S = \{\tau_j^k : r_j(req(R_G^x, \tau_i^k)) \in \Omega\} \tag{4}$$

The preemptions caused by the higher priority tasks can then be given by:

$$I_{\tau_i^k}(R_G^x) = \sum_{\forall \tau_j^k \in S} \left\lceil \frac{r_{\tau_j^k} - req(R_G^x, \tau_i^k)}{p_{\tau_j^k}} \right\rceil \cdot e'_{\tau_j^k}$$

Therefore, the execution time of task τ_i^k is bloated, with respect to resource R_G^x, by:

$$e'_{\tau_i^k}(R_G^x) = e_{\tau_i^k} + \left(B_{\tau_i^k}(R_G^x) - I_{\tau_i^k}(R_G^x) \right)_0 \tag{5}$$

where $(\dots)_0$ specifies that the minimum value of the expression inside the parenthesis is 0.

Finally, task τ_i^k may have more than one (non-nested) critical section. Therefore, the total worst-case execution time bloating incurred by τ_i^k is:

$$e'_{\tau_i^k} = e_{\tau_i^k} + \sum_{\forall R_G^l : \tau_i^k \to R_G^l} \left(B_{\tau_i^k}(R_G^l) - I_{\tau_i^k}(R_G^l) \right)_0 \tag{6}$$

We must consider that the set S may change for every spinlocked resource accessed by the task τ_i^k.

Also, the algorithm 1 considers the bloated execution time of all higher-priority tasks on core C_x. Therefore, the algorithm must be applied to every task on core C_x to obtain the bloated execution time of each $y-1$ tasks on core C_x.

Since F|P spinlocks dominate U|P spinlocks, the execution time bloating under M-HLP will always dominate the execution time bloating under U|P spinlock. Moreover, the analysis in Equation (6) is not pessimistic because we subtract the preemptions caused by higher-priority tasks on core C_k to account for the time when τ_i^k is preempted on its core while waiting in the FIFO queue associated with a spinlocked resource.

3.3 Accounting for Local Blocking

In the previous section, we upper-bounded the blocking contributed by remote tasks. Since spinlock is a busy-wait mechanism, the blocking time due to remote tasks is reflected in the execution time of the *blocked* task. In this section, we derive upper bounds on the blocking a task suffers due to the behavior of other tasks on the same core, hence the term local blocking.

Several factors affect the local blocking a task suffers due to the tasks on the same core. For instance, consider a low-priority task τ_{lp}^k which has acquired a local resource R_L^x. Now, according to the rules of HLP, the priority of τ_{lp}^k is raised to $\overline{P}_{R_L^x}$. If a higher-priority task τ_{hp}^k is released while τ_{lp}^k is still holding R_L^x, τ_{hp}^k will be blocked due to a priority inversion if $\overline{P}_{R_L^x} > P_{\tau_{hp}^k}$. In the worst case, τ_{hp}^k will be blocked for the entire length of the critical section of τ_{lp}^k protected by R_L^x ($\sigma(\tau_{lp}^k, R_L^x)$). A similar scenario can be constructed if τ_{lp}^k acquires a global resource R_G^x.

We present the following lemmas to derive an upper bound on local blocking a task incurs.

Lemma 3.1. *A task τ_i^k incurs priority inversion from at most one lower-priority task which has acquired a local resource such that $\overline{P}_{R_L^x} > P_{\tau_i^k}$.*

Proof. A lower priority task τ_{lp}^k can acquire a local resource R_L^x only when it is executing. Therefore, if the task τ_i^k is released, such that $\overline{P}_{R_L^x} > P_{\tau_i^k}$, τ_i^k is blocked from executing until τ_{lp}^k releases R_L^x and lowers its priority. The task τ_i^k immediately preempts τ_{lp}^k, and any other lower priority task that may have been released in the meantime, when τ_{lp}^k releases R_L^x. Hence, τ_i^k incurs priority inversion from at most one lower-priority task. □

Lemma 3.2. *A task τ_i^k incurs priority inversion from at most one lower-priority task which has acquired a global resource R_G^q which is not accessed by τ_i^k.*

Proof. The proof follows a similar argument as Lemma 3.1. A lower priority task τ_{lp}^k can only acquire R_G^q if it is currently executing. As per the rules of our protocol, the priority of the lower-priority task is raised to the ceiling priority of the core. Therefore, any higher-priority task which is released while τ_{lp}^k is executing its critical section cannot start execution until τ_{lp}^k releases R_G^q. As soon as τ_{lp}^k releases R_G^q, τ_i^k preempts τ_{lp}^k and other lower-priority tasks that may have been released in the meantime. Therefore τ_i^k is blocked by exactly one lower-priority task which acquires a global resource R_G^q which is not accessed by τ_i^k. This completes the proof. □

Local blocking caused according to Lemma 3.1 and Lemma 3.2 can be upper bounded as follows:

$$B_{res} = \max \left(\max_{\substack{\tau_{lp}^k \to R_L^x \\ P_{\tau_{lp}^k} < P_{\tau_i^k} \\ \wedge\, \overline{P}_{R_L^x} > P_{\tau_i^k}}} \sigma(\tau_{lp}^k, R_L^x), \max_{\substack{\tau_{lp}^k \to R_G^q \\ P_{\tau_{lp}^k} < P_{\tau_i^k}}} \sigma(\tau_{lp}^k, R_G^q) \right) \tag{7}$$

According to Lemma 3.1 and Lema 3.2, it is clear that a higher-priority task incurs a priority inversion from exactly one lower-priority task. Also, at any given time, the priority inversion can be due to a local or a global resource but not both. The worst-case blocking is equal to the maximum of the longest local and global critical sections.

Lemma 3.3. *The task τ_i^k incurs blocking from all those lower priority tasks that access the same global resources as τ_i^k. The blocking τ_i^k suffers is equal to the cumulative length of the critical sections of the lower priority tasks, protected by the same global resources as τ_i^k.*

Proof. This lemma stems from the fact that a higher priority task which is spinning for R_G^x yields the core to a lower-priority task from the same core whose request for R_G^x has progressed to the head of the queue. If there are l such lower-priority tasks, then, in the worst case, τ_i^k has to yield for all of them. Each such lower-priority task blocks τ_i^k exactly for the length of their critical section protected by R_G^x. □

Blocking caused by Lemma 3.3 can be upper bounded as follows:

$$B_{FIFO} = \sum_{\substack{\forall\ \tau_{lp}^k \to R_G^x \\ P_{\tau_{lp}^k} < P_{\tau_i^k}}} \sigma(\tau_{lp}^k, R_G^x) \mid \forall R_G^x : \tau_i^k \to R_G^x \tag{8}$$

The total local blocking is therefore given by:

$$\beta_L = B_{res} + B_{FIFO} \tag{9}$$

The worst-case response time of task τ_i^k is:

$$e'_{\tau_i^k} + \beta_L + \sum_{j=1}^{i-1} \left\lceil \frac{R_{\tau_i^k} + r_{\tau_i^k}}{p_{\tau_j^k}} \right\rceil \cdot e'_{\tau_j^k} = R_{\tau_i^k} \tag{10}$$

where $R_{\tau_i^k}$ represents the worst-case response time of the task τ_i^k.

The Equation (10) is recursive which can be solved using fixed point iteration [2, 4, 24]. Note that we are using the bloated execution time of the tasks.

4 Implementation and Evaluation

We implement M-HLP in Elektrobit Tresos®AutoCoreOS. AutoCoreOS is a widely used AUTOSAR-compliant mRTOS that implements a U|P spinlock for multi-core resource sharing. We add our implementation of M-HLP as a new API to the AutoCoreOS.

To implement M-HLP, we use a ticketing system to ensure the FIFO ordering of requests from tasks across cores. Ticket locks are not new and there are several works (for instance: [8, 14]) that implement ticket locks. As stated in [9] "Ticket locks are fair in the strong sense; it eliminates the possibility of starvation.". Tickets are implemented using simple counters: "current ticket" and "now serving" added to the existing spinlock data structure.

Whenever a task requests a spinlock, the API assigns the "current ticket" value to the task and increments the respective counter in an atomic operation. The task then spins, preemptively, on its core, polling the "now serving" counter. When the "now serving" counter is equal to a task's assigned ticket value, the task is eligible to enter its spinlock-protected critical section. The priority of the task, as explained in the rules of M-HLP, is immediately raised to the ceiling priority of the core when the task acquires the spinlock and starts executing its critical section.

When a task completes its critical section and releases the spinlock, the "now serving" counter is incremented in an atomic operation. The priority of the task is lowered to the priority the task had before acquiring the spinlock. It should be noted that the *atomicity* of the operations (incrementing the counters, raising/lowering task priorities, etc.) are guaranteed by hardware-specific mechanisms beyond the scope of this paper.

For a commercial mRTOS such as AutoCoreOS, it is not sufficient to simply implement a ticketing system. We have to consider problems that usually occur in real-world applications. For example, the AutoCoreOS can be configured to "kill" a misbehaving task/application. We have considered such nuances while implementing the ticketing system. For instance, if a spinning task is killed (maybe due to execution budget overrun), the task relinquishes its ticket and the ticket itself is blacklisted. Therefore, the "now serving" ticket counter skips the ticket when the counter reaches the blacklisted ticket. This allows the FIFO-order to be maintained and allows progress of the queue.

AutoCoreOS comes with a configuration tool, Tresos, which is used to configure the tasks, spinlocks, schedule tables, core mapping of the tasks, etc. AutoCoreOS also provides tracing and instrumentation APIs which we use to gather data such as the response time and the execution time of the tasks, etc. We use the TC387XQ (100MHz, 4 Tricore cores) [20] microcontroller-based development board, designed by Infineon Technologies AG, to implement and test M-HLP.

4.1 Task set parameter generation

We fixed the number of cores to $k = 3$ and generated task sets with 5, 10, and 20 tasks per core (15, 30, and 60 tasks, respectively). For each core, and for a given number of tasks per core, we generate the task sets in the following way [22]:

1. We use the Dirichlet-Rescale (DRS) algorithm to generate task utilization [19, 21]. The DRS algorithm generates an unbiased distribution of utilization values that sum up to the required utilization U, which is an input for the algorithm.
   For every core C_k, we set $U_k = 80\%$. We then generate for every task τ_i^k, on core C_k, an unbiased distribution of utilization such that $\sum_{\forall \tau_i^k} U_{\tau_i^k} = U_k$.
2. We generate task periods using a log-uniform distribution [23]. AUTOSAR specifies using schedule tables to activate (one or many) task(s) or set (0 or many) events at specific temporal points called expiry points. For periodic task sets, the temporal length of the schedule table is the *lcm* of the periods of the tasks. The configuration tool Tresos makes configuring a schedule table for every core possible. To ensure maximum interference, we configured all the schedule tables to start simultaneously at $t = 0ms$. The tasks mapped to a schedule table for a specific core are configured to be released simultaneously at $t = 0ms$.
3. Assuming rate-monotonic scheduling, the worst-case response time of a task is considered equal to the period ($p_{\tau_i^k}$) of the respective task (τ_i^k).
4. Standalone execution time of a task is then given by: $e_{\tau_i^k} = p_{\tau_i^k} \cdot U_{\tau_i^k}$. We ensure the schedulability of the generated task set using the exact schedulability conditions defined in [11]. Task sets that are not schedulable are discarded.

Tasks are divided into runnables which are functions that perform calculations (summing n integers, for instance) to simulate a workload. The standalone execution time of each runnable of a task is such that the sum of the execution times of all the runnables of a task, is equal to the standalone execution time of the task. At this point, we have not yet configured any spinlocks.

4.2 Experiments

In this experiment, we generated task sets, using the method explained above, with periods ranging between [5ms - 20ms], which is typical for automotive systems [13]. We configured 3 spinlocks using Tresos. All the tasks are configured to use all the 3 spinlocks. We generate task sets with 5, 10, and 20 tasks per core.

For the experiment, we select two control tasks for which we measure the maximum observed response time over several task sets (50 task sets each for 5, 10, and 20 tasks per core). The control tasks are the highest priority task and the lowest priority task on core C_1. Since all the tasks across all the cores use all 3 spinlocks, the control tasks on core C_1 will incur blocking from remote tasks.

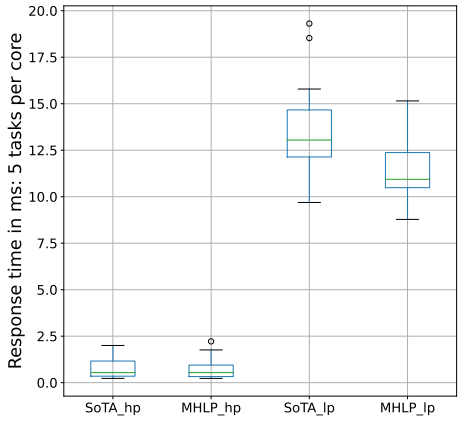
The experiment aims to quantify Equation (10). The highest priority control task does not suffer from preemptions by local tasks but may suffer local blocking as explained in lemmas 3.1, 3.2, and 3.3. The lowest priority control task suffers from preemptions from local higher priority tasks and contributes to the acquisition latency of remote tasks.

For a generated task set, we first measure the maximum observed response time under the state-of-the-art U|P spinlock of AutoCoreOS. For the same task set, we then measure the maximum observed response time under our implementation of M-HLP in AutoCoreOS. The measurements are dumped as a CSV file using a hardware debugger which is then used for further processing and generating the plots.

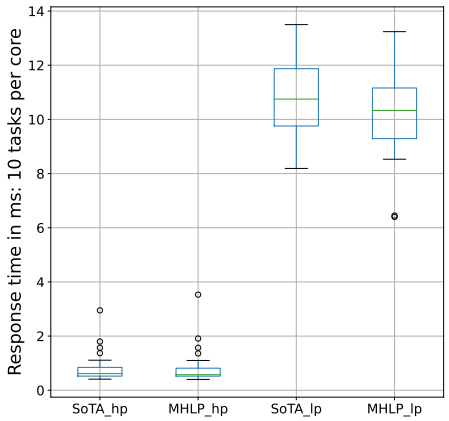
When the control task is the highest priority task, it suffers 0 preemptions from the other tasks on the same core, hence we can see (Figures 4a, 4b, and 4c) that the characteristics of U|P spinlock (SoTA_hp) and M-HLP (MHLP_hp) are similar even as the number of tasks increases from 5 to 20 tasks per core.

Some outliers in the box plot (for example, Figure 4b) were observed for the highest-priority control task signifying that the slight increase in the response time was due to the yielding of the higher-priority control task to a lower-priority task on its core. The outliers for the lowest-priority control task (SoTA_lp, MHLP_lp) show that the response time was better than the state-of-the-art.

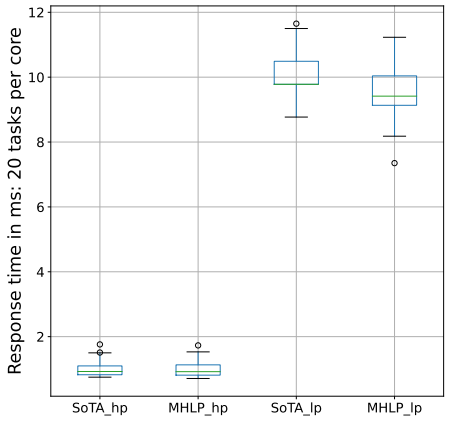
When the control task is the lowest priority task (SoTA_lp and MHLP_lp), it suffers preemptions from all the higher priority tasks on core C_1. Due to the strong progress guarantee of M-HLP, we can see that the response time of the lowest priority control task dominates the U|P spinlock. We can also validate the dominance of F|P spinlocks over U|P spinlocks



(a) Response times of control tasks with 5 tasks per core.



(b) Response times of control tasks with 10 tasks per core.



(c) Response times of control tasks with 20 tasks per core.

by observing that the median response times of the control tasks (highest as well as the lowest priority) under M-HLP are at most as much as under U|P spinlocks.

In the next experiment, we quantify Equation (5). Since the F|P spinlocks dominate U|P spinlocks, the blocking times under M-HLP will be at most as much as compared to U|P spinlock of AutoCoreOS. Spinlock is a busy wait mechanism and therefore, the blocking times are reflected in the execution time of the tasks (bloating). The dominance of M-HLP over U|P spinlock is reflected in the execution time bloating of the tasks.

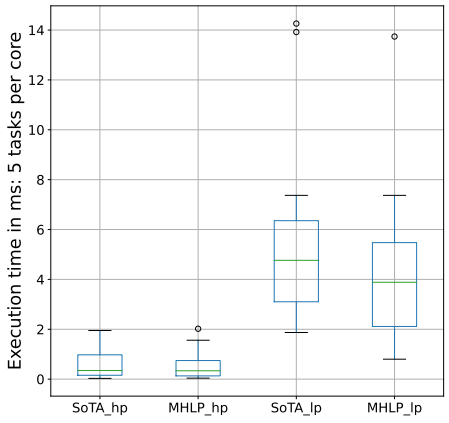
We generate task sets with 5, 10, and 20 tasks per core. The periods of the tasks range between [5ms - 20ms]. We configured 3 spinlocks and all the tasks across all cores use all the 3 spinlocks. We again select the highest and the lowest priority tasks on core C_1 as the control tasks and measure the maximum observed execution times of the tasks under both protocols for every generated task set. Note that we use the built-in execution time measurement API of the AutoCoreOS to measure the maximum execution time of the tasks.

It is clear from the box plots in Figures 5a, 5b, 5c, that as the number of tasks per core increase, the execution times of the highest-priority control task becomes almost equal. In Equation (5), the term $I_{\tau_i^k}(R_G^x)$ will be zero because the highest priority task will not be preempted by any other task on its core. Therefore, the standalone execution time of the task is bloated by the time its request is queued in the FIFO queue of the requested resource ($B_{\tau_i^k}(R_G^x)$ in Equation (5)). Due to the strong progress guarantee of M-HLP, the bloating of execution time is at most as much as compared to U|P spinlock of AutoCoreOS. Moreover, as can be seen in the outliers, for example in Figure 5c, the maximum observed execution time was significantly less under M-HLP which validates the dominance of F|P spinlocks over U|P spinlocks.

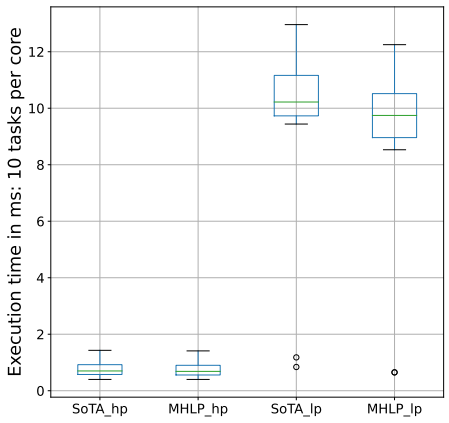
There is a significant difference in the execution time for the lowest-priority control task. In Equation (5), the lowest priority control task suffers from preemptions from all other higher-priority tasks on its core. The bloating caused in this case is considerably less under M-HLP because of the FIFO ordering of the request. Under U|P spinlocks, when the lower-priority task polls for the lock it can, in principle, always lose the arbitration to a remote task and spin for a longer time. Whereas under M-HLP, when the request reaches the head of the queue (task's ticket is equal to the "now serving" counter), it is guaranteed that the task will acquire the lock as soon as the task polls the lock.

We can see that the execution time of the lowest priority control task is less under M-HLP as compared to U|P spinlocks, signifying that the control task incurs less remote blocking under M-HLP as compared to U|P spinlocks.

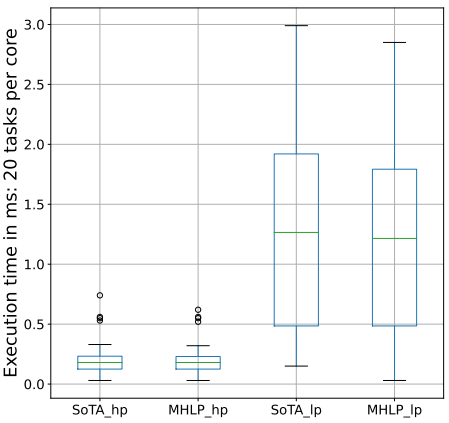
We conclude, with a sound empirical evaluation, that M-HLP dominates the U|P spinlock of AutoCoreOS.



(a) Execution times of control tasks with 5 tasks per core.



(b) Execution times of control tasks with 10 tasks per core.



(c) Execution times of control tasks with 20 tasks per core.

5 Conclusion and Future Work

We have implemented a FIFO-ordered spinlock mechanism for an AUTOSAR-compliant mRTOS using a ticketing system. The protocol provides strong fairness and guarantees freedom from starvation and deadlocks. We have shown that M-HLP can be implemented with relatively low overhead in a complex industrial mRTOS, AutoCoreOS.

We have developed a mathematical framework for analyzing the blocking time a task incurs due to local and remote tasks and derived bounds for the same. We conducted experiments that validate the dominance of FIFO-ordered spinlocks over U|P spinlocks. Evaluations take into account the characteristics of the highest as well as the lowest priority tasks on a core which are stressed by other tasks on the same as well as remote cores. Since all the tasks access all the configured spinlocks, the control tasks are always stressed while requesting access to a spinlocked resource. We have shown that M-HLP suffers significantly less blocking as compared to the U|P spinlock of a state-of-the-art mRTOS. How M-HLP performs against other F|P spinlocks is left as a future work.

Although we allow nesting of spinlock-protected *gcs*, we do not analyse M-HLP for such scenarios. The future work holds a thorough analysis of M-HLP under nesting. For nested spinlocks, we must consider the effect of raising a task's priority when the task acquires a spinlock. Several scenarios, such as nesting of local critical sections within spinlock-protected sections and vice-versa must be considered.

Although FIFO ordering provides a strong progress guarantee, it does not take into consideration the latency timing constraints of the tasks. For instance, consider resource requests from two distinct tasks for the same resource. One request has a tight timing requirement (hard deadline) whereas the other request has a more lenient timing requirement. Due to FIFO ordering, the request with a tighter constraint may be queued after the request which has a lenient timing constraint. This may lead to deadline misses and potential safety-critical issues. Such timing constraints are common in automotive, and more broadly, safety-critical real-time systems. We leave the study and analysis of the effects of such constraints on global resource sharing for future work.

Acknowledgements

This work has been supported by (1) the FFG under contract No.881844: "Pro²Future" (Products and Production Systems of the Future), Graz University of Technology (TU Graz), and Elektrobit Automotive GmbH in the joint research project CompEAS and (2) the TU Graz Open Access Publishing Fund.

References

[1] AUTOSAR Specification of Operating System. https://bit.ly/3kFJ4SC. Accessed January 2023.
[2] A. Wieder and B. B. Brandenburg. 2013. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spinlocks. RTSS '13.
[3] Wieder, A. "Blocking analysis of spin locks under partitioned fixed-priority scheduling." Ph.D. dissertation, Saarland University, 2018.
[4] Brandenburg, B & Anderson, J & Baruah, S & Härtig, H & Prins, J & Donelson, F & Paul, S & Mckenney, E. (2012). B. B. BRANDENBURG: Scheduling and Locking in Multiprocessor Real-Time Operating Systems.
[5] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," ICDCS, Paris, France, 1990
[6] K. Lakshmanan, D. de Niz and R. Rajkumar, "Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors," RTSS'09.
[7] S. Afshar, M. Behnam, R. J. Bril and T. Nolte, "Flexible spin-lock model for resource sharing in multiprocessor real-time systems," SIES'14.
[8] David P. Reed & Rajendra K. Kanodia. 1979. Synchronization with eventcounts and sequencers. Commun. ACM, Feb. 1979.
[9] J. Mellor-Crummey & M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Trans. Comp. Sys '91.
[10] M. Coutinho, J. Rufino and C. Almeida, "Response Time Analysis of Asynchronous Periodic and Sporadic Tasks Sheduled by a Fixed Priority Preemptive Algorithm," ECRTS'08.
[11] J. Lehoczky, L. Sha and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," RTSS'89.
[12] L. Sha, R. Rajkumar & J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," in IEEE Trans on Computers '90.
[13] J. Peleska et al. A Real-world Benchmark Model for Testing Concurrent Real-time Systems in the Automotive Domain. ICTSS'11.
[14] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. ACM Trans. Comput. Syst. 1997.
[15] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok and C. Siemers, "An efficient spin-lock based multi-core resource sharing protocol," IPCCC'14.
[16] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A Flexible Real-Time Locking Protocol for Multiprocessors," RTCSA'12.
[17] Baker, T. A stack-based resource allocation policy for realtime processes. RTSS'91.
[18] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. RTSS'01.
[19] D. Griffin, I. Bate & R. I. Davis, "Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests," RTSS'20.
[20] TC38x 32-Bit Single-Chip Microcontroller. Infineon Technologies AG. https://shorturl.at/hiDEJ. Accessed March 2023.
[21] D. Griffin, I. Bate, and R. I. Davis. Dirichlet-Rescale (DRS) algorithm software: https://doi.org/10.5281/zenodo.4118059.
[22] Davis, Robert & Griffin, David & Bate, Iain. A Framework for Multicore Schedulability Analysis Accounting for Resource Stress and Sensitivity. Real-Time Syst'22.
[23] Emberson, P. & Stafford, R. & Davis, R.I.. (2010). Techniques For The Synthesis Of Multiprocessor Tasksets. WATERS'10.
[24] Min-Allah, N., Khan, S.U., Ghani, N. et al. A comparative study of rate monotonic schedulability tests. J Supercomput '12.