Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions

Pratyush Patel[1], Iljoo Baek[1], Hyoseung Kim[2], Ragunathan (Raj) Rajkumar[1]
[1]*Carnegie Mellon University,* [2]*University of California, Riverside*

***Abstract*—Hardware accelerators such as GP-GPUs and DSPs are being increasingly used in computationally-intensive real-time and multimedia systems. System efficiency is often increased when CPU tasks suspend while using these devices. In this paper, we extend the existing Multiprocessor Priority Ceiling Protocol (MPCP) schedulability analysis in this particular context. We present three methods to improve the traditional MPCP analysis that reduces pessimism in analyzing blocking times. Two of these methods, the request-driven and the job-driven approaches, are motivated by prior work and are adapted to MPCP. The third combines these two approaches in a novel way to consistently outperform either on its own. We note that our underlying observations are general, and that such methods can also be used for analyzing other real-time synchronization protocols.**

Experimental results indicate that our analytical improvements result in a significantly higher schedulability compared to the traditional recursion-based analysis, even when self-suspensions are not considered. Our approach is also competitive with and often outperforms the linear-programming-based FMLP+ analysis, while having a considerably lower runtime complexity. We further substantiate the practical feasibility of suspension-based MPCP and examine its benefits over the busy-waiting approach by presenting a case-study on an NVIDIA TX2 embedded platform using real-world vision applications.

I. Introduction

Task synchronization and communication are key aspects of building complex systems as they enable components to safely share logical and physical resources. Conventional methods for synchronization, such as using mutex locks or binary semaphores, may lead to unbounded priority inversion [36, 38]. In order to overcome this problem, several real-time synchronization protocols have been proposed in the literature which reduce and bound the blocking time [11, 16, 17, 23, 35].

Synchronization protocols for uniprocessors, such as the Priority Inheritance Protocol (PIP), Priority Ceiling Protocol (PCP) [38] and Stack Resource Policy (SRP) [8], have been well-studied in prior work and their schedulability analyses are considered reasonably accurate in practice. However, when it comes to multiprocessor synchronization protocols such as the Multiprocessor Priority Ceiling Protocol (MPCP) and the Distributed Priority Ceiling Protocol (DPCP), traditional recursion-based analysis [29] has been shown to be quite pessimistic [13, 45]. Hence, practically schedulable tasksets under a given scheduling algorithm may be deemed to be unschedulable. Due to the increasing demand for computational capacity in addition to efficient resource sharing (e.g., in self-driving vehicles [28, 41]), multiprocessor synchronization protocols are of significant practical interest. However, the aforementioned analytical pessimism can lead to expensive over-provisioning. This has motivated the development of tighter analytical methods, such as the recent work on a Linear Programming (LP) approach to analyze the FIFO Multiprocessor Locking Protocol (FMLP+) [14]. This LP-based approach can also be applied to MPCP, DPCP and other synchronization protocols [13, 46]. Our work extends this line of research by significantly reducing the pessimism of the traditional recursion-based analysis, and our insights can also be incorporated into LP-based approaches to further improve their schedulability. In particular, we focus on the traditional recursion-based analysis due to its lower computational requirements. Such an approach, coupled with our analytical enhancements, can easily be adopted to runtime admission control and adaptation, which are integral parts of systems operating in changing environments.

Furthermore, we observe that there are two important aspects to task suspension when dealing with real-time synchronization protocols. The first is that of suspending *immediately after a resource request*, if the resource lock is already held by another task. This aspect has been well-studied in the literature [12, 13, 14, 16, 29], which broadly classifies schedulability analyses into "spinlock-based" and "semaphore-based" approaches. Spinlock-based approaches assume that a task requesting access to a resource busy-waits on the CPU until it obtains resource access, thereby disallowing other tasks to execute in the meantime. On the contrary, semaphore-based approaches allow requesting tasks to suspend if the requested resource is in use, thereby letting other tasks utilize the CPU. Although it may be intuitive to expect semaphore-based approaches to always yield better schedulability, both approaches are often considered in practice. This is because spinlock-based approaches do not face significant analytical pessimism and may sometimes outperform semaphore-based analyses, especially for small critical sections [12, 16, 29]. The improvements we propose in this paper are applicable to both these approaches.

The second aspect to task suspension is that of suspending *within critical sections*. Most existing synchronization protocols assume that tasks may not suspend during critical-section execution, which may seem reasonable, because critical sections that modify only a few shared variables are expected to be short in practice. However, synchronization protocols have recently been demonstrated to be effective in accessing shared external resources (such as GPUs and DSPs),

which may require much longer critical sections [20, 22]. In such cases, busy-waiting for the entire critical-section duration significantly compromises CPU utilization [26, 27]. To avoid this problem, it is becoming useful in practice to extend synchronization protocols to allow suspensions within critical sections while the resource is being used. We emphasize that system support for such suspensions already exists. For example, as we explore in Section IV-B, modern GPU hardware and drivers already allow a task to suspend while the GPU is executing its request. Recent work has briefly explored how such self-suspensions may be incorporated into the analysis of FMLP+ [14] and also to certain k-exclusion protocols [44], but, to the best of our knowledge, there is no prior evaluation regarding the schedulability characteristics of suspension-based synchronization protocols and their efficacy in practice.[1] In particular, no such analysis and experiments exist for MPCP in prior work.

In this paper, we generalize MPCP to the aforementioned use-case of suspending within critical sections and examine its practical feasibility. MPCP is a well-studied synchronization protocol and is known to offer comparable performance to other synchronization protocols under partitioned scheduling [13, 37]. We are further motivated to consider MPCP in this paper because it complements the widely-used fixed-priority scheduling approach.

The remainder of this paper is organized as follows. Section II describes our system model and reviews basic definitions and notations. Section III presents three improved blocking analyses for MPCP that also allow self-suspensions within critical sections. Section IV-A contains extensive schedulability experiments comparing our approach with the existing recursion-based analysis for MPCP, demonstrating the benefits of our approach. Section IV-A also compares the traditional MPCP analyses against the LP-based analysis for partitioned FMLP+, with and without self-suspending critical sections. Section IV-B evaluates the practical feasibility of our approach on the NVIDIA TX2 board, using the GPU as an example of a shared heterogeneous resource. Section V reviews prior related work. Section VI concludes the paper.

II. System Model

We consider a taskset $\Gamma = \{\tau_1, \ldots, \tau_n\}$ consisting of n sporadic tasks (subscript denotes the task index). This taskset executes on a multiprocessor platform that consists of m identical CPUs $P_1, \ldots, P_m$. Each task may use any of g serially-reusable shared resources (such as shared memory, heterogeneous accelerators, etc.), whose access is arbitrated by mutually-exclusive locks, denoted by $R_1, \ldots, R_g$ respectively. A task must acquire a lock in order to use the corresponding resource. A task executing on the CPU without holding any locks is said to be in a *non-critical section*. A task is said to be in a *critical section* if it is holding a lock. While executing a critical section, a task may be scheduled on the CPU or it may be suspended. All critical sections are assumed to be non-nested.

For each task τ_i, C_i denotes the cumulative worst-case execution time of all *non-critical sections*, T_i denotes the minimum inter-arrival time of each job and D_i denotes the relative deadline. We assume constrained deadlines for all tasks, i.e., $D_i \leq T_i$. Each task has a maximum of η_i critical sections per job, and uses R_k for a maximum of $\eta_{i,k}$ times in any single job ($\eta_i = \sum_{1 \leq k \leq g} \eta_{i,k}$). $G_{i,j}$ denotes the worst-case execution time of the j^{th} critical section of τ_i, and consists of two components: *(i)* $G^m_{i,j}$, which denotes the worst-case execution time of the critical section on the CPU, and *(ii)* $G^e_{i,j}$, which denotes the worst-case suspension time of that critical section (for instance, while executing on the shared resource). Note that, $G_{i,j} \leq G^m_{i,j} + G^e_{i,j}$, as $G^m_{i,j}$ and $G^e_{i,j}$ may not occur on the same control path. The total critical-section execution time for a task is given by $G_i = \sum_{j=1}^{\eta_i} G_{i,j}$. Correspondingly, $G^m_i = \sum_{j=1}^{\eta_i} G^m_{i,j}$ and $G^e_i = \sum_{j=1}^{\eta_i} G^e_{i,j}$. The maximum number of suspensions in the j^{th} critical section of τ_i is given by $\zeta_{i,j}$. We use E_i to denote the worst-case CPU execution time of task τ_i, where $E_i = C_i + G^m_i$.

$R(\tau_i)$ denotes the complete set of locks accessed by τ_i, and similarly, $R(\tau_{i,j})$ denotes the lock protecting the j^{th} critical section of task τ_i. We define the total utilization of the task as $U_i = (C_i + G_i)/T_i$. For simplicity, we do not consider release jitters in this paper.

We assume preemptive, partitioned fixed-priority scheduling, where each task τ_i is associated with a unique base priority π_i. τ_j has a higher base priority than τ_i if $\pi_j > \pi_i$. $P(\tau_i)$ denotes the set of tasks allocated on the same CPU as τ_i. We use $hp(\tau_i)$ (resp. $lp(\tau_i)$) to denote the set of all tasks which have a priority higher (resp. lower) than τ_i. Similarly, we use $hpp(\tau_i)$ (resp. $lpp(\tau_i)$) to denote the set of higher (resp. lower)-priority tasks allocated on the same CPU as τ_i.

Multiprocessor Priority Ceiling Protocol. Throughout this paper, we assume that locks are governed using MPCP, which was originally proposed for partitioned fixed-priority scheduling for shared-memory multiprocessors [34, 36]. In particular, we consider semaphore-based MPCP. Under this protocol, a task requesting access to a lock held by a different task is suspended and inserted into a priority queue corresponding to that lock. During this time, other ready tasks may use the CPU. When the lock is released, the task at the head of the priority queue is woken up, granted lock access, and is immediately elevated to a ceiling priority. The ceiling priority of the j^{th} critical section of task τ_i accessing lock R_k, is given by $\pi_{i,j} = \Pi_k + \Pi_B$, where Π_B is a priority level greater than the base priority of any task in the system and Π_k is the highest base priority of any task that uses R_k. When τ_i releases a lock, it is returned to its corresponding base priority π_i.

III. Suspension-Based Schedulability Analysis

Taskset schedulability is typically determined by using response-time analysis, which deems a taskset schedulable if each task's worst-case response time (WCRT) is no greater

[1] In this paper, we refer to "suspension-based" synchronization protocols as those which allow suspensions *within* critical sections, and to "busy-waiting" protocols as those which do not.

than its deadline. Under synchronization protocols, response-time analysis requires an upper bound on the protocol-imposed blocking duration for each task. In this section, we discuss three types of blocking analyses that help determine taskset schedulability. Prior work has shown that the WCRT W_i of a task τ_i under fixed-priority scheduling can be obtained by solving the following recurrence equation [7, 29]:

$$W_i = C_i + G_i + B_i + \sum_{\tau_h \in hpp(\tau_i)} I_{i,h}, \tag{1}$$

where, B_i denotes the upper bound on the maximum synchronization-based blocking delay faced by τ_i and $I_{i,h}$ denotes the worst-case interference (preemption time) incurred by τ_i due to the execution of task τ_h on the same CPU. A taskset is considered schedulable iff $\forall \tau_i, W_i \leq D_i$.

A. Higher-Priority Task Preemptions

A task τ_i can be preempted by all higher-priority tasks executing on the same CPU as τ_i which increases its WCRT.

Def. 1. *For each higher-priority task τ_h, $\alpha_{i,h}$ is defined as an upper bound on the number of instances of τ_h released during the execution of a single job of τ_i. $\alpha_{i,h}$ is given by*

$$\alpha_{i,h} = \left\lceil \frac{W_i + W_h - E_h}{T_h} \right\rceil. \tag{2}$$

Eq. 2 takes into account the jitter incurred due to self-suspensions ($W_h - E_h$), as it affects the worst-case start time of the job. A detailed proof can be found in prior work on self-suspension analysis [10]. We incorporate the maximum preemption delay from each higher-priority task by using

$$I_{i,h} = \alpha_{i,h} \cdot E_h, \tag{3}$$

because, by definition, τ_h executes for a maximum of E_h on the CPU per job release.

B. Blocking Definition

A task τ_i is said to be blocked if a local task τ_j with a lower base priority is scheduled while τ_i is pending (because τ_j executes at its ceiling priority when granted lock access), or if any task τ_k has locked the resource that τ_i is waiting for. We define blocking time B_i as the maximum amount of time that any job of τ_i is blocked. Note that, due to the inclusion of self-suspensions with critical sections, we do not explicitly distinguish between local blocking (caused due to priority inversions) and remote blocking (caused due to lock-acquisition delays) as done in prior work [29, 45]. Under suspension-based approaches, a lower-priority task τ_l may acquire a resource and suspend, which a higher priority task τ_h (on the same CPU as τ_l), may have to wait for, if it makes a request for the same resource after τ_l's suspension. Such cases do not arise in the traditional busy-waiting approach. Whenever critical sections are known to not suspend, we can revert to the original separation of local and remote blocking to obtain higher schedulability.

The blocking term can be decomposed into three types: *(i)* direct blocking, which is incurred when any task τ_j is using τ_i's requested resource, *(*ii) indirect blocking, which is incurred when a task τ_x accessing a resource with a higher-priority ceiling preempts the execution of τ_j, which is holding the resource that τ_i is waiting for, and *(iii)* prioritized blocking, which is incurred when lower-priority tasks executing with priority ceilings preempt the CPU execution of τ_i.[2]

Task	C_i	G_i	η_i	$G_{i,1}$	$G_{i,2}$	T_i = D_i	CPU
τ_1	1	1	1	1	-	102	1
τ_2	1	100	1	100	-	10000	2
τ_3	1000	2	2	1	1	1106	3

TABLE I: Example taskset parameters.

C. Blocking Analysis Intuition

We begin by making a few observations regarding the blocking term and later capture this intuition in our analysis. Let us consider an example taskset with parameters given in Table I. We assume that each task is allocated to a separate CPU and that they all access a single lock R_1 for all their critical sections. We also assume that no task has self-suspending critical sections.

We now consider the response time for task τ_3. First, as each task has exclusive access to its own CPU, prioritized blocking and the interference from higher-priority tasks (corresponding to the last term of Eq. 1), is disregarded for all tasks including τ_3. Secondly, because all tasks share a single resource, there is no indirect blocking delay. Therefore, τ_3 only incurs direct blocking from τ_1 and τ_2.

We now describe two methods to evaluate this direct blocking delay. We then make observations to combine the benefits from both methods to introduce a new method which results in a tighter bound on the blocking term. Our first method uses a *request-driven approach*, which is analogous to the blocking analysis presented by Lakshmanan et al. [29]. The request-driven approach individually considers the worst-case blocking delay from *all* tasks for *each* resource request made by τ_3. Because τ_3 makes two resource requests in a single job, its request-driven blocking delay, B_3^{dr}, is given by $B_{3,1}^{dr} + B_{3,2}^{dr}$, where $B_{3,1}^{dr}$ and $B_{3,2}^{dr}$ correspond to the worst-case direct blocking caused by higher-priority task requests during the first and second resource requests by τ_3, respectively. Each resource request of τ_3 may be delayed by up to two requests of τ_1 and one request of τ_2 in the worst case. This is because τ_1 may issue a second resource request while τ_2 executes its critical section. Therefore, $B_{3,1}^{dr} = B_{3,2}^{dr} = 102$, and $B_3^{dr} = 204$.

Our second method uses a *job-driven approach* to evaluate the worst-case blocking term, and takes into account the maximum number of resource requests made by each higher-priority task during the entire execution time of a single job

[2]Prioritized blocking has sometimes been referred to as "preemption blocking" in prior work. We avoid using "preemption blocking" in this paper as it could be confused with the usual higher-priority task preemptions.

of τ_3. Because no more requests can be made by higher-priority tasks within τ_3's execution duration, this approach also results in a safe blocking bound. The job-driven approach is inspired by the memory-interference bounding [25] and server-based GPU arbitration [27] analyses. It takes into account the *entire* execution time of a job of τ_i to find its worst-case direct blocking delay. The total number of higher-priority resource requests during the execution of a job of τ_3 can be captured by a ceiling term similar to traditional response-time analysis (wherein the total number of higher-priority job releases are taken into account, as described in Eq. 7). Such an analysis amounts to 12 requests from τ_1 and a single request from τ_2, resulting in a total job-driven blocking delay of $B_3^{dj} = (12 \times 1) + (1 \times 100) = 112$.

Although the job-driven approach results in a tighter blocking bound than the request-driven approach for the considered taskset, we posit that both these approaches can be unnecessarily pessimistic.[3] A third approach is possible which bounds the worst-case blocking delay of each request of τ_3, while simultaneously bounding the maximum possible requests that can be issued by each higher-priority task. This effectively *combines the benefits* offered by the request-driven and the job-driven approaches. Under this *hybrid approach*, we capture the maximum blocking using B_3^{dm}. We observe that due to the large period of τ_2, the maximum number of resource requests it can issue during a single job of τ_3 can only be 1. On the other hand, while τ_1 can issue a maximum of 12 requests during the execution of τ_3, each request of τ_3 can only be delayed by up to two such requests. This makes the total blocking delay $B_3^{dm} = (2 \times 2 \times 1) + (1 \times 100) = 104$, which is lower than both the request-driven and the job-driven approaches.

Key Observations. On the one hand, the request-driven approach can result in a pessimistic blocking delay as η_i increases because the *worst-case* blocking is unnecessarily taken into account for *each* request of τ_i. On the other hand, the job-driven analysis can result in a pessimistic bound as C_i (and correspondingly W_i) increases, because it considers that resource requests can block τ_i even when τ_i is *not attempting* to use a resource. Therefore, the hybrid approach can yield a tighter bound by carefully combining the best of both worlds. It is worth noting that the hybrid approach that we propose in this paper differs from the prior attempts [25, 27] that simply take the minimum of the request-driven and the job-driven approaches, i.e., $B_3^{dm} < \min(B_3^{dr}, B_3^{dj})$. We formalize these observations in our subsequent blocking analyses for MPCP.

D. Direct Blocking

The direct blocking for a task τ_i captures its total waiting duration before lock access is granted, across all its lock-acquisition requests. We analyze it as follows.

Request-Driven Approach. This approach is similar to the blocking analysis proposed by Lakshmanan et al. [29]. However, we modify the original blocking term by incorporating the suspension-based analysis proposed by Chen et al. [18] to obtain a tighter bound. The maximum direct-blocking delay a task can incur under this approach is calculated by summing the blocking delay for individual lock-acquisition requests, treated separately. This means that, each time a job of task τ_i issues a request to obtain any lock it may be blocked by other tasks that use the same lock. The worst-case blocking for each lock-acquisition request is summed up, resulting in the total direct blocking under this approach. Therefore,

$$B_i^{dr} = \sum_{0<j\leq\eta_i} B_{i,j}^{dr}. \tag{4}$$

Lemma 1. *Under the request-driven approach, the worst-case direct blocking for the j^{th} lock-acquisition request of τ_i is upper-bounded by*

$$B_{i,j}^{dr} = \max_{\substack{\tau_l \in lp(\tau_i) \\ \wedge R(\tau_{i,j})=R(\tau_{l,k})}} (H_{l,k}) + \sum_{\substack{\tau_h \in hp(\tau_i) \\ \wedge R(\tau_{i,j})=R(\tau_{h,k})}} \beta_{i,j,h} \cdot H_{h,k}, \tag{5}$$

where, $H_{x,k}$ is the worst-case response time of the k^{th} critical section of τ_x (discussed in Section III-E[4]) and $\beta_{i,j,h}$ is an upper bound on the number of activations of τ_h during the blocking duration $B_{i,j}^{dr}$ (similar to $\alpha_{i,h}$ in Eq. 2). $\beta i, j, h$ is given by

$$\beta_{i,j,h} = \left\lceil \frac{B_{i,j}^{dr} + W_h - E_h}{T_h} \right\rceil. \tag{6}$$

Proof. When τ_i makes its lock-acquisition request, it may have to wait for at most one lower-priority task that is already holding the lock and for higher-priority tasks that are already queued to access the lock. In addition, while these requests are serviced, higher-priority tasks on different CPUs or on the same CPU (assuming self-suspending critical sections), may issue additional locking requests which will be serviced first.[5] The first term in Eq. 5 captures the maximum blocking caused by such a critical section of a lower-priority task. The critical section with the highest response time is used to reflect the worst case. The second term in Eq. 5 captures the maximum blocking delay caused by critical sections of higher-priority tasks. Eq. 6 accounts for deferred execution when upper-bounding the number of higher-priority task requests, which represents the same blocking delay as derived in prior work [10, 18, 23]. Since both these factors can occur independently, the upper bound on direct blocking under the request-driven approach is given by the sum of these two terms. ■

Job-Driven Approach. The job-driven approach captures job-level characteristics (rather than per request) by bounding the maximum possible interference during the entire execution of any job of τ_i. The key intuition is that this can be bounded

[3] Note that, neither the request-driven approach nor the job-driven approach dominates the other in general, as shown in Section IV-A.

[4] $H_{x,k}$ is analogous to $W'_{i,k}$ in Eq. (2) by Lakshmanan et al. [29].

[5] If critical sections were assumed to not suspend, the second term of Eq. 5 would only involve *remote* higher-priority task requests.

by considering the maximum number of releases (and hence lock-acquisition requests) of each higher-priority task before the completion of τ_i.

Lemma 2. *Under the job-driven approach, the worst-case direct blocking incurred by τ_i is given by*

$$B_i^{dj} = \sum_{r \in R(\tau_i)} \eta_{i,r} \cdot \max_{\substack{\tau_l \in lp(\tau_i) \\ \wedge R(\tau_{l,k}) = r}} (H_{l,k}) + \sum_{\substack{\tau_h \in hp(\tau_i) \\ \wedge R(\tau_{h,k}) \in R(\tau_i)}} \alpha_{i,h} \cdot H_{h,k}. \tag{7}$$

Proof. The maximum direct blocking caused by lower-priority tasks is captured by the first term of Eq. 7 across each of the $\eta_{i,r}$ lock-acquisition requests of τ_i (analogous to the request-driven approach). In addition, during the execution of a job of τ_i, each higher-priority τ_h can be released for a maximum of $\alpha_{i,h}$ times (from Def. 1). Each higher-priority task release implies that all its critical sections that use the same resources as τ_i can block τ_i in the worst case. This is captured in the second term. Since both these factors can occur independently, the upper bound on the direct blocking delay under the job-driven approach is given by the sum of these two terms. ■

Hybrid Approach. The hybrid approach combines the benefits of both the request-driven and the job-driven approaches to result in a tighter blocking bound. We consider the direct blocking delay from higher-priority (B_i^{dmh}) and lower-priority (B_i^{dml}) tasks separately, i.e., $B_i^{dm} = B_i^{dml} + B_i^{dmh}$.

Lemma 3. *Under the hybrid approach, the worst-case direct blocking due to higher-priority tasks is given by*

$$B_i^{dmh} = \sum_{\substack{\tau_h \in hp(\tau_i) \\ \wedge R(\tau_{h,k}) \in R(\tau_i)}} \delta_{i,h} \cdot H_{h,k}, \tag{8}$$

where, $\delta_{i,h}$ upper-bounds the cumulative number of requests by τ_h to the locks accessed by critical sections of τ_i. $\delta_{i,h}$ is given by

$$\delta_{i,h} = \min \left(\alpha_{i,h}, \sum_{\substack{0 < j \leq \eta_i \\ \wedge R(\tau_{i,j}) \in R(\tau_h)}} \beta_{i,j,h} \right). \tag{9}$$

Proof. For each higher-priority task τ_h, the maximum number of times its critical sections can block τ_i is safely upper-bounded by $\alpha_{i,h}$ under the job-driven approach (Lemma 2). Alternatively, it is also safely upper-bounded under the request-driven approach in which the j^{th} lock-acquisition request of τ_i incurs direct blocking due to a maximum of $\beta_{i,j,h}$ activations of τ_h if also τ_h accesses the same lock. As both of these safe upper bounds, their minimum represented by $\delta_{i,h}$ is also safe. Therefore, Eq. 8 is obtained by summing this term across all the higher-priority task critical sections that access the same locks as τ_i, as the direct blocking due to other critical sections is zero. ■

Next, to characterize the direct blocking due to lower-priority tasks, we define the following terms.

Def. 2. *$\theta_{i,l}$ is defined as an upper bound on the number of instances of a lower-priority task τ_l that may be active during the execution of τ_i, and is given by*

$$\theta_{i,l} = \left\lceil \frac{W_i + D_l - E_l}{T_l} \right\rceil. \tag{10}$$

Unlike Eq. 2, we use D_l rather than W_l in the ceiling term. This is because we begin calculating WCRTs starting from the highest-priority task under schedulability analysis, and therefore we cannot know the WCRT (W_l) of any of the lower-priority tasks while analyzing τ_i [18].

Def. 3. *$Q_{i,j}$ is defined as the set of worst-case response times of critical sections that access lock R_j and belong to tasks that have a lower priority than τ_i. Hence, $Q_{i,j} = \{H_{l,x} | \tau_l \in lp(\tau_i) \wedge R(\tau_{l,x}) = j\}$.*

We use $L_{i,j,k}$ to denote the k^{th} longest critical section in $Q_{i,j}$ and $X_{i,j,k}$ to denote the task index corresponding to $L_{i,j,k}$ (recall that tasks are indexed from 1 to n).[6]

Lemma 4. *Under the hybrid approach, the worst-case direct blocking due to lower-priority tasks is given by*

$$B_i^{dml} = \sum_{R_j \in R(\tau_i)} \sum_{0 < k \leq |Q_{i,j}|} \psi_{i,j,k} \cdot L_{i,j,k}, \tag{11}$$

where, $\psi_{i,j,k}$ upper-bounds the maximum number of times that $L_{i,j,k}$ can block τ_i. $\psi_{i,j,k}$ is given by

$$\psi_{i,j,k} = \min \left(\eta_{i,j} - \sum_{0 < t < k} \psi_{i,j,t}, \theta_{i,X_{i,j,k}} \right). \tag{12}$$

Proof. The maximum direct blocking delay caused by lower-priority tasks for a single lock-acquisition request of τ_i to lock R_j is equal to the longest WCRT of any of their critical sections that access R_j. However, τ_i only locks R_j for up to $\eta_{i,j}$ times in a single job. This forms an upper bound on the number of times this critical section can block τ_i. In addition, because the lower-priority task corresponding to the longest critical section is activated up to $\theta_{i,X_{i,j,1}}$ times during the execution of τ_i (by definition, $L_{i,j,1}$ is the critical section with the longest WCRT, and $X_{i,j,1}$ is its task index), this too forms an upper bound on the number of times this critical section can block τ_i.

Now, two possibilities arise: *(i)* If $\eta_{i,j} \leq \theta_{i,X_{i,j,1}}$, then no other lower-priority task's critical section that accesses R_j can block τ_i because this critical section can block each of τ_i's requests in the worst case, or *(ii)* If $\eta_{i,j} > \theta_{i,X_{i,j,1}}$, it is possible for other lower-priority task critical sections that access R_j to block τ_i, assuming such critical sections exist. At this point, the 2^{nd} longest critical section among lower-priority tasks, i.e. $L_{i,j,2}$, may block the remaining locking requests of τ_i to R_j for up to $\eta_{i,j} - \theta_{i,X_{i,j,1}}$ times (according to request-driven analysis) or up to $\theta_{i,X_{i,j,2}}$ times (according to job-driven analysis). The minimum term is taken to represent the tighter bound. This approach is safe because, at each step,

[6] Ties can be broken arbitrarily, but consistently.

the longest possible critical section is taken into account to reflect the worst case. Eq. 12 captures this aspect for the k^{th} longest critical section.

The two possibilities described above may arise again, and the same reasoning can be applied. The analysis is complete for lock R_j when the sum total of requests amounts to $\eta_{i,j}$, or if there are no more lower-priority task critical sections that can cause further blocking. Generalizing across all such critical sections in the set $Q_{i,j}$ yields the inner summation in Eq. 11. Finally, because R_j may represent any lock, the total lower-priority blocking is given by summing this term across all locks accessed by τ_i. ■

E. Indirect Blocking

As defined in Section III-B, indirect blocking is caused when a task τ_x executing its critical section preempts the critical section execution of a task τ_j, that task τ_i is waiting for. This occurs when the ceiling priority of the corresponding resource of τ_x is greater than the ceiling priority of the resource accessed by τ_j and τ_i. We capture indirect blocking on each critical section instance of τ_j and incorporate it into our analysis through its impact on direct blocking.

Lemma 5. *The worst-case indirect blocking incurred upon a single instance of the p^{th} critical section of τ_j is given by*

$$B_{j,p}^{ir} = (\zeta_{j,p} + 1) \left(\sum_{\substack{\tau_q \in P(\tau_j) \\ \wedge \pi_{q,k} > \pi_{j,p}}} \max(G_{q,k}^m) \right). \quad (13)$$

Proof. Before τ_j begins executing the p^{th} critical section (but right after it is granted lock access), in the worst case, it may be blocked by all the tasks that execute on the same CPU as τ_j with a higher critical-section priority than $\pi_{j,p}$ (second factor of Eq. 13). $G_{q,k}^m$ is used instead of $G_{q,k}$ because it is the upper bound on the CPU execution time of that critical section, which can cause such blocking. As captured in Eq. 13, each time the critical section suspends, the same requests may be re-issued by the other tasks up to a maximum of $\zeta_{j,p}$ times. Note that, requests from other tasks that access the same resource as $R(\tau_{j,p})$ are implicitly not considered under indirect blocking, because it is only caused by critical sections executing at a higher ceiling priority.

■

We use the indirect blocking per critical section from Eq. 13 to obtain the worst-case response time of that critical section($H_{j,x}$). The worst-case response time of the x^{th} critical section of task τ_j is given by

$$H_{j,x} = G_{j,x} + B_{j,x}^{ir}. \quad (14)$$

Hence, each time τ_i is directly blocked by the x^{th} critical section of τ_j, τ_i incurs an indirect blocking delay of $B_{i,x}^{ir}$, which is implicitly captured in the direct blocking analysis discussed in Section III-D.

F. Prioritized Blocking

Prioritized blocking captures the maximum time that non-critical sections of τ_i may be preempted by lower-priority tasks executing on the same CPU as τ_i. This occurs because, under MPCP, when the priority of a lock-holding lower-priority job is boosted, it effectively executes its critical section with a higher priority than τ_i's base priority. We analyze the prioritized blocking delay as follows.

Request-Driven Approach. The request-driven approach for prioritized blocking follows the analysis by Lakshmanan et al. [29], but we modify it to reflect critical-section suspensions.

Lemma 6. *Under the request-driven approach, the worst-case prioritized blocking is given by*

$$B_i^{pr} = (\eta_i + 1) \left(\sum_{\tau_l \in lpp(\tau_i)} \max(G_{l,k}^m) \right). \quad (15)$$

Proof. A non-critical section of τ_i can be blocked by any single critical section of each local lower-priority task due to the latter executing at ceiling priority. In the worst case, this is the longest CPU critical section of each local lower-priority task, as captured in the summation term of Eq. 15. However, such interference may occur at most (η_i+1) times in the worst case, because τ_i has $(\eta_i + 1)$ non-critical sections. ■

Job-Driven Approach. The job-driven approach for prioritized blocking takes into account the maximum number of activations of each lower-priority task during the execution of any job of τ_i. This approach can be quite pessimistic if several lower-priority resource-using tasks are scheduled on the same CPU, but can provide benefits over the request-driven approach for fewer low-priority tasks.

Lemma 7. *Under the job-driven approach, the worst-case prioritized blocking is given by*

$$B_i^{pj} = \sum_{\tau_l \in lpp(\tau_i)} \theta_{i,l} \cdot G_l^m. \quad (16)$$

Proof. A local lower-priority task τ_l can block τ_i's non-critical sections for a maximum of G_l^m per activation, and from Def. 2, there can be a maximum of $\theta_{i,l}$ such activations. Eq. 16 captures this effect across all local lower-priority tasks. ■

Hybrid Approach. To incorporate benefits from both the request-driven and the job-driven approaches, we upper-bound the lower-priority task activations by $\theta_{i,l}$ (job-driven aspect) and limit it by the maximum number of times such interference can occur across each non-critical segment of τ_i, i.e. $(\eta_i + 1)$ times (request-driven aspect).

Lemma 8. *Under the hybrid approach, the worst-case prioritized blocking is given by*

$$B_i^{pm} = \sum_{\tau_l \in lpp(\tau_i)} \sum_{0 < k \leq \eta_l} \phi_{i,l,k} \cdot S_{l,k}, \quad (17)$$

where, $S_{l,k}$ denotes the k^{th} longest CPU critical section of τ_l (i.e., sorted by decreasing $G_{l,x}^m$) and $\phi_{i,l,k}$ is an upper bound

on the number of times $S_{l,k}$ can preempt τ_i. $\phi_{i,l,k}$ is given by

$$\phi_{i,l,k} = \min\left(\eta_i + 1 - \sum_{0<t<k} \phi_{i,l,t}, \theta_{i,l}\right). \quad (18)$$

Proof. A single critical section of a local lower-priority task τ_l can block τ_i up to a maximum of $(\eta_i + 1)$ times. In the worst case, this is the longest such critical section ($S_{l,1}$). However, the same critical section can be released at most $\theta_{i,l}$ times during the execution of τ_i due to as many activations of τ_l in this duration. A similar line of reasoning as in Lemma 4 can be followed, which again results in two cases: *(i)* If $(\eta_i+1) \leq \theta_{i,l}$, then no other (smaller) critical sections from τ_l can cause greater blocking and the obtained minimum is safe, or *(ii)* if $(\eta_i+1) > \theta_{i,l}$, the second longest critical section (i.e. $S_{l,2}$) can cause additional blocking. This process terminates when the number of the critical section activations accounted for adds up to $(\eta_i + 1)$, or if all critical sections from τ_l have already been accounted for up to $\theta_{i,l}$ times. This is finally summed over all local lower-priority tasks in Eq. 17. ■

G. Putting It All Together

We summarize the conclusion of the above sections in the following theorem.

Theorem 1. *Under the request-driven approach, the worst-case blocking time B_i of a task τ_i that accesses shared locks protected by MPCP is upper-bounded by $B_i^{dr} + B_i^{pr}$; under the job-driven approach by $B_i^{dj} + B_i^{pj}$; and under the hybrid approach by $B_i^{dm} + B_i^{pm}$.*

Proof. By Lemmas 1 to 8 and the above discussion. Note that indirect blocking is already accounted for under the direct blocking terms for all the approaches. ■

The B_i obtained from Theorem 1 can be substituted in Eqs. 1 and 3 to obtain the WCRT of τ_i under suspension-based MPCP. Because the blocking analyses under the job-driven and the hybrid approaches depend on W_i, and because W_i is non-decreasing, it can be substituted back into the corresponding blocking analyses from the left-hand side of Eq. 1, until both sides converge (or until W_i exceeds D_i, at which point the taskset is unschedulable). As previously noted, we evaluate response times in decreasing order of task priority.

We briefly note how our enhancements may also benefit the LP-based analysis for MPCP [13]. First, our blocking model can be used to extend the LP-based approach to incorporate a notion of self-suspensions within critical sections. Secondly, we can introduce job-driven constraints per critical section for direct, indirect and prioritized blocking, by upper-bounding the number of locking requests before the completion of the analyzed job (given by Eqs. 2 and 10, respectively). Such constraints are not exactly captured by the existing LP-based approaches [13, 14], and our findings could further improve their schedulability.

Parameters	Values
Number of CPUs (m)	4
Number of shared resources (g)	[1, 3]
Number of tasks per CPU	[3, 6]
Percentage of tasks with critical sections	[10, 40]%
Task period and deadline ($T_i = D_i$)	[30, 500]ms
Utilization per CPU	[40, 60]%
Ratio of crit. sec. len. to non-crit. sec. len. (G_i/C_i)	[10, 30]%
Number of critical sections per task (η_i)	[1, 3]
Ratio of CPU-using operations in $G_{i,j}$ ($G^m_{i,j}/G_{i,j}$)	[10, 30]%
Number of suspensions in a critical section ($\zeta_{i,j}$)	[1, 2]

TABLE II: Base parameters for taskset generation.

IV. Evaluation

We first perform schedulability experiments to evaluate our proposed suspension-based MPCP analyses and compare them with prior work. We then present a case study of suspension-based MPCP on the NVIDIA TX2 and compare it with the busy-waiting MPCP approach. Note that we use semaphore-based protocols throughout this section (i.e., tasks suspend if the requested lock is already held by another task).

A. Schedulability Experiments

We contrast the proposed request-driven, job-driven and hybrid approaches for suspension-based MPCP with the traditional busy-waiting MPCP analysis [29] (henceforth referred to as "original analysis"). To understand the schedulability and performance characteristics of these traditional approaches compared to LP-based approaches, we also evaluate the LP-based analyses for the busy-waiting and suspension-based FMLP+ under partitioned scheduling [13, 14], denoted by `FMLP-BW` and `FMLP-SS`, respectively.

We use SchedCAT [5] in conjunction with the GNU Linear Programming Kit (GLPK) to evaluate the LP-based approaches. As the suspension-based analysis for FMLP+ was neither evaluated nor implemented in prior work, we implement it in SchedCAT by modifying Constraint 11 and incorporating Constraint 21 from the extended version of the original paper [13]. We also modify SchedCAT to make use of Eqs. 1, 2 and 3 for suspension-based response time analysis.

Taskset Generation. For each experimental setting, we evaluate $10,000$ randomly-generated tasksets based on the parameters given in Table II. Our base taskset parameters are inspired from prior work [13, 26, 29], but are modified in some experiments to match specific use cases such as the popular single-GPU use case in embedded platforms [24, 26, 27]. To generate each taskset, the number of CPUs, the utilization per CPU, the number of tasks per CPU and the number of resources (each arbitrated by a unique lock) are first determined by selecting these parameters uniformly at random based on Table II. Next, by using these parameters, individual task utilizations are allocated according to the well-studied UUniFast algorithm [9]. Task period T_i with a minimum granularity of 1 microsecond is chosen at random from the given task period range, and we consider implicit deadlines throughout our experiments ($D_i = T_i$). Among all generated

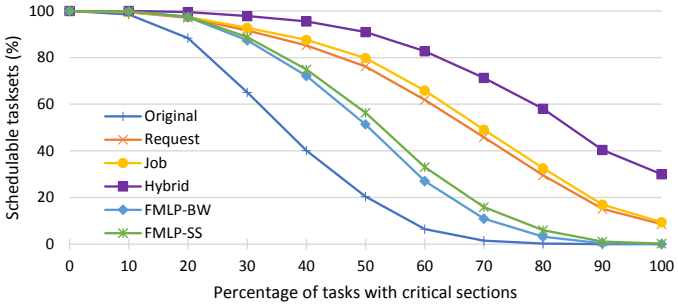


Fig. 1: Schedulability w.r.t. the percentage of tasks having critical sections. All critical sections access a single resource and suspend for their entire execution duration.

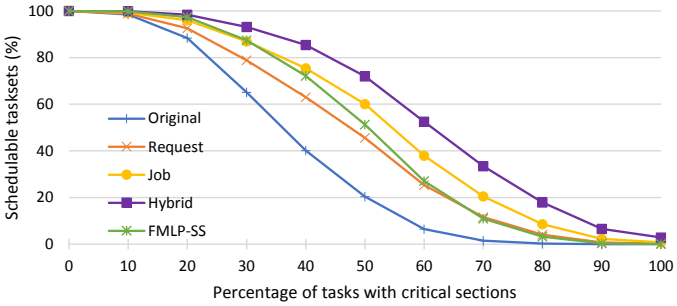


Fig. 2: Schedulability w.r.t. the percentage of tasks having critical sections. All critical sections access a single resource and busy-wait for their entire execution duration.

tasks, a certain number of tasks are designated to access critical sections based on the aforementioned taskset parameters. If the task τ_i does not access any lock (or, has no critical sections), then C_i is set to $(U_i \cdot T_i)$ and G_i is set to zero. Otherwise, G_i is selected such that G_i/C_i is consistent with Table II and $(C_i + G_i) = (U_i \cdot T_i)$. A random η_i is then chosen from the range provided in Table II, and G_i is split into η_i randomly-sized pieces to obtain $G_{i,j}$. These critical sections are assigned a random resource from the generated resource pool and are then instantiated with appropriate $G^m_{i,j}$, $G^e_{i,j}$, and $\zeta_{i,j}$ assuming that $G_{i,j} = G^m_{i,j} + G^e_{i,j}$. In the case of busy-waiting experiments or for analyses that enforce busy-waiting, G^m_i is set equal to G_i and G^e_i is set to zero. Finally, tasks are assigned priorities according to the rate-monotonic policy [31], with ties broken arbitrarily.

Results and Observations. We perform schedulability analyses using the aforementioned approaches on a variety of workloads and capture the percentage of schedulable tasksets in each experimental setting.

Figure 1 depicts the percentage of schedulable tasksets as the percentage of tasks sharing a single resource increases. Such a scenario may occur when several tasks access a single shared accelerator (such as a GPU), which reflects the majority of the available embedded platforms today [2, 4]. To capture the benefit of incorporating self-suspensions into suspension-based analyses, we assume that tasks suspend for their entire critical sections as soon as they acquire the corresponding locks. As seen in this figure, the request-driven, the job-driven and the hybrid approaches significantly outperform the original busy-waiting approach by as much as 55.7%, 59.4% and 76.2% respectively. In particular, the hybrid approach can result in substantially higher schedulability than even the request-driven or the job-driven approaches as it combines benefits from both. Our hybrid approach also outperforms both versions of the LP-based FMLP+. Although `FMLP-SS` incorporates self-suspensions within critical sections, it does not seem to effectively capture their benefit, as it only provides a marginal improvement over `FMLP-BW`. This suggests that the LP-based approaches could benefit by incorporating insights from our paper. For clearer presentation, we omit results from `FMLP-BW` in the subsequent figures.

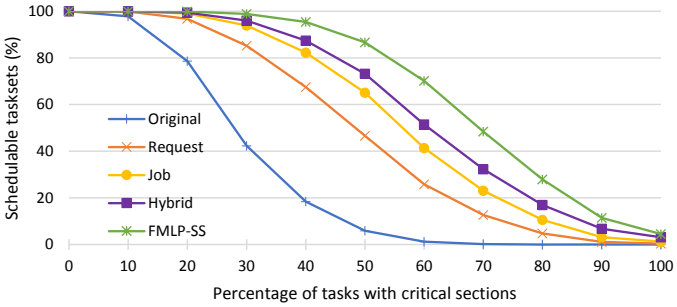


Fig. 3: Schedulability w.r.t. the percentage of tasks having critical sections. All tasks access a single resource. Tasks have a shorter period range of [300, 500]ms.

Our proposed analyses yield schedulability benefit due to self-suspensions within critical sections. To assess our analyses when such benefit is minimized, in Figure 2, we assume that all critical sections do not suspend. All other parameters remain the same as in Figure 1. All our analysis methods continue to outperform the original MPCP approach, and the hybrid approach outperforms the LP-based FMLP+. Such improvements to busy-waiting schedulability further indicate the effectiveness of the improvements proposed in this paper.

Prior work has shown MPCP to be less effective compared to the FIFO-based synchronization protocols when the ratio of the longest deadline and the shortest deadline among tasks is small [13]. While this intuitively makes sense, we postulate that our improvements could enable MPCP (and other priority-based protocols [23, 35]) to have better schedulability than before, under similar conditions. We test this hypothesis by evaluating tasks with shorter period ranges ($[300, 500]$ms), as shown in Figure 3. As expected, FMLP+ performs much better in this condition, and all the MPCP analyses show lower schedulability than in Figure 1. However, although the original and the request-driven analyses (on which prior MPCP analyses are based [29, 35]) decline quickly in performance, this is not the case with the job-driven and the hybrid approaches. This is because fewer resource requests can actually be made during the execution of each task when there is not a significant difference between task deadlines, and this is aptly captured by incorporating job-level characteristics.

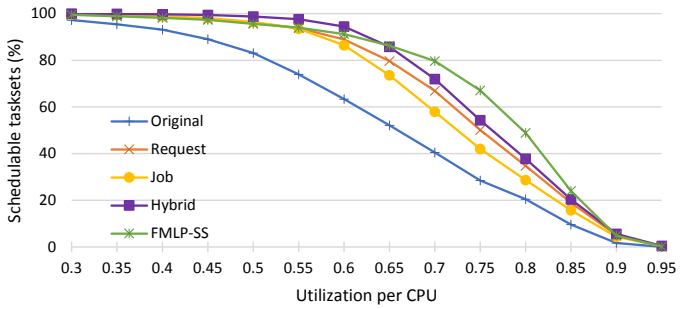


Fig. 4: Schedulability w.r.t. the taskset utilization per CPU.

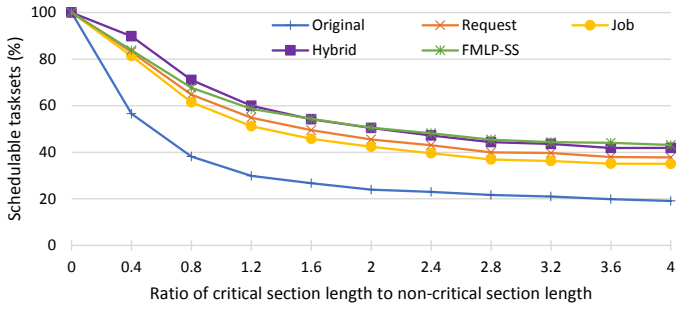


Fig. 5: Schedulability w.r.t. the critical section size (G_i/C_i).

Next, we examine schedulability with increasing utilization per CPU, shown in Figure 4. Here, although the original busy-waiting schedulability begins to quickly decline at a utilization of 35% per CPU, the hybrid analysis continues to show near-perfect schedulability up to a utilization of 55% per CPU. This indicates that suspension-based analyses can allow system resources to be much better utilized. Neither the hybrid nor the LP-based FMLP+ dominates the other in this experiment. FMLP+ performs better than the hybrid approach when the per-CPU utilization ranges from 70% to 85%, and ties with or underperforms the hybrid approach in other cases.

We vary the critical section length, the number of CPUs and the number of tasks per CPU in Figures 5, 6 and 7, respectively. Here, we notice that, while the original MPCP analysis continues to underperform, the request-driven analysis generally outperforms the job-driven analysis unlike in Figures 1, 2, 3, and 8. This indicates that the request-driven and the job-driven approaches can have orthogonal benefits based

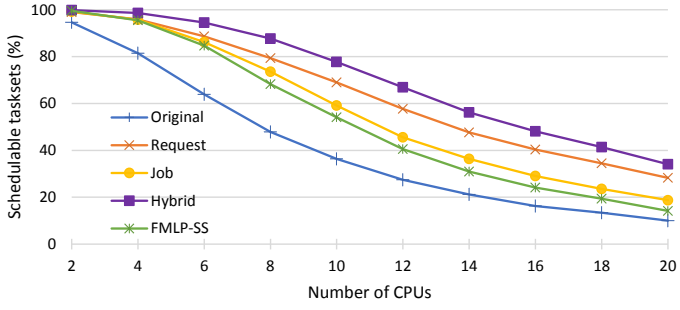


Fig. 6: Schedulability w.r.t. the number of CPUs.

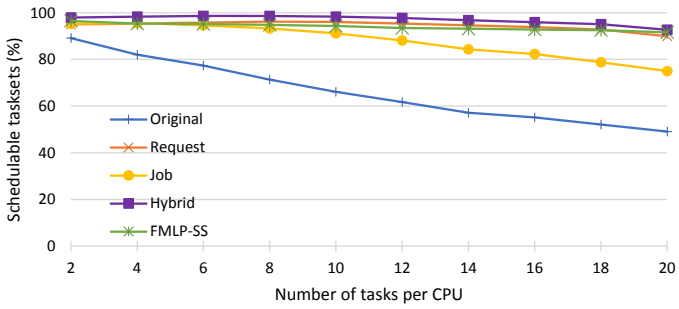


Fig. 7: Schedulability w.r.t. the number of tasks per CPU. Note that the utilization per CPU remains unchanged.

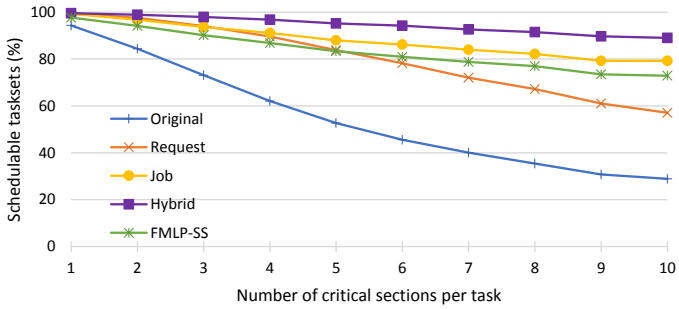


Fig. 8: Schedulability w.r.t. the number of critical sections per task.

on taskset parameters, but the hybrid approach, by design, consistently outperforms both. Figures 6 and 8 show that the hybrid approach performs better than `FMLP-SS` when the system has a high number of CPUs and tasks per CPU, respectively. Again, this supports the possibility that our approach captures suspension-based benefits effectively and can be quite promising for arbitrating access to resources that are typically contended for by many requesters, such as GPUs. We expect suspension-based MPCP to similarly be competitive in terms of schedulability with the recent server-based approach [26].

Figures 9 and 10 examine the effects of the length of suspensions and the number of suspensions within each critical section, respectively. To amplify their effects, a system with long critical sections (i.e., $G_i/C_i = [150, 200]\%$) is

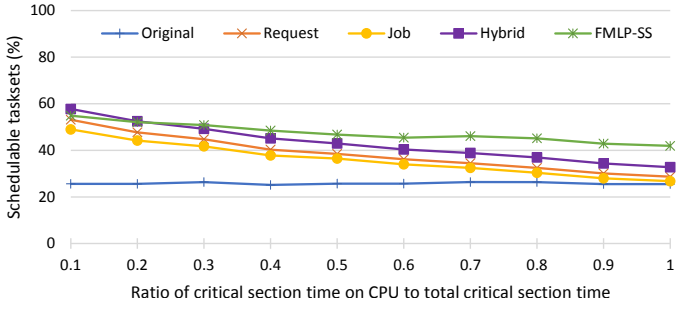


Fig. 9: Schedulability w.r.t. the suspension time in critical sections (G_m/G_i). Longer critical sections are evaluated by setting G_i/C_i = [150, 200]%.

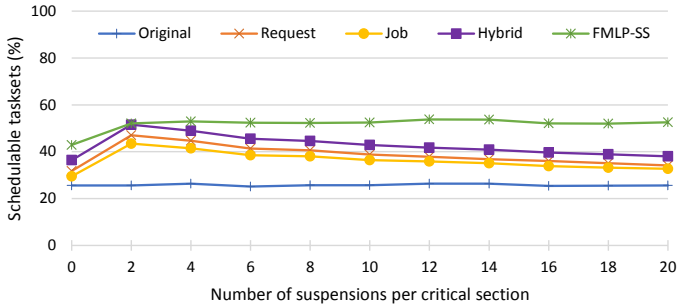
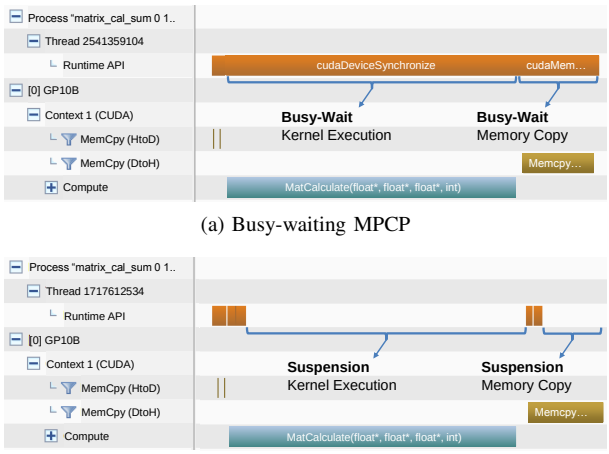


Fig. 10: Schedulability w.r.t. the number of suspensions per critical section. Longer critical sections are evaluated by setting G_i/C_i = [150, 200]%.



(a) Busy-waiting MPCP

(b) Suspension-based MPCP

Fig. 11: GPU execution timeline captured using `nvprof` [3].

assumed. We see no discernible effect on the original busy-waiting MPCP analysis as its critical sections continue to busy wait regardless of any change in those parameters. However, such changes are detrimental to the suspension-based MPCP analyses because they result in increasing preemption as the duration of suspension reduces or increasing indirect blocking delay as the number of suspensions per critical section increases, as explained in Section III-E. Figure 9 shows that the hybrid approach begins to underperform `FMLP-SS` when the ratio of CPU-execution time to the total critical section time is 0.3, indicating that it may better fit use cases with longer suspension durations. We expect that such cases are likely with computationally-intensive tasks that use shared accelerators.

Interestingly, in Figure 10, while the schedulability offered by our analyses steadily decreases, `FMLP-SS` is largely unaffected by the number of suspensions within critical sections. We believe that if our analytical enhancements are incorporated into such LP-based analyses [13, 14], the detrimental effects due to the number of critical section suspensions can be significantly reduced. This is because LP-based approaches can impose a tighter bound on indirect blocking by using job-driven constraints, which is difficult to express under traditional analysis. Furthermore, we expect such cases rarely arise in practice—for example, a GPU-using critical section typically suspends only twice or thrice as shown in Section IV-B: once to copy data, and once to execute commands on the GPU.

Summary. Our proposed analyses provide significant improvements in schedulability for several taskset configurations. All three of our analyses dominate the existing analysis for MPCP, even when critical sections are assumed to busy-wait. Furthermore, the hybrid approach is competitive with and often outperforms the LP-based FMLP+ when: *(i)* the percentage of tasks with critical sections is high, *(ii)* the ratio of the longest to the shortest task deadlines is large, *(iii)* the number of critical sections per task is high, or *(iv)* the critical sections spend less time busy-waiting. There are some cases where the LP-based FMLP+ analysis performs significantly better than our approach when: *(i)* the ratio of the longest to the shortest task deadlines is small, *(ii)* the critical sections are longer and spend more time busy-waiting, or *(iii)* the number of suspensions in critical sections is large.[7] However, such a benefit comes at a much higher computational cost of up to 100× to 200× longer analysis time compared to our hybrid approach on a server-class machine (with 32 GB RAM, running Intel Xeon E5-1620 at 3.50 GHz). This difference in computational overheads further increases with an increase in the number of analyzed tasks. We expect these performance issues to be prohibitively expensive for runtime admission control in adaptive and dynamic embedded real-time systems. Our recursion-based hybrid approach is highly practicable with its competitive performance and much more efficient runtime than LP-based analysis.

B. Case Study

Motivated by recent work that uses real-time synchronization for predictable GPU arbitration [20, 22, 26, 27], we examine the practical feasibility and efficacy of suspension-based MPCP and compare it with the busy-waiting approach in arbitrating access to a shared GPU on the NVIDIA TX2 embedded platform [2].

Hardware and Implementation. The TX2 is equipped with a quad-core 2.0-GHz ARMv8 A57 processor, a dual-core 2.0-GHz ARMv8 Denver processor, and an integrated Pascal GPU. We limit our experiments to two CPU cores of the A57 processor used in conjunction with the GPU. We run Ubuntu 16.04 with Linux kernel v4.4.15-tegra and use CUDA 8.0 for GPU programming [33]. We also configure each CPU to run at its maximum frequency of 2 GHz and disable all extraneous GPU-using tasks (such as lightDM) in order to avoid unnecessary GPU interference.

We use a `pthread_mutex` to implement semaphore-based MPCP. We suspend a task if the lock that it requests

[7] We note that the comparison search space has multiple features (such as the number of tasks, task parameters, number and type of critical sections, "short" vs "large" critical sections, etc.) and observe that further study may be required to clearly establish boundaries between configurations where one approach outperforms the other. However, we emphasize that neither approach dominates the other in general.

is already held by another task. To ensure priority-based task wakeup, suspended tasks are enqueued in priority queues and the highest-priority task is woken up when the held mutex is released. The mutex is managed via shared memory to make it accessible to all tasks. Once a task enters its critical section, the GPU execution is carried out by using one of two modes, as shown in Figure 11: *(i)* busy-waiting mode, which is the default CUDA setting that uses synchronous API calls and busy waits using `cudaDeviceSynchronize()`, and *(ii)* suspension-based mode, which is implemented using asynchronous CUDA functions. For the latter, GPU calls make use of streams and callbacks [1], and the GPU-using tasks use a POSIX conditional variable to suspend on the CPU. The callbacks are executed after all GPU operations enqueued in the stream have been completed. These callbacks are responsible for waking up the suspended CPU tasks. For both approaches, memory transfers between the GPU and the CPU are performed using pinned memory to minimize CPU involvement [19]. This results in extremely quick memory transfers between the GPU and the CPU. Hence, we configure our applications to suspend only during GPU execution and not during memory transfers.[8]

Tasks and Scheduling. Our tasks are motivated by the software system of the self-driving car developed at CMU [41]. Among various algorithmic tasks of the car, we chose two GPU-accelerated vision applications, namely the work-zone detector (`WZ`) [30] and the lane-change detector (`LC`), that periodically process images collected from a mono camera.[9] Arithmetic/matrix calculation tasks (`AM1`, ..., `AM5`) with varying parameters are also used to represent a subset of the other data-parallel tasks of the car. Unique task priorities are assigned based on the rate-monotonic policy [31]. Each task is pinned to a specific CPU, as described in Table III. Each task is also structured to run in either busy-wait mode or suspension mode during its critical section. Task releases are performed at absolute times using the `POSIX clock_nanosleep()` call and they are scheduled with real-time priority under the `SCHED_FIFO` policy. The response time of each task across each period is collected over a total duration of 30 seconds under each experimental setting.

Experiments. We conduct three experiments in our case study. In Expt. ①, we execute the taskset {`LC`, `WZ`, `AM1`, `AM2`, `AM3`} with busy-waiting and suspension-based MPCP and compare the observed WCRT of each task in Figure 12. In Expts. ② and ③, we execute `LC` and `WZ` with a varying number of instances of `AM4` and `AM5`, respectively. This is captured in Figures 13 and 14.

Observation 1. Suspension-based MPCP can offer significantly lower response-times than the busy-waiting approach in practice, especially for lower-priority tasks. This is because suspension-based MPCP allows other tasks to use the CPU during critical section execution. Figures 12 and 13 support this observation.

[8]Note that, for longer memory transfers (potentially without pinned memory), suspension is still possible by using asynchronous memory copy calls.

[9]We configure `LC` and `WZ` to use locally stored images in our experiments.

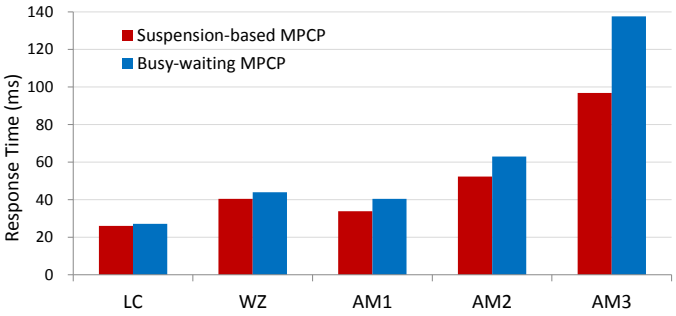


Fig. 12: WCRT comparison between suspension-based MPCP and busy-waiting MPCP.

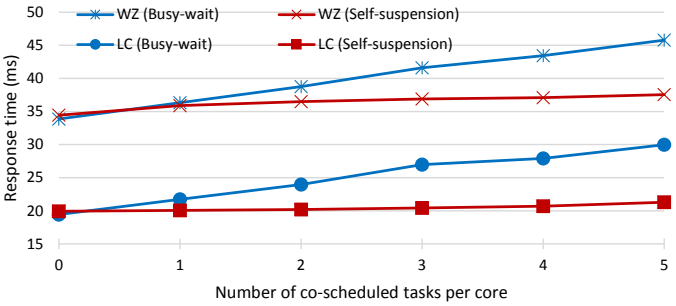


Fig. 13: WCRT w.r.t. the number of co-scheduled `AM4` tasks.

Observation 2. None of the deadlines are missed in Expt ①, either in the busy-waiting, or in the suspension-based cases. However, the original, the request-driven and the job-driven analyses all (pessimistically) indicate that the taskset is unschedulable. Only the hybrid blocking analysis confirms that the taskset is indeed schedulable. This shows the practical importance of using tighter schedulability analysis.

Observation 3. The overhead of GPU-related self-suspensions negatively affects the response times under the suspension-based approaches. This effect is particularly noticeable in Figure 14, where the co-scheduled `AM5` tasks have small GPU segments. Such overhead is incurred due to GPU callbacks, performing task suspensions and then waking up the suspended task. Such overheads are completely avoided in busy-waiting approaches. We measured this overhead and found that

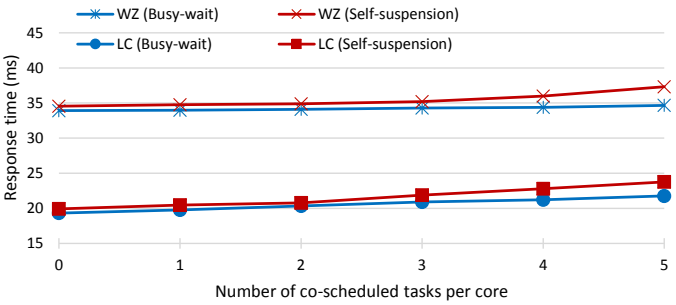


Fig. 14: WCRT w.r.t. the number of co-scheduled `AM5` tasks.

Task	C_i	G_i	η_i	$G^e_{i,1}$	$G^m_{i,1}$	$\zeta_{i,1}$	$G^e_{i,2}$	$G^m_{i,2}$	$\zeta_{i,2}$	$T_i = D_i$	CPU	Used In Experiment
LC	13.5	3.19	2	0	0.64	0	0.45	2.10	1	39.5	1	①,②,③
WZ	29.48	4.04	1	1.72	2.32	1	-	-	-	50	2	①,②,③
AM1	11.05	5.12	1	4.89	0.23	1	-	-	-	100	1	①
AM2	8.81	9.38	1	9.17	0.21	1	-	-	-	165	1	①
AM3	32.97	10.88	1	10.52	0.36	1	-	-	-	300	2	①
AM4	1.15	2.70	1	2.10	0.60	1	-	-	-	120	1 & 2	②
AM5	1.15	0.70	1	0.15	0.55	1	-	-	-	120	1 & 2	③

TABLE III: Tasks used in the case study. All times are in milliseconds (ms).

it is approximately $200\mu s$ per critical section on the NVIDIA TX2, which exceeds AM5's suspension duration. Furthermore, we observe that the disadvantage due to suspension-based overhead is easily overcome for tasks with longer GPU segments, such as with AM4 in Figure 13. Therefore, based on the critical section length and on the suspension-based overhead for a specific platform, it may be best to allow some critical sections to suspend and let the others (typically the smaller ones) busy wait.

V. Other Related Work

A significant body of related work on synchronization protocols and blocking analysis has already been covered in Sections I and III. We briefly discuss the other relevant approaches in this section.

In the past, synchronization protocols have been proposed and evaluated for partitioned, global and clustered scheduling, and their asymptotic optimality has been characterized and proven [12, 14, 16, 46]. We note that, although the MPCP analysis is not asymptotically optimal, it has been shown to offer competitive performance under fixed-priority partitioned scheduling [13, 37]. In addition, an alternate analytical method has also been recently proposed to improve MPCP schedulability, which takes into account the best-case execution time (BCET) of tasks to obtain tighter blocking bounds [45]. However, this approach requires estimation of BCETs and does not allow suspensions within critical sections. Prior research has also shown how fine-grained nested critical sections may be supported [39]. In future work, we plan to explore how a unified analysis can incorporate nested critical sections with BCET-aware suspension-based MPCP analysis.

With regard to multi-resource systems, various k-exclusion locking protocols have been proposed, which can arbitrate access to pools of identical resources (i.e., GPUs). Although most prior k-exclusions approaches make use of suspension-oblivious analyses [15, 21, 40], recent work has extended them to include semaphore-based analyses [43, 44], and to allow tasks to request multiple resource replicas [32]. The key advantage of such protocols is that they eliminate the need to statically assign real-time tasks to individual GPUs, and in a way, act like global schedulers. In contrast, we target the use of a single synchronization protocol to arbitrate access to various types of accelerators and shared resources partitioned among tasks, without necessitating a pool of similar resources. Moreover, we believe that incorporating suspensions within critical sections and taking into account job-level characteristics into schedulability analyses of k-exclusion protocols could provide further benefits.

Finally, self-suspension behavior of tasks has also been extensively studied in the context of real-time systems, which has led to corrections in existing semaphore-based synchronization analyses [6, 10, 18, 42]. Such studies are partly also motivated by the fact that I/O devices can suspend while being accessed. Our extensions to MPCP are based on their research findings.

VI. Concluding Remarks

In this paper, we presented new analyses to capture the effects of self-suspensions within critical sections when CPU tasks need to access shared hardware accelerators like GPUs and DSPs. In particular, we adapted two analytical methods, namely the request-driven and the job-driven approaches, to perform suspension-based blocking analysis. We additionally proposed a novel hybrid approach that derives insights from the prior approaches and offers better schedulability than both. Our analytical approaches significantly outperform the traditional busy-waiting analysis in terms of schedulability for a very wide range of practical test parameters. Our hybrid approach is very competitive with and often outperforms the LP-based FMLP+ analysis, while being faster by two orders of magnitude in performing schedulability analysis. Such lower runtime complexity is particularly appealing to practical real-time embedded systems where runtime admission control is required. We have also demonstrated that suspension-based MPCP can be implemented on a real platform with notable benefits compared to busy-waiting.

As a next step, it would be interesting to perform a detailed study of the suspension overhead on modern platforms with various accelerators. This might enable automatic decisions for choosing critical section behavior, i.e., to busy wait or to suspend. Suspension-based analysis also encourages exploration of new partitioning schemes geared towards efficient shared resource utilization. Finally, the original MPCP analysis has been compared extensively with several other synchronization protocols in the past (e.g., in [16, 23, 26]). Thus, equipped with enhanced analytical methods, such results should be revisited in the future.

Acknowledgement

We would like to thank General Motors R&D.

REFERENCES

[1] CUDA stream management and callbacks. http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[2] NVIDIA Jetson TX1/TX2 Embedded Platforms. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html.

[3] NVIDIA Visual Profiler User's Guide. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[4] NXP i.MX6 Processors. http://www.nxp.com.

[5] The Schedulability Test Collection and Toolkit (SchedCAT). http://github.com/brandenburg/schedcat.

[6] N. Audsley and K. Bletsas. Realistic analysis of limited parallel software/hardware implementations. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2004.

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[8] T. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[9] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[10] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, 2015.

[11] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[12] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, Chapel Hill, NC, USA, 2011.

[13] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[14] B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[15] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *ACM International Conference on Embedded Software (EMSOFT)*, 2011.

[16] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2013.

[17] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol – MrsP. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[18] J.-J. Chen et al. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund, 2016.

[19] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[20] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[21] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.

[22] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[23] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*, 2016.

[24] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*, 2011.

[25] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[26] H. Kim, P. Patel, S. Wang, and R. Rajkumar. A server-based approach for predictable GPU access control. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[27] H. Kim, P. Patel, S. Wang, and R. Rajkumar. A server-based approach for predictable GPU access with improved analysis. arXiv pre-print, https://arxiv.org/abs/1709.06613, 2017.

[28] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[29] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[30] J. Lee, Y.-W. Seo, W. Zhang, and D. Wettergreen. Kernel-based traffic sign tracking to improve highway workzone recognition for reliable autonomous driving. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2013.

[31] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[32] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson. Multiprocessor real-time locking protocols for replicated resources. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[33] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[34] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[35] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[36] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1988.

[37] J. Ras and A. Cheng. An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an SMP system. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

[38] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[39] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[40] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[41] J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar, and B. Litkouhi. Towards a viable autonomous driving research platform. In *IEEE Intelligent Vehicles Symposium (IV)*, 2013.

[42] M. Yang, J.-J. Chen, and W.-H. Huang. A misconception in blocking time analyses under multiprocessor synchronization protocols. *Real-Time Systems*, 53(2):187–195, 2017.

[43] M. Yang, H. Lei, Y. Liao, and Z.-W. Chen. Partitioned k-exclusion real-time locking protocol motivated by multicore multi-GPU systems. *Journal of Electronic Science and Technology*, 14(3):193–198, 2016.

[44] M. Yang, H. Lei, Y. Liao, and F. Rabee. PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi-GPU sharing under partitioned scheduling. In *IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2013.

[45] M. Yang, H. Lei, Y. Liao, and F. Rabee. Improved blocking time analysis and evaluation for the multiprocessor priority ceiling protocol. *Journal of Computer Science and Technology*, 29(6):1003–1013, 2014.

[46] M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *IEEE Real-Time Systems Symposium (RTSS)*, 2015.