Multiprocessor Real-Time Locking Protocols: from Homogeneous to Heterogeneous *

Kecheng Yang
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

In this project, we focus on real-time multiprocessor locking protocols. We first survey three relatively recently proposed such protocols, namely, FMLP, OMLP, and RNLP. FMLP is the first such protocol that can be applied to global EDF. OMLP is the first such protocol that is asymptotically optimal under s-oblivious analysis. Optimality results, which are proposed along with OMLP, for pi-blocking under both s-oblivious and s-aware analyses are also covered. RNLP is the first such protocol that supports fine-grained nested resource requests. We also explore the real-time synchronization problems arising in the heterogeneous case. Several interesting pi-blocking issues when the underlying multiprocessor platform is extended from an identical one to a uniform one are discussed.

1 Introduction

Since the advent of multi-core chips, the focus of real-time systems research community has shifted from uniprocessors to multiprocessors. In order to efficiently utilize the underlying multi-core platforms, researchers have been developing multiprocessors scheduling algorithms and shared resource allocation protocols. Such research on both sides has been very productive. In this project, we focus on the latter—real-time shared resource allocation protocols. In particular, we emphasize multiprocessor real-time locking protocols.

In many real-time scheduling analyses, it is usually assumed that the real-time tasks in the system are independent, but that is, in fact, usually not true. Shared resources among tasks are a typical source of task dependencies. In particular, shared resources may be protected by locks; each task requests the lock, holds the lock to proceed its execution, and finally release the lock when the resource access is finished. Therefore, when multiple tasks share a resource, *priority inversions*, *i.e.*, the situation where a lower priority job is executing while a higher priority one is waiting due to its resource lock request not satisfying, could happen. In such situations, a higher priority task could be blocked by a lower priority one, and we call such blocking as *priority inversion blocking*, or *pi-blocking*. Furthermore, when multiple tasks share multiple resources, a deadlock may occur.

In real-time systems, deadlocks should be prevented and the duration of pi-blocking must be upper bounded and accounted for in analysis. In uniprocessor systems, pi-blocking is well defined. Furthermore, both the priority ceiling protocol (PCP) [6] and the stack-based resource allocation protocol (SRP) [1] prevent deadlocks and ensure the duration of pi-blocking for each job is at most the length of of one outermost critical section, which is clearly asymptotically optimal.

However, when it turns to multiprocessors, the analysis and even the definition of pi-blocking become more complicated. On multiprocessors, it is possible that *some* of the processors are idle while a job is waiting, which should also be considered as pi-blocking. That is, pi-blocking may happen even when no lower priority job is scheduled.

*A course project report for Prof. James Anderson's class, COMP 735 Distributed and Concurrent Algorithms.

Multiprocessor locking protocols. Rajkumar *et al.* [12, 13, 14] were the first to propose real-time locking protocols for multiprocessors. They presented two multiprocessors PCP variants for partitioned static-priority schedulers, the multiprocessor priority ceiling protocol (MPCP)[12] and the distributed priority ceiling protocol (DPCP)[14]. The MPCP is more recently extended to support "virtual spinning" [10]. Subsequently, Chen *et al.* [5] proposed two protocols that apply only to periodic, but not sporadic, tasks, Lopez *et al.* [11] proposed a heuristic that partitions resources as tasks are also scheduled by partitioned EDF, and Gai *et al.* [9] proposed a spin-based SRP multiprocessor extension. Also, Easwaran *et al.* [7] proposed two suspension-based protocols for global static-priority schedulers.

Despite these cited prior multiprocessor locking protocols, in this project, we mainly survey the following three protocols, FMLP, OMLP and RNLP. The flexible multiprocessor locking protocol (FMLP) was proposed by Block *et al.* [2]. FMLP can be applied not only to partitioned EDF but also to global EDF, and it allows shorter critical sections to be implemented by busy-waiting (spin-based lock) while longer critical sections to be implemented by suspending blocked tasks. The $\mathcal{O}(m)$ locking protocol (OMLP)[3] and OMLP family[4] were proposed by Brandenburg *et al.* In their work, two kinds of pi-blocking analysis are concluded: suspension-oblivious (s-oblivious) and suspension-aware (s-aware). A lower bound of $\Omega(m)$ of pi-blocking per job is established for s-oblivious analysis while OMLP achieves a upper bound of $\mathcal{O}(m)$ pi-blocking per job under s-oblivious analysis and therefore asymptotically optimal; a lower bound of $\Omega(n)$ of pi-blocking per job is established for s-aware analysis while a variant of FMLP achieves a upper bound of $\mathcal{O}(n)$ pi-blocking per job under s-aware analysis and therefore asymptotically optimal. The real-time nested locking protocol (RNLP) was proposed by Ward *et al.* [15]. RNLP is the first multiprocessor real-time locking protocol that supports fine-grained nested resource requests. Subsequently, RNLP was further improved in terms of reducing blocking time by integrating spinning and replicas [16], and enabling concurrency by a reader/writer variant[17].

Heterogeneous multiprocessors. Although several multiprocessor locking protocols exist as cited above, to the best of our knowledge, all of such analyses pertain to homogeneous multiprocessors where each processor is identical. However, recent years, heterogeneous platforms are emerging. In terms of heterogeneity, processors in a heterogeneous platform may differ from *functionality* and/or *speed*. The former may result in that some tasks can only be scheduled on some function-specific processors, and therefore is more likely to partition tasks according to functionality, so that analysis can be done within each function group. Thus, we focus on the heterogeneous multiprocessors where every processor shares the same instruction set architecture but may differ from clock frequency, *i.e.*, processors have the same functionality but may have different speed. We call such platform *uniform* heterogeneous multiprocessors.

Project Statement. In this project, we focus on real-time multiprocessor locking protocols. Our first step is to survey three relatively recently proposed multiprocessors real-time locking protocol, FMLP, OMLP and RNLP. All of the three protocols are proposed for identical multiprocessors. Then, we look into the heterogeneous case, and explore, in those results we surveyed, what are preserved and what may change. We propose several initial real-time synchronization problems arising in the heterogeneous case. However, more detailed discussion and formal solutions for those problems remain as future work.

2 Background

We consider a system with m processors and n implicit sporadic tasks, *i.e.*, the task set $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. A *job* is an invocation of a task. $J_{i,j}$ denotes the jth invocation of task τ_i. We omit j when it is insignificant. For each task τ_i, T_i is its minimum separation between job releases, or its *period*. C_i is its *worst-case execution requirement*. The deadlines are implicit, *i.e.*, each job J_i has to complete at most T_i time units after its release. A job is *pending* since its release until it completes.

Resource model. We consider a system with q shared resources (other than processors), which are denoted as $l_1, l_2, \cdots, l_q$. When a job J_i requires a resource l_k, it issues a *request* $\mathcal{R}_{i,k}$, which is *satisfied* as soon as J_i holds l_k and completes when J_i releases l_k. Once a request $\mathcal{R}_{i,k}$ is issued, the job J_i is *spinning* or *suspended* until the

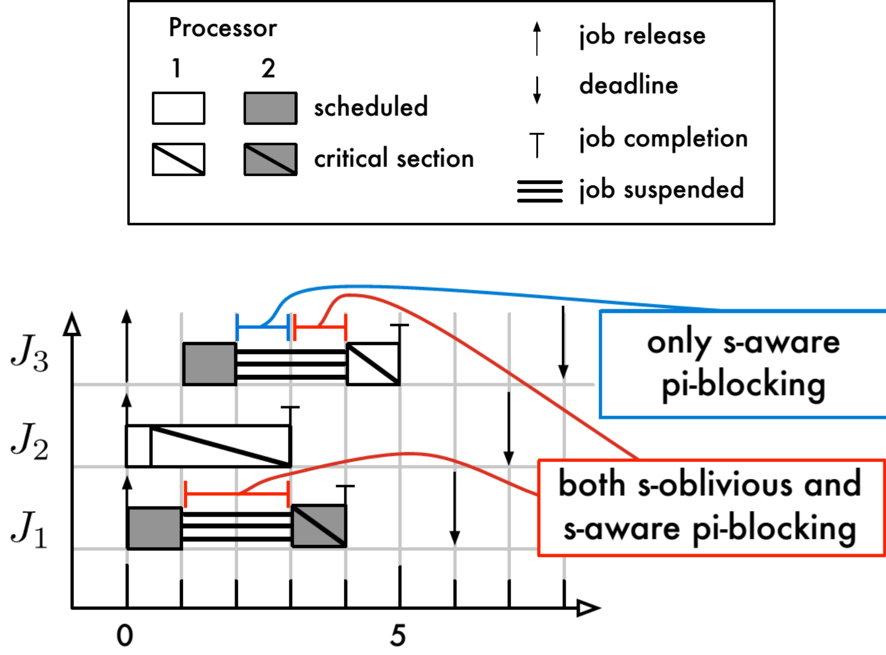

Figure 1: Illustration for s-oblivious and s-aware pi-blocking, cited from [4].

request is satisfied. A pending job that is not spinning or suspended is *ready*. We also let $N_{i,k}$ denote the maximum number of times that any J_i requests l_k, and let $L_{i,k}$ denote the maximum length of such a request.

Scheduling. We consider *clustered* scheduling, where global scheduling and partitioned scheduling can be considered as special cases of clustered scheduling as the number of clusters is 1 and m, respectively. Within each cluster, a work-conserving job-level fixed-priority (JLFP) scheduler is applied. Under JLFP scheduling, each job's *base priority* is fixed, although its *effect priority* may dynamically change to exceed its base priority, depending on the locking protocol.

The definition of pi-blocking on multiprocessors. The following definitions are for global scheduling. For clustered scheduling, it can be defined analogously with respect to each cluster [3, 4].

> **Def. 1.** Under *s-oblivious* schedulability analysis, a job J_i experiences pi-blocking if J_i is pending but not scheduled and fewer than m higher-priority jobs are *pending*.
>
> **Def. 2.** Under *s-aware* schedulability analysis, a job J_i experiences pi-blocking if J_i is pending but not scheduled and fewer than m higher-priority jobs are *ready*.
>
> Fig. 1 is an illustration for s-oblivious and s-aware pi-blocking.

Heterogeneous multiprocessors. In terms of heterogeneity, there are three categories of multiprocessors according to assumptions about processor speeds [8].

- **Identical multiprocessors.** Every job is executed on any processor at the same speed, which is usually normalized to be 1.0 for simplicity.

- **Uniform multiprocessors.** Different processors may have different speeds, but on a given processor, every job is executed at the same speed. The speed of processor p is denoted s_p.

- **Unrelated multiprocessors.** The execution speed of a job depends on both the processor on which it is executed and the task to which it belongs, *i.e.*, a given processor may execute jobs of different tasks at different speeds. The execution speed of task τ_i on processor p is denoted $s_{p,i}$.

Note that identical multiprocessors are a special case of uniform multiprocessors, which are a special case of unrelated multiprocessors. In this project, we only consider uniform multiprocessors.

3 FMLP

In this section, we survey the *flexible multiprocessor locking protocol* (FMLP) [2]. The "flexible" in its name is because of its applicability under both partitioned and global scheduling. Generally, FMLP has three key properties. First, it treats *short* and *long* resource requests differently. The former is handled by *busy-waiting* while the latter is handled by *suspension*. Second, the busy-waiting time is minimized by non-preemptively executing. Third, nested resources are *grouped*.

The locking protocol FMLP. Under FMLP, resources are grouped, where each group has only either short or long resources. Also, each group is protected by a lock, which is either a non-preemptive queue lock (for short resource group) or a semaphore (for long resource group). Also, FMLP group and only group resources that can be nestedly requested and are both either long or short. Non-nested resources are grouped individually. That improves parallelism.

When a outermost request for a short resource is issued, the corresponding group lock of the resource must be acquired. The blocked processes are busy-waiting for the queue lock in FIFO order. When attempting to acquire the queue lock, a job becomes non-preemptive until relinquishing the lock. Any request for other resources contained in this outermost request is satisfied immediately, and the lock is only relinquished when the outermost request completes.

When an outermost request for a long resource is issued, the corresponding group lock of the resource must be acquired as well. However, since the lock is a semaphore for long resources, blocked jobs do not busy-wait but do suspend. Suspended jobs are added to a queue in FIFO order. When a job holds the group lock, it inherits the highest priority of those jobs that blocked on a resource that is in this group. Then, the job is scheduled preemptively. Any request for other long resources contained in this outermost request is satisfied immediately, whereas any request for other short resources contained in this outermost request is either a outermost short resource request or is contained in such a request and therefore can be handled by the prior paragraph for short resources.

Furthermore, Thm.3 in [2] shows that FMLP is deadlock free.

The blocking time analysis. The term *blocking* refers to *pi-blocking*. Under FMLP, there are three kinds of blocking as follows.

- *Busy-wait blocking* occurs when a job must busy-wait in order to acquire a short resource. The maximum total amount of time for which any job of a task τ_i can busy-wait is denoted as $\mathsf{BW}(\tau_i)$.

- *Non-preemptive blocking* occurs when one of the m highest-priority jobs is pending but not scheduled due to a lower-priority job is executing in its non-preemptive section. The maximum total amount of non-preemptive blocking any job of a task τ_i can incur is denoted as $\mathsf{NPB}(\tau_i)$.

- *Direct blocking* occurs when one of the m highest-priority jobs is suspended because the group lock is held by some other job. The maximum total length of time any job of task τ_i can be directly blocked is denoted as $\mathsf{DB}(\tau_i)$.

The blocking time for any job of the task τ_i is $B(\tau_i)$, which consists of *busy-wait blocking* $\mathsf{BW}(\tau_i)$, non-preemptive blocking $\mathsf{NPB}(\tau_i)$, and direct blocking $\mathsf{DB}(\tau_i)$. That is,

$$\mathsf{B}(\tau_i) = \mathsf{BW}(\tau_i) + \mathsf{NPB}(\tau_i) + \mathsf{DB}(\tau_i).$$

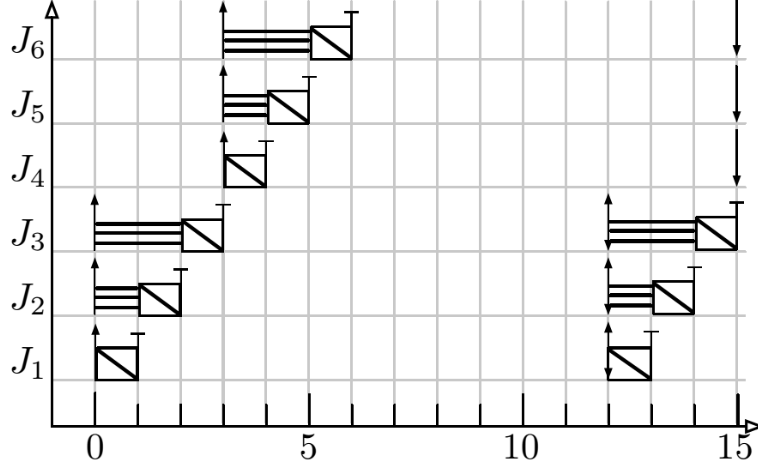

Figure 2: Example for establishing a lower bound of $\Omega(m)$ of pi-blocking per job under s-oblivious analysis.

Each of $\mathsf{BW}(\tau_i)$, $\mathsf{NPB}(\tau_i)$, and $\mathsf{DB}(\tau_i)$ can be upper bounded respectively. See the appendix of longer version of [2] for details (http://cs.unc.edu/ anderson/papers/rtcsa07along.pdf).

FMLP extensions. The discussion above is mainly for global EDF scheduled systems. [2] also showed the FMLP can be extended to systems scheduled by partitioned EDF or PD^2. However, under such variants, some issues have arisen. For example, the difference between global and local resources needs to be handled.

4 OMLP

In this section, we survey *the* $\mathcal{O}(m)$ *locking protocol* (OMLP) [3, 4] as well as the optimality results for multiprocessor real-time locking.

Lower bounds. In [3], Brandenburg and Anderson established a lower bound of $\Omega(m)$ of pi-blocking per job under s-oblivious analysis and a lower bound of $\Omega(n)$ of pi-blocking per job under s-aware analysis by the following lemmas, which consider the task set as follows.

Def. 3. Let $\tau^{seq}(n)$ denote a task set of n identical tasks that share one resource l_1 such that $C_i = 1$, $T_i = 2n$, $N_{i,1} = 1$, and $L_{i,1} = 1$ for each τ_i, where $n \geq m \geq 2$.

Lemma 1 (Lemma 1 in [3])**.** *There exists an arrival sequence for* $\tau^{seq}(n)$ *such that, under s-oblivious analysis,* $\max_{\tau_i \in \tau}\{b_i\} = \Omega(m)$ *and* $\sum_{i=1}^{n} b_i = \Omega(nm)$ *under any locking protocol and JLSP scheduler.*

Fig. 2 illustrates Lem. 1. See [3] for a detailed mathematical proof.

Lemma 2 (Lemma 10 in [3])**.** *There exists an arrival sequence for* $\tau^{seq}(n)$ *such that, under s-oblivious analysis,* $\max_{\tau_i \in \tau}\{b_i\} = \Omega(n)$ *and* $\sum_{i=1}^{n} b_i = \Omega(n^2)$ *under any locking protocol and JLSP scheduler.*

Fig. 3 illustrates Lem. 2. See [3] for a detailed mathematical proof.

The locking protocol OMLP. The general idea of OMLP is to apply two queues for each resource, where the first queue is a priority queue PQ and the second one is a FIFO queue FQ. A simplified algorithm description of OMLP for *global* JLFP schedulers is as follows.

- If a job request a resource, then add it to PQ of this resource;
- Jobs in PQ compete for a m-exclusion lock;
- The m highest-priority jobs in PQ hold the m-exclusion lock and hence are added to FQ, which is a FIFO queue;

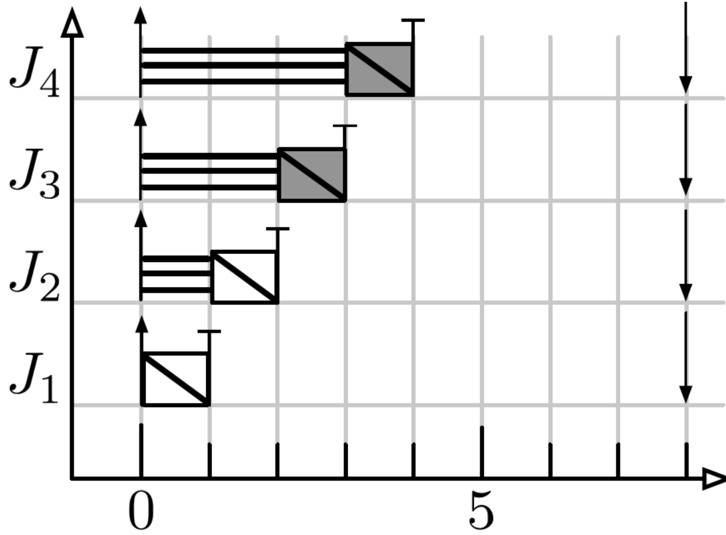

Figure 3: Example for establishing a lower bound of $\Omega(n)$ of pi-blocking per job under s-aware analysis.

- The job that is the head of FQ hold the FIFO lock, and therefore has its resource request to be satisfied.
- The job executes its critical section;
- The job release the FIFO lock;
- The job release the m-exclusion lock.

Furthermore, the FIFO lock holder inherits the highest priority of any job that is blocked on this resource.

See [3] for a formal and detailed description of the protocol as well as the other OMLP version for partitioned schedulers. [3] also analyzed the asymptotic blocking time, which is $\mathcal{O}(m)$ under OMLP.

Theorem 1 (Theorem 1 and 3 in [3])**.** *S-oblivious pi-blocking under OMLP is* $\mathcal{O}(m)$.

Intuitively, the Thm. 1 holds as follows.

- A job can be blocked by $\mathcal{O}(m)$ *lower-priority* jobs (which have already held the m-exclusion lock and are in the FIFO queue) in acquiring the m-exclusion lock.
- A job can be blocked by $\mathcal{O}(m)$ jobs (higher- or lower-priority) in acquiring the FIFO lock.

Thus, it is $\mathcal{O}(m)$ pi-blocking in total for each job.

Optimality results. By Lem. 1 and Thm. 1, pi-blocking under OMLP is $\mathcal{O}(m)$ under s-oblivious analysis, which is asymptotically optimal. On the other hand, Lem. 2 established a lower bound of $\Omega(n)$ of pi-blocking per job under s-aware analysis. In the global case, the suspension-based FMLP for "long" resources ensures $\mathcal{O}(n)$ pi-blocking under s-aware analysis, therefore it is asymptotically optimal. In the partitioned case, [3] proposed a simple variant of FMLP, called SPFP, which ensures $\mathcal{O}(n)$ pi-blocking under s-aware analysis and therefore is asymptotically optimal. In summary, Table. 1 is a comparison of real-time multiprocessor locking protocols, and shows the optimality results.

Scheduler	Protocol	Queue	Maximum pi-blocking	
			s-oblivious	s-aware
global	FMLP [2]	FIFO	$\Theta(n)$	$\Theta(n)$
	PIP [7]	priority	$\Omega(n)$	$\Omega(mn)$
	PCP [7]	priority	$\Omega(n)$	$\Omega(mn)$
	OMLP	hybrid	$\Theta(m)$	$\Omega(mn)$
partitioned	DPCP [14]	priority	$\Omega(n)$	$\Omega(mn)$
	MPCP [12]	priority	$\Omega(n)$	$\Omega(mn)$
	FMLP [2]	FIFO	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
	OMLP	hybrid	$\Theta(m)$	$\Omega(mn)$
	SPFP	FIFO	$\Theta(n)$	$\Theta(n)$

Table 1: A summary of asyptotically pi-blocking under real-time multirprocessor locking protocols.

5 RNLP

In this section, we survey the *real-time nested locking protocol* (RNLP) [15] and its extensions [16, 17]. The RNLP is the first multiprocessor real-time locking protocol that supports fine-grained nested resource requests. Unlike prior work that uses group locks, resources are not statically grouped under the RNLP, which only requires a partial order on resource acquisition. The notation for the partial order is $\prec$, *i.e.*, a job holding resource l_b cannot request resource l_a if $l_a \prec l_b$.

The RNLP has a k-exclusion token lock, which restricts the number of jobs that have incomplete resource request, and a request satisfaction mechanism (RSM), which determines when requests are satisfied. Each job that issues a resource request must first acquire a token, and then can compete for the resource via the RSM. According to different system and analysis configurations and assumptions, the RNLP can apply different RSM to yield an asymptotically optimal locking protocol that supports nested requests.

For each resource l_a, let RQ_a denote its resource queue of length at most k. Let $ts(J_i)$ denote the timestamp of job J_i, and let $hd(a)$ denote the head of RQ_a. There are a set of rules that are common for all RSMs as follows [15].

Q1 When J_i acquires a token at time t, its timestamp is recorded: $ts(J_i) := t$. We assume a total order on such timestamps.

Q2 All jobs in RQ_a are waiting with the possible exception of $hd(a)$.

Q3 A job J_i acquires resource l_b when it is the head of the RQ_b, *i.e.*, $J_i = hd(b)$, and there is no resource l_a such that $l_a \prec l_b$ and $ts(hd(a)) < ts(J_i)$.

Q4 When a job J_i issues a request for resource l_a it is enqueued in RQ_a in increasing timestamp order.

Q5 When a job releases resource l_a it is dequeued from RQ_a and the new head of RQ_a can gain access to l_a, subject to Rule Q3.

Q6 When J_i completes its outermost critical section, it releases its token.

Nonetheless, the common rules above do not specify how waiting is realized. A specific RSM may employ either spinning or suspending.

In [15], four specific RSMs are provided—the spin RSM (S-RSM), boost RSM (B-RSM), inheritance RSM (I-RSM), and donation RSM (D-RSM).

S-RSM. The S-RSM uses spinning to realize waiting. Spinning outperforms suspension when the lengths of critical sections are short. Under S-RSM, the following rule is added.

S1 All token-holding jobs execute non-preemptively. A job that is waiting in a resource queue spins.

The S-RSM applies to partitioned, clustered, or globally scheduled systems.

B-RSM. The B-RSM uses suspension to realize waiting, and uses priority boosting to ensure progress. The B-RSM requires the following rule.

B1 The (at most) m jobs with the earliest timestamps among the resource-holding jobs without outstanding resource requests (*i.e.*, that are not waiting) are boosted above the priority of all non-resource-requesting jobs.

The B-RSM applies to partitioned, clustered, or globally scheduled systems as well. Also, B-RSM ensures progress in any JLFP scheduled system; however, it does not always lead to an asymptotically optimal locking protocol.

I-RSM. The I-RSM uses suspension to realize waiting as well, but uses priority inheritance instead of priority boosting to ensure progress. Let $p(J_i, t)$ denote the effect priority of job J_i at time t. The following is the rule for I-RSM.

I1 A ready job J_i holding resource l_k inherits the highest priority of the jobs for which it is an inheritance candidate:

$$p(J_i, t) = \max_{J_k \in \{J_i\} \cup ICS(J_i, t)} p(J_k, t),$$

where $ICS(J_i, t)$ denotes the *inheritance candidate set* (see [15] for formal definition) of job J_i at time t. Unlike the prior two RSMs, I-RSM is only applicable to global schedulers because priorities of all jobs that request resources need to be compared. However, under I-RSM, the number of tokens in the system (*i.e.*, k) is tunable, which can yield better parallelism and better worst-case pi-blocking.

D-RSM. The I-RSM uses suspension to realize waiting as well while ensures progress via priority donation, which is designed for cluster scheduled systems (hence applicable to global or partitioned scheduled systems). Let c denote the number of processors in a cluster. D-RSM has no additional RSM rule, but requires the following constraint on the token lock.

C1 A token-holding job has one of the highest c effective priorities in its cluster.

Note that the D-RSM itself does not cause pi-blocking for non-resource-requesting jobs; however, the Property C1, which the D-RSM requires, does.

Extensions to RNLP. After being proposed, the RNLP was subsequently extended in several ways.

[16] improved the RNLP by considering two specific use cases, GPUs and cache lines, and then proposed and addressed three issues. First, when there is *a priori* knowledge for a task to request multiple resources, acquiring each resource individually in a nested fashion may incur unnecessary system-call overheads. This issue is addressed by proposing *dynamic group locks* (DGLs), which is a hybrid of coarse- and fine-grained locking. Second, the RNLP may cause *short-on-long* blocking, *i.e.*, short requests may be blocked by long requests. This issue is addressed by more greedily satisfying short requests, which can cause additional *long-on-short* blocking but it is usually an acceptable tradeoff at runtime. Third, the RNLP does not support replicated resources. This issue is addressed by altering the queue structure to allow multiple jobs to concurrently hold replicas of the same resource, and hence introducing the support for replicated resources to the original RNLP.

Furthermore, the RNLP treats all resources as mutex resources, *i.e.*, every resource can be accessed by only one task at a time. However, in practice, many applications access resources in a read-only fashion, which should have a higher degree of parallelism. In such circumstance, the RNLP significantly limits concurrency. To address this problem,a reader/writer variant of the RNLP (the R/W RNLP for short) is presented [17]. The R/W RNLP

improves the concurrency by allowing read-only resource requests to be satisfied simultaneously. Adopted from *phase-fair reader/writer locks*, the concept of reader and writer phases is applied in the R/W RNLP. Under the R/W RNLP, the worst-case acquisition delay is $\mathcal{O}(1)$ for readers and $\mathcal{O}(m)$ for writers.

6 Heterogeneous Platforms

As in prior sections, we surveyed three real-time multiprocessor locking protocols. However, all of them are with respect to identical multiprocessors. In this section, we would like to discuss several issues that are novel when the underlying multiprocessor platform is extended from an identical one to a uniform one.

Optimality results. In the optimality results from [3], the blocking time of each individual critical section is deemed as constant, and hence derive the asymptotically optimal pi-blocking. On uniform platform, the only change is that the processors may have different speed, but this has no effect on those optimality results, assuming the blocking time of each individual critical section is still deemed as constant. More specifically, the blocking time of each individual critical section can be pessimistically provisioned by the worst-case execution time for this critical section on the *slowest* processor. That can be still deemed as a constant. Thus, the asymptotically optimality results are preserved on uniform platforms.

Next, we will discuss some issues with respect to the blocking time of each individual critical sections on uniform platform.

Non-scalable critical section. In fact, not all task executions, especially critical sections, are scalable by processor speed. Many critical sections are related to I/O accesses, which may need the I/O devices to make some operations that is totally not dependent on CPU processor speed. In the identical processors case, this situation can be provisioned by analyzing the maximum time needed to finish this I/O accesses, and then inflate the worst-case execution time of this task by that amount. However, in uniform platforms, to analogously inflate the worst-case execution requirement of this task, the speeds of processors have to be involved. Under global scheduling, we need to always pessimistically consider such critical sections are executing on the fastest processor, and therefore inflate the worst-case execution requirement by the access time multiplying the fastest speed. Thus, in this case, we probably want a more intelligent algorithm that executes those critical sections on slower processors. Nonetheless, in this case, the worst-case blocking time is constant regardless processor speeds. This may help the analysis.

Scalable critical section. For other critical sections that are CPU executions and are therefore scalable by processor speeds, the situation is different. In this case, a certain critical section of a certain task could have multiple blocking times to block others, depending on where this blocking task is scheduled. In this case, the execution requirement of the critical sections can be directly accounted in the worst-case execution requirement of the task. However, under global scheduling, to derive a sufficiently safe analysis, significant pessimism may have to be introduced to the pi-blocking analysis. Specifically, it may be necessary to assume that when a task is blocking others, it executes on the slowest processor, and hence yields a maximum blocking time; when a task is blocked, it is blocked on the fastest processors, and hence in s-oblivious analysis, its execution requirement needs to be inflated by the suspension time multiplying the fastest processor speed.

Partitioned scheduling and clustered-by-speed scheduling. As mentioned in the above two paragraphs, under global scheduling, significant analytical pessimism is necessary. Therefore, we would like to have a look at other alternative scheduling approaches. If the system is scheduled by a partitioned scheduler, then the pi-blocking time and impact of each critical section are much more predictable, because the processor it will execute on is deterministic. Alternatively, we can also consider clustered scheduling, where each cluster only has processors of the same speed. In this case, pi-locking time and impact of the critical sections are also as predictable as partitioned scheduling. However, when partitioned or cluster scheduling is applied, it implies the introduction of the bin-packing problem, which causes system utilization loss.

Dedicated processor to handle all critical sections. Another idea is that the most parts of tasks are still scheduled globally, but all the critical sections are scheduled on a dedicated processor that does not execute other non-critical globally-scheduled parts. Applying this approach, we can take advantage of higher system utilization of global

scheduling while also have the critical sections more predictable. However, there are also several issues in this approach. First, the dedicated processor may be overutilized barely by critical sections. Then, we need multiple such dedicated processors to handle critical sections, where each processor handled the critical sections for certain resources. Moreover, when a single processor handles critical sections of several resources, then the accesses of those resources may be unnecessarily serialized, which reduces parallelism and increases blocking times. Finally, dedicating such processors for critical sections may fundamentally incur system utilization loss. That is, similar to the prior paragraph, it is going to be a tradeoff between platform utilization loss and per job blocking time analysis.

7 Conclusion

In this project, we started with real-time multiprocessor synchronizations, specifically with real-time multiprocessor locking protocols. We first surveyed three relatively recently proposed such protocols, namely, FMLP, OMLP, and RNLP. FMLP is the first such protocol that can be applied to global EDF. FMLP is optimized for non-nested resources and able to support nested resource to some extend. OMLP is the first such protocol that is asymptotically optimal under s-oblivious analysis. In the paper where OMLP proposed, asymptotically lower bounds for both s-oblivious and s-aware analyses are provided, and besides OMLP, which is asymptotically optimal under s-oblivious analysis, a variant of FMLP is shown asymptotically optimal under s-aware analysis. RNLP is the first such protocol that supports fine-grained nested resource requests. RNLP is also further extended by improving blocking analysis and introducing the concept of reader/writer phases. We also discussed several interesting issues regarding to real-time multiprocessor locking protocols when the underlying multiprocessor platform is extended from an identical one to a uniform one, although more formal solutions for that remain as future work.

References

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

[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th RTCSA*, 2007.

[3] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *31st RTSS*, 2010.

[4] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2014.

[5] C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. *Technical Report CS-TR-3252, University of Maryland*, 1994.

[6] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2:325–346, 199.

[7] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *30th RTSS*, 2009.

[8] S. Funk. *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2004.

[9] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C.Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *9th RTAS*, 2003.

[10] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *30th RTSS*, 2009.

[11] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):3968, 2004.

[12] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th ICDCS*, 1990.

[13] R. Rajkumar. Synchronization in real-time systems a priority inheritance approach. *Kluwer Academic Publishers*, 1991.

[14] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *9th RTSS*, 1988.

[15] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *24th ECRTS*, 2012.

[16] B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *21st RTNS*, 2013.

[17] B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *28th IPDPS*, 2014.