The FMLP+: An Asymptotically Optimal Real-Time Locking Protocol for Suspension-Aware Analysis

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Abstract—Multiprocessor real-time locking protocols that are asymptotically optimal under suspension-oblivious schedulability analysis (where suspensions are pessimistically modeled as processor demand) are known for partitioned, global, and clustered job-level fixed priority (JLFP) scheduling. However, for the case of more accurate suspension-aware schedulability analysis (where suspensions are accounted for explicitly), asymptotically optimal protocols are known only for partitioned JLFP scheduling. In this paper, the gap is closed with the introduction of the first semaphore protocol for suspension-aware analysis that is asymptotically optimal under global and clustered JLFP scheduling. To this end, a new progress mechanism that avoids repeated priority inversions is developed and analyzed, based on the key observation that if lock-holding, low-priority jobs are priority-boosted, then certain other non-lock-holding, higher-priority jobs must be co-boosted.

I. INTRODUCTION

The purpose of suspension-based real-time locking protocols is to limit priority inversions [22], which, intuitively, occur when a high-priority task that should be scheduled is instead delayed by a remote or lower-priority task. Such locking-related delay, also called priority inversion blocking (pi-blocking), is problematic because it can result in deadline misses. However, some pi-blocking is unavoidable when using locks and thus must be bounded and accounted for during schedulability analysis.

Clearly, an “optimal” locking protocol should minimize pi-blocking to the extent possible. Formally, a locking protocol is asymptotically optimal if it ensures that, for any task set, maximum pi-blocking is bounded within a constant factor of the pi-blocking unavoidable in some task set [11]. Interestingly, there exist two classes of schedulability analysis that yield different lower bounds: under suspension-oblivious (s-oblivious) analysis, $\Omega(n)$ pi-blocking is fundamental, whereas under suspension-aware (s-aware) analysis, $\Omega(n)$ pi-blocking is unavoidable in the general case [7, 11], where $m$ and $n$ denote the number of processors and tasks, respectively. As the names imply, the key difference is that suspensions are accounted for explicitly under s-aware analysis, whereas they are (pessimistically) modeled as processor demand in the s-oblivious case. In principle, s-aware schedulability analysis is preferable, but s-oblivious analysis is easier to derive and permits simpler pi-blocking bounds.

And indeed, for the simpler s-oblivious case, asymptotically optimal locking protocols are known for partitioned, global, and clustered job-level fixed-priority$^3$ (JLFP) scheduling [9, 11, 12]. In contrast, the s-aware case is analytically much more challenging and less understood: asymptotically optimal protocols are known so far only for partitioned JLFP scheduling [7, 11]. The general problem of optimal s-aware locking under global and clustered JLFP scheduling, however, has remained unsolved.

A. Contributions

We answer this fundamental question by introducing the generalized FIFO Multiprocessor Locking Protocol (FMLP$^+$), the first semaphore protocol for clustered scheduling that ensures $O(n)$ maximum s-aware pi-blocking under any JLFP policy.

While it was initially assumed [11] that a variant of Block et al.’s Flexible Multiprocessor Locking Protocol (FMLP) [6]—which uses $O(n)$ FIFO queues together with priority inheritance [22]—is asymptotically optimal under global scheduling, we show in Sec. III that this holds only under some, but not all global JLFP schedulers. In fact, we show that both priority inheritance and (unrestricted) priority boosting [22], which are the two mechanisms used in all prior locking protocols for s-aware analysis to avoid unbounded pi-blocking, can give rise to non-optimal $\Omega(\Phi)$ pi-blocking, where $\Phi$ is the ratio of the longest and the shortest period (and not bounded by $m$ or $n$).

To overcome this lower bound, we introduce in Sec. IV-A a new progress mechanism called “restricted segment boosting,” which boosts at most one carefully chosen lock-holding job in each cluster while simultaneously “co-boosting” certain other, non-lock-holding jobs to interfere with the underlying JLFP schedule as little as possible. Together with simple FIFO queues, this ensures $O(n)$ maximum s-aware pi-blocking (within about a factor of two of the lower bound, see Sec. IV-C). Notably, our analysis permits non-uniform cluster sizes, allows each cluster to use a different JLFP policy, supports self-suspensions within critical sections (Sec. IV-F), and can be easily combined with prior work [25] to support nested critical sections (Sec. IV-G).

Finally, while answering the s-aware blocking optimality question in the general case is the main contribution of this paper, Sec. V presents a schedulability study that shows the FMLP$^+$ to outperform s-oblivious approaches if the underlying s-aware schedulability analysis is sufficiently accurate.

B. Related Work

On uniprocessors, the blocking optimality problem has long been solved: both the classic Stack Resource Policy [3] and the Priority Ceiling Protocol [22, 24] limit pi-blocking to at most one (outermost) critical section, which is obviously optimal.

On multiprocessors, there are two major lock types: spin locks, wherein blocked jobs busy-wait, and suspension-based semaphores. Spin locks are well understood and it is not difficult to see that non-preemptable FIFO spin locks, which ensure $O(n)$
blocking [6, 13, 15], are asymptotically optimal. Intuitively, spin locks are appropriate for short critical sections, whereas busy-waiting becomes problematic with longer critical sections (and especially if critical sections contain self-suspensions).

Numerous semaphore protocols have been proposed in recent years (e.g., [6, 14, 16, 17, 19, 20]; see [7, 8, 13] for recent overviews); due to space constraints, we focus here on the most relevant work pertaining to blocking optimality.

Blocking optimality in multiprocessor real-time systems was first considered in [11], which established lower bounds on s-aware and s-oblivious pi-blocking and introduced protocols (asymptotically) matching upper bounds, namely the O(m) Locking Protocol (OMLP) for partitioned and global scheduling under s-oblivious analysis, and a simple, but impractical proof-of-existence protocol for partitioned scheduling under s-aware analysis. It was also suggested in [11] that the global FMLP [6] is asymptotically optimal under global scheduling w.r.t. s-aware analysis, which, however, is only the case under certain JLFP schedulers, as we discuss in Sec. III. The OMLP was later extended to clustered scheduling [12], and the O(m) Independence-Preserving Locking Protocol (OMLP) [9] was introduced as an (also asymptotically optimal) alternative to the OMLP for systems with stringent latency requirements.

A precursor to this paper is the partitioned FIFO Multiprocessor Locking Protocol (P-FMLP+), which was introduced in [7] as a refinement of Block et al.'s earlier partitioned FMLP [6]. The original partitioned FMLP [6] is not asymptotically optimal under s-aware analysis due to a limiting tie-breaking rule, which the P-FMLP+ corrects to ensure asymptotically optimal s-aware pi-blocking [7]. However, the P-FMLP+ is based on priority boosting and hence achieves asymptotic optimality only under partitioned scheduling. In this paper, we generalize the P-FMLP+ to clustered scheduling, starting from first principles. To disambiguate the earlier variant from the generalized version developed in this paper, we refer to the earlier partitioned version [7] exclusively as the P-FMLP+ and reserve the name FMLP+ for the new, generalized protocol introduced in Sec. IV.

Most recently, Ward and Anderson [25, 26] presented the RNLP, which is the first multiprocessor real-time locking protocol to support fine-grained locking. The generalized FMLP+ presented in this paper can be integrated with the RNLP to support nested critical sections, which we discuss in Sec. IV-G.

Finally, to the best of our knowledge, no asymptotically optimal s-aware locking protocol for the general case of clustered JLFP scheduling has been proposed in prior work. We present our solution in Sec. IV after first introducing needed background in Sec. II and demonstrating the sub-optimality of existing progress mechanisms in Sec. III.

II. DEFINITIONS AND ASSUMPTIONS

We consider a real-time workload consisting of n sporadic tasks τ = \{T_1, ..., T_n\} scheduled on m identical processors. We denote a task T_i's worst-case execution cost as e_i, its minimum inter-arrival time (or period) as p_i, and its relative deadline as d_i. Task T_i has an implicit deadline if d_i = p_i, a constrained deadline if d_i ≤ p_i, and an arbitrary deadline otherwise. We let J_{i,j} denote the j^{th} job of T_i, and let J_i denote an arbitrary job of T_i. A task's utilization is defined as u_i = e_i/p_i. A job J_i is pending from its release until it completes. T_i's worst-case response time r_i denotes the maximum duration that any J_i remains pending. If a job J_{i,j+1} is released before its predecessor J_{i,j} has completed (i.e., due to a deadline miss or if d_i > p_i), then J_{i,j+1} is not eligible to execute until J_{i,j} has completed (i.e., tasks are sequential).

While pending, a job is either ready (and can be scheduled) or suspended (and not available for scheduling). Jobs suspend either when waiting for a contended lock (Sec. II-A), or may also self-suspend for other, locking-unrelated reasons. We model locking-unrelated self-suspensions explicitly because they affect the locking protocol presented in Sec. IV. To this end, we let \(u_{i,j}\) denote the maximum number of self-suspensions of any \(J_i\).

For simplicity, we assume integral time: all points in time and all task parameters are integer multiples of a smallest quantum (e.g., a processor cycle), and a point in time \(t\) represents the interval \([t, t+1)\).

A. Shared Resources

Besides the \(m\) processors, the tasks share \(n\) serially-reusable shared resources \(\ell_1, ..., \ell_n\) (e.g., I/O ports, network links, data structures, etc.). We let \(N_{i,q}\) denote the maximum number of times that any \(J_i\) accesses \(\ell_q\), and let \(L_{i,q}\) denote \(T_i\)'s maximum critical section length, that is, the maximum time that any \(J_i\) uses \(\ell_q\) as part of a single access (\(L_{i,q} = 0\) if \(N_{i,q} = 0\)). As a shorthand, we define \(N_i = \sum_{q=1}^{n} N_{i,q}\) and \(L_{i,q}^{\max} = \max\{L_{i,q} | T_i \in \tau \land 1 \leq q \leq n\}\). The worst-case execution cost \(e_i\) includes all critical sections.

A request for a shared resource is nested if the requesting job already holds a resource, and outermost otherwise. We assume that jobs release all resources before completion, and that they must be scheduled to issue requests for shared resources.

To ensure mutual exclusion, shared resources are protected by a locking protocol. If a job \(J_i\) needs a resource \(\ell_q\) that is already in use, \(J_i\) must wait and incurs acquisition delay until its request for \(\ell_q\) is satisfied (i.e., until \(J_i\) holds \(\ell_q\)'s lock). In this paper, we focus on semaphore protocols, wherein waiting jobs suspend.

B. Multiprocessor Real-Time Scheduling

We consider the general class of clustered job-level fixed-priority (JLFP) schedulers. Under a JLFP policy, the priorities of tasks may change over time, but each job is assigned a fixed, unique priority. The two most commonly used JLFP policies are earliest-deadline first (EDF) and fixed-priority (FP) scheduling. We let \(Y(J_i)\) denote the (fixed) priority of job \(J_i\), where \(Y(J_h) < Y(J_i)\) indicates that \(J_h\) has higher priority than \(J_i\). We assume that priorities are unique (e.g., by breaking any ties in favor of lower-indexed tasks) and transitive (i.e., if \(Y(J_h) < Y(J_i)\) and \(Y(J_i) < Y(J_j)\), then \(Y(J_h) < Y(J_j)\)).

Under clustered scheduling the \(m\) processors are organized into \(K\) disjoint subsets (or clusters) of processors \(C_1, ..., C_K\), where \(m_k\) denotes the number of processors in the \(k\)th cluster and \(\sum_{k=1}^{K} m_k = m\). Each task is statically assigned to one of the clusters; we let \(C(T_i)\) denote the cluster that \(T_i\) has
been assigned to, define \( n_k \triangleq |\{ T_i \mid C(T_i) = C_k \}| \), and let \( \text{ready}(C_k, t) \) denote the set of ready jobs in cluster \( C_k \) at time \( t \).

In each cluster, at any point in time the \( m_k \) highest-priority ready jobs are selected for scheduling (if that many exist) as determined by the employed JLFP policy, unless the regular prioritization is overruled by a locking protocol’s progress mechanism (see Sec. II-D). Jobs may migrate freely within each cluster, but not across cluster boundaries. Clusters may be of non-uniform size and may each employ a different JLFP policy.

Two prominent special cases are partitioned scheduling (where \( K = m \) and \( m_k = 1 \) for all \( C_k \)) and global scheduling (where \( K = 1 \) and \( m_1 = m \)). We consider clustered EDF (G-EDF) as a representative JLFP policy, and further discuss global EDF (G-EDF) and global FP (G-FP) scheduling in Sec. III, and partitioned FP (P-FP) scheduling in Sec. V. The main result of this paper (Sec. IV), however, applies to any JLFP policy.

C. Priority Inversions, PI-Blocking, and Asymptotic Bounds

Locking protocols give rise to priority inversions [11, 22, 24], which intuitively occur if a job that should be scheduled (i.e., one among the \( m_k \) highest-priority jobs in its assigned cluster) is not scheduled (e.g., while waiting for a lock). Priority inversions are problematic as they constitute “blocking” that increases a task’s worst-case response time [22, 24]; a real-time locking protocol’s primary purpose is to limit such priority inversion blocking (pi-blocking) so that it can be accounted for by schedulability tests. Pi-blocking includes any locking-related delay that is not anticipated by a schedulability analysis assuming independent jobs, that is, any delay that cannot be attributed to the regular processor demand of higher-priority jobs. Consequently, the exact definition of “priority inversion” depends on the type of the employed schedulability analysis [11].

Schedulability tests are either suspension-aware (s-aware) or suspension-oblivious (s-oblivious). An s-aware schedulability test (such as uniprocessor FP response-time analysis [11]) models self-suspensions and locking-related suspensions explicitly, whereas s-oblivious analysis (such as Baruah’s G-EDF schedulability test [4]) assumes that jobs are always ready. S-oblivious analysis can still be used in the presence of suspensions, but any suspensions must be pessimistically modeled as processor demand by inflating each job’s execution cost by the maximum suspension time prior to applying a schedulability test, which affects the definition of pi-blocking [11].

Def. 1. A job \( J_i \) incurs s-aware (resp., s-oblivious) pi-blocking at time \( t \) if (i) \( J_i \) is not scheduled at time \( t \) and (ii) there are fewer than \( m_k \) higher-priority jobs scheduled (resp., pending) in \( J_i \)’s assigned cluster \( C_k = C(T_i) \) at time \( t \).

The difference between the two types of blocking is illustrated in Fig. 1 for \( m_k = 2 \). During [2, 3], job \( J_3 \) incurs s-aware pi-blocking, but not s-oblivious pi-blocking, as there are \( m_k = 2 \) higher-priority pending jobs (which rules out s-oblivious pi-blocking), but only one higher-priority scheduled job, namely \( J_2 \). Note that it follows from Def. 1 that whenever a job incurs s-oblivious pi-blocking it also incurs s-aware pi-blocking (there cannot be more scheduled jobs than there are pending jobs). As a result, any safe upper bound on s-aware pi-blocking is implicitly also an upper bound on s-oblivious pi-blocking [11].

The focus of this paper is s-aware analysis. We let \( b_i \) denote a bound on the maximum total pi-blocking incurred by any \( J_i \).

An “optimal” real-time locking protocol should minimize priority inversions to the extent possible. To study the fundamental limits of real-time synchronization, prior work [7, 11] proposed maximum pi-blocking (formally, \( \max_{T_i \in \tau} (b_i) \)) as a blocking complexity metric and established lower bounds of \( \Omega(n) \) maximum s-aware (resp., \( \Omega(m) \) maximum s-oblivious) pi-blocking. In other words, there exist pathological task sets such that maximum s-aware (resp., s-oblivious) pi-blocking is linear in the number of tasks (resp., number of processors) under any semaphore protocol [7, 11]. When stating asymptotic bounds, it is assumed that the number and duration of critical sections and self-suspensions per job are bounded by constants (i.e., \( L_{\text{max}} = O(1), N_i = O(1), \) and \( w_i = O(1) \)).

To be asymptotically optimal, a locking protocol must ensure that maximum pi-blocking for any task set is always within a constant factor of the lower bound, which requires a careful choice of progress mechanism to prevent unbounded pi-blocking without causing “too much” pi-blocking itself.

D. Prior Progress Mechanisms and Optimal Locking Protocols

Several progress mechanisms that prevent unbounded pi-blocking by temporarily overruling the underlying JLFP policy have been developed in the past. Under classic priority inheritance [22, 24], which is effective only on uniprocessors and under global scheduling [7, 12], a lock-holder’s priority is raised to that of the highest-priority job that it blocks (if any). Under unrestricted priority boosting [21, 23], which is also effective under clustered and partitioned scheduling, a lock-holder’s priority unconditionally exceeds that of non-lock-holding jobs.

In the s-oblivious case, a bound of \( O(m) \) maximum pi-blocking is asymptotically optimal, which is achieved by the OMLP [12] and the OMIP [9] under any clustered JLFP scheduler. The OMLP is based on priority donation [12], a variant of priority boosting that is suitable only for s-oblivious analysis [7]. The OMIP is based on migratory priority inheritance, a simple priority inheritance extension wherein jobs inherit not only the priorities, but also the cluster assignment of blocked jobs.

In the s-aware case, \( O(n) \) maximum pi-blocking is asymptotically optimal, which is achieved by the P-FMLP+ [7], but only under partitioned JLFP scheduling. The P-FMLP+ is a fairly simple protocol based on priority boosting, where each
resource is protected by a FIFO queue and priority-boosted jobs are scheduled (on each core) in order of non-decreasing lock request times (i.e., jobs executing earlier-issued requests may preempt jobs executing later-issued requests). Key to the analysis of the P-FMLP$^+$ is that local lower-priority jobs are not scheduled (and hence cannot issue requests) while a higher-priority job executes, which, however, is not the case if $m_k > 1$.

Finally, this paper covers the remaining two cases—$O(n)$ maximum s-aware pi-blocking under global and clustered JLFP scheduling—by introducing a novel locking protocol and progress mechanism. We call this new locking protocol the generalized FMLP$^+$ because it effectively reduces to the (simpler) P-FMLP$^+$ under partitioned scheduling, in the sense that they generate the same schedule if $m_k = 1$.

Next, we motivate that a new progress mechanism is needed for the generalized FMLP$^+$ by showing that neither priority inheritance nor unrestricted priority boosting is a suitable foundation for asymptotic optimality under global JLFP scheduling.

### III. Suboptimal S-Aware PI-Blocking

In prior work [11], it was suggested that the global FMLP [6], which simply combines per-resource FIFO queues with priority inheritance, ensures $O(n)$ maximum s-aware pi-blocking under global scheduling. This is indeed the case under G-EDF with implicit deadlines and if all jobs complete by their deadline (see [7, Ch. 6] for a proof). That is, there indeed exist global JLFP policies and common workloads under which the FMLP is asymptotically optimal under s-aware analysis.

However, there also exist global JLFP policies and workloads under which it is not. For instance, under G-FP scheduling, in the presence of arbitrary deadlines, or if jobs may complete after their deadlines, priority inheritance can give rise to non-optimal maximum s-aware pi-blocking since it may cause lock-holding jobs to preempt higher-priority jobs repeatedly. Further, a similar effect occurs also with unrestricted priority boosting. In general, it is thus impossible to construct semaphore protocols that are asymptotically optimal w.r.t. the entire class of JLFP policies using either progress mechanism.

To show this, we construct a task set $\tau^\phi$ for which there exists an arrival sequence such that an independent job incurs $\phi$ time units of s-aware pi-blocking under G-EDF or G-FP scheduling on $m = 2$ processors, where $\phi$ can be chosen arbitrarily (i.e., $\max_{T_i \in \tau^\phi} \{b_i\}$ cannot be bounded in terms of $m$ or $n$).

**Def. 2.** Let $\tau^\phi = \{T_1, \ldots, T_4\}$ denote a set of four tasks with parameters as given in Table I that share $n_i = 1$ resource.

Note that $\max_{\tau^\phi} (p_i) \approx \phi$ (for large $\phi$). This permits an arrival sequence such that a job of $T_3$ incurs s-aware pi-blocking each time that $T_1$, $T_2$, and $T_3$ release jobs in a certain pattern, which, by design, can occur up to $\phi$ times while a job of $T_3$ is pending.

**Lemma 1.** Under G-EDF or G-FP scheduling on $m = 2$ processors, if priority inheritance or unrestricted priority boosting is used as a progress mechanism, then $\max_{T_i \in \tau^\phi} \{b_i\} = \Omega(\phi)$.

**Proof:** We first consider priority inheritance under G-EDF and construct a legal arrival sequence such that $b_3 = \Omega(\phi)$ for any integer $\phi > 0$ under s-aware analysis. Note that the order in which waiting jobs are queued is irrelevant since the single resource is shared only among two tasks. Suppose jobs of $T_4$ require $\ell_1$ for the entirety of their execution, and that jobs of $T_2$ require $\ell_1$ for the latter half of their execution. Further, suppose each job executes for exactly $\ell_i$ time units. If $T_3$ releases its first job at time 0, $T_4$ at time 0.5, and $T_1$ and $T_2$ at time 1, and all tasks release a job periodically every $p_i$ time units, then $J_{3,1}$ (the first job of $T_3$) incurs $\phi$ time units of s-aware pi-blocking. The resulting schedule for $\phi = 4$ is shown in Fig. 2(a). By construction, the first $\phi$ jobs of $T_4$ have lower priority than $J_{3,1}$, whereas the first $\phi$ jobs of $T_1$ and $T_2$ have higher priority than $J_{3,1}$ (assuming w.l.o.g. that deadline ties are resolved in favor of lower-indexed tasks). As a result, $J_{3,1}$ is preempted whenever a job of $T_2$ requests the shared resource, thereby incurring one time unit of s-aware pi-blocking for each of the $\phi$ jobs of $T_4$ that are released while $J_{3,1}$ is pending.

By construction, the schedule of the relevant jobs remains unchanged if G-FP scheduling is assumed and tasks are prioritized in index order (i.e., if $T_3$ has the highest priority and $T_1$ the lowest). Hence, $\max_{T_i \in \tau^\phi} \{b_i\} = \Omega(\phi)$ under priority inheritance and either G-EDF or G-FP.

Finally, a similar schedule arises if unrestricted priority boosting is used instead of priority inheritance: if jobs of $T_4$ are priority-boosted while holding a lock, then $J_{3,1}$ is preempted whenever higher-priority jobs of $T_1$ are released, as illustrated in Fig. 2(b). As a result, $J_{3,1}$ still incurs s-aware pi-blocking for one time unit for each of the $\phi$ jobs of $T_4$ that are released while $J_{3,1}$ is pending. Hence, $\max_{T_i \in \tau^\phi} \{b_i\} = \Omega(\phi)$ under unrestricted priority boosting and either G-EDF or G-FP.

In the general case, it is thus not possible to ensure $O(n)$ maximum s-aware pi-blocking when using priority inheritance or unrestricted priority boosting. However, while some s-aware pi-blocking is clearly unavoidable in the depicted schedule, it is possible to distribute the pi-blocking among the jobs such that no job accumulates “too much” pi-blocking. Such a schedule is shown in Fig. 2(c). In this example, $J_{3,1}$ incurs no s-aware pi-blocking at all since pi-blocking is incurred only by new jobs that arrive after a request is issued. The key observation is that in JLFP schedules pi-blocking is fundamentally tied to preemptions, which occur only if a job is released or resumed. It is thus possible to shift pi-blocking to newly arrived jobs instead of accumulating it in “long-running” jobs such as $J_{3,1}$. We formalize this idea next.

### IV. The FIFO Multiprocessor Locking Protocol

In this section, we formally introduce the generalized FMLP$^+$ and “restricted segment boosting,” the underlying progress mechanism, and prove them to ensure $O(n)$ maximum s-aware pi-blocking. For the sake of simplicity, we initially assume that critical sections do not contain self-suspensions (jobs may still
An interval $[t_0, t_1]$ is a request segment of a job $J_i$ iff $J_i$ issues a lock request at time $t_0 = t - 1$ (i.e., just prior to the beginning of the interval) and releases the lock it requested at time $t_1$. During $[t_0, t_1]$, $J_i$ is either suspended and waiting to acquire a lock, or ready and holding a lock.

Explicitly separating a job’s execution into individual segments allows bounding the pi-blocking incurred during each segment, which at times requires “protecting” certain jobs executing earlier-started segments. To this end, we let $t_r(J_i, t) =$

Def. 3. An interval $[t_0, t_1]$ is an independent segment of a job $J_i$ iff

- $J_i$ is ready and not holding a resource throughout $[t_0, t_1]$.
- $J_i$ is either released or resumed at time $t_0$, or $J_i$ releases a lock at time $t_0 = t - 1$ (i.e., just prior to the beginning of the interval), and
- $J_i$ completes, suspends, or issues a lock request at time $t_1$.

Note that it follows from the above definition that a job is always scheduled at the end of an independent segment since an independent segment ends only when a job suspends, completes, or when it issues a lock request, each of which requires invoking OS services, which in turn a job can do only when it is scheduled.

Complementary to independent segments, request segments denote times during which $J_i$ interacts with the locking protocol.

Def. 4. An interval $[t_0, t_1]$ is a request segment of a job $J_i$ iff $J_i$ issues a lock request at time $t_0 = t - 1$ (i.e., just prior to the beginning of the interval) and releases the lock it requested at time $t_1$. During $[t_0, t_1]$, $J_i$ is either suspended and waiting to acquire a lock, or ready and holding a lock.
In other words, in each cluster \( C_k \) and at any point in time \( t \), the set of priority-boosted jobs \( B(C_k, t) \) includes at most one lock-holding job \( J_b \), and, if \( J_b \) exists, the (up to) \( m_k - 1 \) highest-priority jobs in \( J_b \)’s co-boosting set \( cb(J_b, t) \). Finally, if \( B(C_k, t) \) contains fewer than \( m_k \) jobs, then the remaining \( m_k - |B(C_k, t)| \) processors are used to service the highest-priority jobs in \( \text{ready}(C_k, t) \) that are not already scheduled. An example schedule illustrating restricted segment boosting is shown in Fig. 3 and discussed shortly in Sec. IV-B below.

Restricted segment boosting strikes a balance between progress and pi-blocking by forcing the progress of at least one lock-holding job, thus preventing unbounded priority inversions, but without disturbing the regular, priority-based schedule “too much” by boosting at most one lock-holding job.\(^2\) This provides a strong analytical foundation, which together with the following simple protocol leads to asymptotic optimality.

**Def. 6.** Under the generalized **FIFO Multiprocessor Locking Protocol** (FMLP\(^+\)), there is one FIFO queue \( FQ_q \) for each resource \( q \). Resource requests are satisfied as follows.

1. A scheduled job may request a resource at any time. When a job \( J_l \) requests resource \( q \), it is enqueued in \( FQ_q \) and suspended if \( FQ_q \) was previously not empty.
2. \( J_l \) holds the resource \( q \) when it becomes the head of \( FQ_q \). While holding \( q \), \( J_l \) is eligible for restricted segment boosting, as defined in Def. 5.
3. When \( J_l \) releases \( q \), it becomes ineligible for restricted segment boosting and is dequeued from \( FQ_q \), and the new head of \( FQ_q \) (if any) is resumed.

Next, we briefly illustrate how the FMLP\(^+\) works with an example, and then establish its asymptotic optimality.

**B. Example Schedule**

Fig. 3 shows a C-EDF schedule (with \( K = 2 \) and \( m_1 = m_2 = 2 \)) of seven jobs sharing two resources under the FMLP\(^+\).

We focus on the locking-related events in cluster \( C_1 \). At time 2, \( J_4 \) suspends because it requested \( \ell_1 \), which is being used by \( J_7 \) at the time. \( J_3 \) similarly suspends at time 3 when it requests \( \ell_2 \). At time 5, \( J_3 \) and \( J_4 \) are resumed simultaneously and co-boosting takes effect. First, note that \( \text{boosted}(C_1, 5) = J_4 \) even though \( J_3 \) has higher priority (i.e., an earlier deadline) because

\[ t_r(J_4, 5) = 2 < t_r(J_3, 5) = 3. \]

Further, \( cb(J_4, 5) = \{ J_2 \} \) as \( t_r(J_2, 5) = 0 < t_r(J_3, 5) = 3 \) and \( \Upsilon(J_2) < \Upsilon(J_3) \), and thus \( B(C_1, 5) = \{ J_2, J_4 \} \) and \( H(C_1, 5) = \emptyset \). Note that under unrestricted priority boosting, both \( J_1 \) and \( J_2 \) would have been preempted and incurred pi-blocking, whereas here only \( J_1 \) is preempted and \( J_2 \) is left undisturbed because it is co-boosted. In contrast, \( J_3 \notin cb(J_4, 5) \) since \( J_1 \)’s current segment started at time 4, which is later than \( J_3 \)’s current segment start time, and \( J_5 \notin cb(J_4, 5) \) because it has lower priority than \( J_4 \). Analogously, \( J_2 \) remains scheduled at time 6. Note that because \( J_2 \) benefits from co-boosting, \( J_1 \) incurs pi-blocking instead.

While it may appear counter-intuitive that the higher-priority job \( J_1 \) incurs pi-blocking in place of the lower-priority job \( J_2 \), this is deliberate and highlights a key property of the FMLP\(^+\) with restricted segment boosting: whenever there is a choice, pi-blocking affects jobs with later segment start times instead of jobs with earlier segment start times.

In general, to expedite the completion of critical sections in lower-priority jobs (e.g., \( J_3 \) and \( J_4 \) in Fig. 3), inevitably some jobs have to incur pi-blocking. By shifting the negative effects of priority boosting to later-arrived jobs, restricted segment boosting prevents the accumulation of pi-blocking in individual “long-running” jobs (Fig. 2(c) is in fact an FMLP\(^+\) schedule). This design choice is key to attaining asymptotic optimality, as will become apparent in the following analysis of the FMLP\(^+\).

**C. Asymptotic Optimality: Proof Overview**

We next establish that the FMLP\(^+\) in conjunction with restricted segment boosting ensures \( O(n) \) maximum s-aware pi-blocking under clustered JLFPP scheduling, which matches the known lower bound of \( \Omega(n) \) maximum s-aware pi-blocking [11].

In the remainder of this section, let \( J_i \) denote an arbitrary job of the task under analysis \( T_i \), and let \( C_k = C(T_i) \) denote the cluster in which \( J_i \) executes. We first show that s-aware pi-blocking is limited to \( n - 1 = O(n) \) critical section lengths in each request segment (Lemma 5), and then show that \( J_i \) incurs s-aware pi-blocking for the cumulative duration of at most \( n_k - 1 = O(n) \) critical section lengths during each independent segment as well (Lemma 13). Asymptotic optimality follows if the number of segments is bounded by a constant (Theorem 1).

A source of complexity in the proof is that co-boosted jobs may issue lock requests. As a result, it is not immediately clear that \( J_i \) is not pi-blocked multiple times by a lower-priority job that manages to repeatedly issue requests while being co-boosted (although this is in fact impossible). Key to the proof is hence Lemma 11, which establishes that, if a lower-priority job \( J_b \) causes \( J_i \) to incur pi-blocking for (hypothetically) a second time during one of \( J_i \)’s independent segments, then \( J_b \) was not among the priority-boosted jobs when it issued its request. Intuitively, the main argument of the proof is that if \( J_i \) is not scheduled when \( J_b \) executes its critical section, then this implies the presence of higher-priority jobs, which would have already prevented \( J_b \) from being scheduled at the time that it issued its request. This property lies at the heart of the FMLP\(^+\) and stems from the fact that it resolves all contention strictly in FIFO order.

We begin by bounding pi-blocking during request segments, which are easier to analyze than independent segments.
D. S-Aware PI-Blocking during Request Segments

In Lemma 5 below, we show that maximum s-aware pi-blocking is limited to \((n - 1)\) critical section lengths during each request. In preparation, we first establish three simple lemmas that together encapsulate the observation that, once \(J_i\) has “waited long enough,” there are no more earlier-issued requests that could delay \(J_i\). We begin by noting that the current segment start time always increases when a lock is released.

**Lemma 2.** Let \(J_i\) denote a job of task \(T_b\) pending at time \(t\). If \(J_b\) or an earlier job of \(T_b\) unlocks a shared resource at time \(t_u\) and \(t_u < t\), then \(t_r(J_b, t) > t_u\).

**Proof:** Recall that \(t_r(J_b, t)\) denotes the start time of \(J_b\)’s current segment. If \(J_b\) itself releases a resource at time \(t_u\), then by Defs. 3 and 4 it starts a new segment at time \(t_u + 1\), and hence \(J_b\)’s current segment at time \(t\) cannot have started prior to time \(t_u + 1\). If \(J_b\) is not yet pending at time \(t_u\) (i.e., if an earlier job of \(T_b\) unlocks a resource at time \(t_u\)), then \(J_b\) arrives later at some time \(t_u\), where \(t_u < t\), which implies \(t_u < t \leq t_r(J_b, t)\). \(\Box\)

Next, we observe that any pi-blocking during a request segment implies the existence of another lock-holding job with an earlier-or-equal segment start time.

**Lemma 3.** If \(rs(J_i, t)\) and \(J_i\) incurs s-aware pi-blocking at time \(t\), then there exists a \(J_x\) s.th. \(rs(J_x, t)\) and \(t_r(J_x, t)\) \(\leq t_r(J_i, t)\).

**Proof:** There are two cases to consider.

**Case 1:** \(J_i\) is ready at time \(t\). Then it holds a resource, but is not scheduled. As exactly one lock-holding job is priority-boosted according to Def. 5, and since jobs are boosted in order of non-decreasing segment start times (recall the definition of \(boosted(C_k, t)\)), it follows that another lock-holding job with a segment start time no later than \(t_r(J_i, t)\) is boosted at time \(t\).

**Case 2:** \(J_i\) is not ready at time \(t\). Then it is waiting for the requested resource, which implies that some other job holds the resource at time \(t\). Since under the FMLP+ resource access is granted in FIFO order, it follows that the job holding the requested resource issued its request no later than \(J_i\). \(\Box\)

We observe next that restricted segment boosting guarantees the progress of at least one lock-holding job.

**Lemma 4.** If there exists a lock-holding job at time \(t\), then (one of) the lock-holding, ready job(s) with the earliest segment start time (with ties in segment start time broken arbitrarily) progresses towards completion of its critical section at time \(t\).

**Proof:** The progress guarantee of restricted segment boosting follows immediately from Def. 5. A job is prevented from progressing towards the completion of its critical section if it is ready but not scheduled. By Def. 5, in each cluster \(C_k\), (one of) the job(s) with the earliest segment start time, namely \(boosted(C_k, t)\), is scheduled whenever it is ready. \(\Box\)

Finally, with Lemmas 2–4 in place, it is possible to bound the maximum pi-blocking during any individual request segment.

**Lemma 5.** Let \([t_0, t_1]\) denote a request segment of \(J_i\). During \([t_0, t_1]\), \(J_i\) incurs s-aware pi-blocking for the cumulative duration of at most one critical section per each other task (in any cluster), for a total of at most \(n - 1\) critical sections.

**Proof:** By Lemma 3, \(J_i\) incurs s-aware pi-blocking at time \(t \in [t_0, t_1]\) only if a lock-holding job with a segment start time no later than \(t_r(J_i, t) = t_0\) exists at time \(t\). It follows from Lemma 4 that at least one lock-holding job with a segment start time no later than \(t_0\) progresses towards the completion of its critical section at any time \(t\) that \(J_i\) is pi-blocked. By Lemma 2, once a task unlocks a resource at some time \(t_0\) after time \(t_0\), it has a segment start time later than \(t_0\). Therefore, after a task has completed a critical section at some time \(t_0\) after time \(t_0\), it can no longer pi-block \(J_i\) during \([t_0, t_1]\). Thus at most one critical section per each task other than \(T_i\) causes \(J_i\) to incur pi-blocking during \([t_0, t_1]\), for a total of at most \(n - 1\) critical sections. \(\Box\)

This concludes our analysis of request segments. Next, we consider independent segments.

E. S-Aware PI-Blocking during Independent Segments

The maximum pi-blocking incurred by \(J_i\) during an independent segment is substantially more challenging to analyze because jobs that issue a request after \(J_i\) started its independent segment may still pi-block \(J_i\) (but only once, as we are going to show in Lemma 12). To begin, we establish two simple lemmas on the conditions necessary for \(J_i\) to incur s-aware pi-blocking.

In the following discussion, we let \(B(C_k, t)\) denote the set of boosted jobs and \(H(C_k, t)\) the set of non-boosted jobs selected for scheduling at time \(t\) in cluster \(C_k\), as defined in Def. 5.

**Lemma 6.** If \(is(J_i, t)\) and \(J_i\) incurs s-aware pi-blocking at time \(t\), then \(boosted(C_k, t) \neq \emptyset\).

**Proof:** Follows from the definition of restricted segment boosting. Recall from Def. 1 that, to incur s-aware pi-blocking at time \(t\), two conditions must be met: (i) \(J_i\) must not be scheduled and (ii) fewer than \(m_k\) higher-priority jobs are scheduled in \(J_i\)’s cluster at time \(t\). Suppose no ready, lock-holding job exists at time \(t\) in \(C_k\) (i.e., \(boosted(C_k, t) = \emptyset\)). Then, according to Def. 5, \(B(C_k, t) = \emptyset\). Since \(J_i\) is ready and, according to (i), not scheduled, this implies that \(H(C_k, t) = scheduled(C_k, t)\) contains \(m_k\) higher-priority jobs, which contradicts (ii). \(\Box\)

Next we establish that any lock-holding, ready job that is priority-boosted in cluster \(C_k\) while \(J_i\) incurs s-aware pi-blocking has lower priority than \(J_i\).

**Lemma 7.** If \(is(J_i, t)\), \(J_i\) incurs s-aware pi-blocking at time \(t\), and \(J_b = boosted(C_k, t)\), then \(\mathcal{Y}(J_i) < \mathcal{Y}(J_b)\).

**Proof:** By Lemma 6, \(J_b\) exists. Suppose \(J_i\) does not have a higher priority than \(J_b\) (i.e., \(\mathcal{Y}(J_b) < \mathcal{Y}(J_i)\) since job priorities are unique). Then, assuming that the job priority order is transitive, each \(J_y \in cb(J_b, t)\) has higher priority than \(J_i\) since, by the definition of \(cb(J_b, t)\), for each such \(J_y\), \(\mathcal{Y}(J_y) < \mathcal{Y}(J_b)\) and, by assumption, \(\mathcal{Y}(J_b) < \mathcal{Y}(J_i)\). Hence all jobs in \(B(C_k, t)\) have higher priority than \(J_i\) and, since \(J_i\) is not scheduled at time \(t\), all jobs in \(H(C_k, t)\) also have higher priority than \(J_i\). Thus there are \(m_k\) higher-priority jobs scheduled, which contradicts the assumption that \(J_i\) incurs pi-blocking at time \(t\). \(\Box\)

Having established that \(J_i\) is only pi-blocked when a lower-priority job \(J_0\) is priority-boosted, we next analyze in Lemmas 8–12 the conditions that exist when \(J_0\) issues its request. In the end, the conditions established in the following lemmas will allow
Consider the jobs in \( cb'(J_b, t) \). By Lemma 8, each job in \( cb(J_b, t) \), and hence also each \( J_y \in cb'(J_b, t) \), is ready and not holding a lock at time \( t_x \), and, by the definition of \( cb(J_b, t) \), each such \( J_y \) has higher priority than \( J_y \cap \mathcal{Y}(J_y) < \mathcal{Y}(J_y) \).

By assumption, \( T_b \) unlocks a resource at time \( t_u \); it thus follows from Lemma 2 that \( t_u < t_r(J_b, t_x) \). And since by assumption \( t_r(J_b, t_x) < t_r(J_b, t_x) \), we have \( t_r(J_b, t_x) < t_r(J_b, t_x) \).

Further, recall that \( cb'(J_b, t) \) denotes the \( m_k - 1 \) jobs in \( cb(J_b, t) \) with the earliest segment start times. Since \( J_b \in cb(J_b, t) \), but \( J_b \notin cb'(J_b, t) \), we have \( t_r(J_b, t_x) < t_r(J_b, t_x) \) for each \( J_y \in cb'(J_b, t) \), and therefore also \( t_r(J_b, t_x) < t_r(J_b, t_x) \).

Thus, each \( J_y \in cb'(J_b, t) \) has a higher priority than \( J_b \), has an earlier segment start time than \( J_b \), and is ready and not holding a lock at time \( t_x \) (i.e., \( is(J_y, t_x) \)). Thus, by the definition of \( cb'(J_b, t_x) \), if \( J_y \in cb'(J_b, t_x) \), then also \( J_y \neq cb'(J_b, t_x) \) for each \( J_b \in cb'(J_b, t_x) \), which implies \( |cb'(J_b, t_x)| \leq k - 1 \) and, since \( J_b \notin cb'(J_b, t_x) \), \( |cb'(J_b, t_x)| = k - 1 - 1 \), which implies \( m_k - 1 \geq |cb'(J_b, t_x)| \geq |cb'(J_b, t_x)| + 1 = m_k - 1 + 1 = m_k \). Contradiction.

Lemma 10 shows that \( J_b \) is not among the regularly scheduled jobs at time \( t_x \), and Lemma 11 shows that, under certain conditions, \( J_b \) is not priority-boosted at time \( t_x \). Together, they imply that \( J_b \) cannot be scheduled at time \( t_x \), which conflicts with the assumption that \( t_x \) is the time at which \( J_b \) issues the request that blocks \( J_b \) at time \( t_x \). Next, we establish that the conditions required to apply Lemma 11 would in fact be met if \( J_b \) were blocked twice by the same task.

Consider the jobs in \( cb'(J_b, t) \). By Lemma 8, each job in \( cb(J_b, t) \), and hence also each \( J_y \in cb'(J_b, t) \), is ready and not holding a lock at time \( t_x \), and, by the definition of \( cb(J_b, t) \), each such \( J_y \) has higher priority than \( J_y \cap \mathcal{Y}(J_y) < \mathcal{Y}(J_y) \).

By assumption, \( T_b \) unlocks a resource at time \( t_u \); it thus follows from Lemma 2 that \( t_u < t_r(J_b, t_x) \). And since by assumption \( t_r(J_b, t_x) < t_r(J_b, t_x) \), we have \( t_r(J_b, t_x) < t_r(J_b, t_x) \).

Further, recall that \( cb'(J_b, t) \) denotes the \( m_k - 1 \) jobs in \( cb(J_b, t) \) with the earliest segment start times. Since \( J_b \in cb(J_b, t) \), but \( J_b \notin cb'(J_b, t) \), we have \( t_r(J_b, t_x) < t_r(J_b, t_x) \) for each \( J_y \in cb'(J_b, t) \), and therefore also \( t_r(J_b, t_x) < t_r(J_b, t_x) \).

Thus, each \( J_y \in cb'(J_b, t) \) has a higher priority than \( J_b \), has an earlier segment start time than \( J_b \), and is ready and not holding a lock at time \( t_x \) (i.e., \( is(J_y, t_x) \)). Thus, by the definition of \( cb'(J_b, t_x) \), if \( J_y \in cb'(J_b, t_x) \), then also \( J_y \neq cb'(J_b, t_x) \) for each \( J_b \in cb'(J_b, t_x) \), which implies \( |cb'(J_b, t_x)| \leq k - 1 \) and, since \( J_b \notin cb'(J_b, t_x) \), \( |cb'(J_b, t_x)| = k - 1 - 1 \), which implies \( m_k - 1 \geq |cb'(J_b, t_x)| \geq |cb'(J_b, t_x)| + 1 = m_k - 1 + 1 = m_k \). Contradiction.

Lemma 12 implies that no task can block \( J_b \) more than once during a single independent segment, which yields the desired \( O(n) \) bound on per-segment pi-blocking.

Lemma 13. Let \( [t_0, t_1] \) denote an independent segment of job \( J_b \). During \( [t_0, t_1] \), \( J_b \) incurs s-aware pi-blocking for the cumulative duration of at most one critical section per each other task in cluster \( C_k = C(T_k) \), for a total of \( n_k - 1 \) critical sections.

Proof: By Lemma 6, whenever \( J_b \) incurs s-aware pi-blocking, a lock-holding job \( J_b \) is scheduled, which implies
that \( J_B \) progresses towards the completion of its critical section. Thus, for more than \( n_k - 1 \) critical sections to block \( J_i \), some task \( T_x \) assigned to cluster \( C_k \) needs to block \( J_i \), with at least two critical sections during \([t_0, t_1]\). Let \( t \in [t_0, t_1] \) denote a point in time at which a job of \( T_x \) blocks \( J_i \) with a second critical section. Since tasks request at most one lock at a time, this implies that \( T_x \) unlocked a resource at some point in time before \( t \), and after \( t \), which by Lemma 12 is impossible.

Having established a bound on maximum s-aware pi-blocking during independent segments, we are finally ready to state the main result that establishes the FMLP\(^+\)'s asymptotic optimality.

**Theorem 1.** Under clustered JLFP scheduling with the FMLP\(^+\), \( \max_{T \in \tau} \{b_t\} = O(n) \) for any \( \tau \).

**Proof:** By Lemma 13, during each independent segment, a job \( J_i \) of a task assigned to cluster \( C_k = C(T_k) \) incurs s-aware pi-blocking for the total duration of at most \( n_k - 1 = O(n) \) critical sections. By Lemma 5, during each request segment, \( J_i \) incurs s-aware pi-blocking for the total duration of at most \( n - 1 = O(n) \) critical sections. Recall from Sec. II that \( w_i \) denotes the maximum number of locking-unrelated self-suspensions and \( N_i \) the maximum number of requests for any resource. From Defs. 3 and 4, it follows that any job \( J_i \) has at most \( N_i \) request segments and \( 1 + w_i + N_i \) independent segments since any two independent segments must be separated by a locking-unrelated self-suspension or a request segment. Thus \( b_i \leq N_i \cdot (n - 1) \cdot L^{max} + (1 + w_i + N_i) \cdot (n_k - 1) \cdot L^{max} = O(n) \), assuming \( w_i + N_i = O(1) \) and \( L^{max} = O(1) \) as explained in Sec. II.

The lower bound on maximum s-aware pi-blocking is \((n - 1)\) critical section lengths per request [11]. Assuming \( w = 0 \), the FMLP\(^+\) is hence asymptotically optimal within a factor of

\[
\frac{(1 + 2N_i) \cdot (n - 1) \cdot L^{max}}{N_i \cdot (n - 1) \cdot L^{max}} = 2 + \frac{1}{N_i} \approx 2.
\]

In the remainder of this section, we lift the two simplifying assumptions made so far, namely that critical sections do not contain self-suspensions and that critical sections are non-nested.

**F. Critical Sections with Self-Susceptions**

When synchronizing access to physical resources (e.g., I/O devices), jobs may need to self-suspend during a critical section. For example, a device driver might first acquire a lock serializing access to a device, then prepare buffer contents and request an I/O transaction, self-suspend to wait for the transaction to complete, finally perform cleanup activities when resumed, and only then release the lock. Fortunately, such self-suspensions within critical sections do not invalidate the preceding analysis.

First, note that jobs do not start a new segment when resuming while holding a lock since by Def. 3 a job does not hold a lock during an independent segment, and because by Def. 4 a request segment extends until the lock is released, regardless of whether a job is ready. Second, when a lock-holding job self-suspends, it no longer requires a processor to progress towards completion of its critical section. It therefore does not need to be priority-boosted, and thus does not cause jobs executing independent segments to incur pi-blocking. Thus, the maximum self-suspension time is bounded by a constant and appropriately reflected in the maximum critical section length, the preceding analysis remains valid. More precisely, if \( L_{s,i} \) denotes the maximum self-suspension length of any \( J_i \), while using \( \ell_q \) and \( L_{q,i} \) denotes maximum execution time of any \( J_i \) while using \( \ell_q \) (both w.r.t. a single critical section), then \( L_{q,i} = L_{s,i} + L_{q,i}^{\ast} \) but other jobs executing independent segments incur at most \( L_{q,i}^{\ast} \) pi-blocking due to \( J_i \) being priority-boosted.

Next, we consider nested lock requests.

**G. Nested Critical Sections**

Ward and Anderson showed how to permit fine-grained nesting such that asymptotic optimality is retained with the RNLP [25]. The RNLP consists of two key components, a token lock and a request satisfaction mechanism (RSM), and is generic in the sense that it can be instantiated with different token locks and RSMs to be asymptotically optimal under various scheduling and analysis approaches. Under s-aware analysis, the RNLP’s token lock essentially serves to record a timestamp of earliest segment starting times \( (J_k, t) \) is executing the same segment at times \( t \) and \( t' \) while using the same request satisfaction mechanism \( (RSM) \), and is generic in the sense that it can be instantiated with different token locks and RSMs to be asymptotically optimal under various scheduling and analysis approaches.
while causing only $O(n)$ s-aware pi-blocking itself, it can simply be substituted into Ward and Anderson’s B-RSM.

To summarize, nested critical sections can be supported by redefining request segments to correspond to the start and end of outermost requests, by adopting the generalized FMLP$^+$’s restricted segment boosting (instead of the P-FMLP$^+$’s unrestricted priority boosting) in the RNLP’s B-RSM for s-aware analysis, and by replacing the FMLP$^+$’s simple FIFO queues with the RNLP’s queue structure and access rules (which are also based on FIFO queueing). This concludes our discussion of blocking optimality. We present an empirical comparison of the s-aware FMLP$^+$ with prior s-oblivious locking protocols next.

V. EMPIRICAL EVALUATION

Prior work [8] has shown the P-FMLP$^+$ [7] to be competitive with and—depending on the task set parameters—superior to the two classic locking protocols for P-FP scheduling under s-aware analysis, namely the MPCP [17, 21, 22] and the DPCP [22, 23]. Since the FMLP$^+$ generalizes the P-FMLP$^+$ considered in [8], we compare it instead with two s-oblivious locking protocols for clustered JLFP scheduling: the OMLP [12] and the OMIP [9].

Effective schedulability analysis requires fine-grained blocking bounds that take into account constant factors (e.g., job arrival rates, individual critical section lengths, etc.) that are omitted from the asymptotic bound derived in Sec. IV. We derived suitable bounds (provided online [10]) using a previously introduced analysis technique based on linear programming [8].

A. Experimental Setup

Our experimental setup closely resembles those used in several prior studies [8, 9, 12]. In short, we considered systems with $m \in \{4, 8, 16, 32\}$ processors and, for each system, generated task sets ranging in size from $n = m$ to $n = 10m$. For a given $n$, tasks were generated by randomly choosing a period $p_i$ and utilization $u_i$, and then setting $c_i = p_i \cdot u_i$ (rounding to the next-largest microsecond). Periods were chosen randomly from either a uniform or a log-uniform distribution ranging over $[10\, ms, 100\, ms]$; utilizations were chosen from two exponential distributions ranging over $[0, 1]$ with mean 0.1 (light) and mean 0.25 (medium), and two uniform distributions ranging over $[0, 1]$ (light) and $[0, 0.4]$ (medium).

Critical sections were generated according to three parameters: the number of resources $n_r$, the access probability $p_{acc}$, and the maximum requests parameter $N_{max}$. Each of the $n_r$ resources was accessed by a task $T_i$ with probability $p_{acc}$ and, if $T_i$ was determined to access $\ell_q$, then $N_{i,q}$ was randomly chosen from $\{q, \ldots, N_{max}\}$, and set to zero otherwise. In our study, we considered $n_r \in \{m/4, m/2, m, 2m\}$, $p_{acc} \in \{0.1, 0.25, 0.5\}$, and set $N_{max} = 5$. For each $N_{i,q} > 0$, the corresponding maximum critical section length $L_{i,q}$ was randomly chosen using three uniform distributions ranging over $[1\, \mu s, 15\, \mu s]$ (short), $[1\, \mu s, 100\, \mu s]$ (moderate), and $[5\, \mu s, 1280\, \mu s]$ (long).

We evaluated each parameter combination under both P-FP (using s-aware response-time analysis [1]) and C-EDF with uniform clusters of size $m_c \in \{2, 4\}$. For s-aware analysis under C-EDF, we used Liu and Anderson’s s-aware G-EDF schedulability test [18] on a per-cluster basis; for s-oblivious analysis, we additionally used three s-oblivious G-EDF schedulability tests [2, 4, 5], claiming a task set schedulable if it passed at least one of the tests. We tested at least 400 task sets for each $n$ and each of the 3,456 possible combinations of the listed parameters. All resulting graphs are available online [10]; we focus on major trends in select example graphs due to space constraints.

B. Results

Whether an s-aware locking protocol yields higher schedulability than an s-oblivious locking protocol inevitably depends on the accuracy of the underlying schedulability test (i.e., whether the impact of suspensions is overestimated) and on the tested parameter ranges (i.e., the simpler s-oblivious analysis benefits from $m$ being small). Our results confirm this dependency.

Response-time analysis for uniprocessor fixed-priority scheduling [1] is arguably one of the most accurate s-aware schedulability tests currently known. Consequently, under P-FP scheduling, using locking protocols specifically designed for s-aware analysis can result in substantially higher schedulability than using locking protocols intended for s-oblivious analysis. One such case is shown in Fig. 5(a). The FMLP$^+$ maintains high schedulability until $n \approx 100$, whereas schedulability under the OMIP and the OMLP starts decreasing rapidly already at $n \approx 80$. This is due to the s-oblivious approach of inflating execution times, which results in pessimistic response-time bounds for low-priority tasks with long periods (since each higher-priority task contributes with multiple inflated execution costs). In contrast, the more accurate s-aware analysis under the FMLP$^+$ counts only actual execution as interference, which results in tighter response-time bounds in this scenario. While there also exist of course scenarios without discernible differences between the protocols (e.g., if there is only little contention and blocking is not the “schedulability bottleneck”) and even scenarios in which the s-oblivious protocols outperform the FMLP$^+$ (e.g., the structure of the OMIP and the OMLP, and s-oblivious analysis in general, are favored if $m$ is small and contention is high), our experiments generally show the P-FP/FMLP$^+$ combination to perform well compared to both s-oblivious protocols.

In contrast, this is not the case if the underlying analysis is Liu and Anderson’s s-aware schedulability test for G-EDF [18], which is the best-performing s-aware schedulability test for G-EDF currently available [18]. For example, Fig. 5(b) depicts a scenario where the FMLP$^+$ under s-aware analysis clearly performs significantly worse than both the OMLP and the OMIP.

To confirm that this is due to the underlying analysis of suspensions, and not due to the FMLP$^+$ itself, we also compared the FMLP$^+$ under s-aware analysis with the FMLP$^+$ under s-oblivious analysis. That is, given a task set $\tau$, we first computed for each task $T_i$ a bound $b_i$ on s-aware pi-blocking, and then tested $\tau$ for schedulability once with s-aware analysis [18] and once with s-oblivious analysis [2, 4, 5] using the same bound.$^3$

The results are shown in Fig. 5(c). Interestingly, the FMLP$^+$ under s-oblivious analysis outperforms the FMLP$^+$ under s-aware analysis, even though the FMLP$^+$ is asymptotically

$^3$Recall from Sec. II-C that any bound on s-aware pi-blocking also bounds s-oblivious pi-blocking because the latter implies the former [11].
optimal under s-aware analysis, but not under s-oblivious analysis. In other words, simply treating the identical bound as execution time rather than self-suspension time leads to improved results. This shows that the state-of-the-art analysis of self-suspensions under global scheduling [18] has not yet progressed to the point where self-suspensions due to locking protocols can be efficiently analyzed.

Given the general difficulty of deriving effective multiprocessor schedulability analysis, this is perhaps not too surprising; it does, however, indicate that substantial advances are still required before the analysis of multiprocessor real-time systems can match the current understanding of uniprocessor systems.

VI. CONCLUSION

Prior to this work, it was not known whether it is possible to construct real-time semaphore protocols that ensure $O(n)$ maximum s-aware pi-blocking under global and clustered JLFP scheduling. We have answered this question positively with the generalized FMLP+, the first asymptotically optimal locking protocol for clustered scheduling under s-aware analysis. Notably, the generalized FMLP+ supports non-uniform cluster sizes, non-uniform JLFP policies, self-susceptions, and can be combined with the RNLP [25] to support nested critical sections.

The generalized FMLP+ uses a number of new techniques. Rather than priority inheritance or unrestricted priority boosting, it relies on restricted segment boosting, a novel progress mechanism that tracks the individual independent and request segments of a job at runtime and imposes a FIFO order w.r.t. segment start times. Perhaps counterintuitively, at most one lock-holding job is boosted in each cluster $C_k$, but up to $m_k - 1$ other, non-lock-holding jobs may be co-boosted to prevent the accumulation of pi-blocking in individual jobs.

In future algorithmic work, it will be interesting to explore these techniques in the context of reader-writer and $k$-exclusion protocols [12]. In future systems-oriented work, it will be interesting to assess the practicality of the generalized FMLP+ from the point of view of runtime overheads. While keeping track of segment start times is not problematic (the OS is involved in suspending and resuming jobs and acquiring and releasing semaphores anyway), co-boosting may cause additional preemptions, the overheads of which must be accounted for.

REFERENCES