A Contention-Sensitive Fine-Grained Locking Protocol for Multiprocessor Real-Time Systems*

Catherine E. Jarrett, Bryan C. Ward, and James H. Anderson
Dept. of Computer Science, University of North Carolina at Chapel Hill

ABSTRACT

Prior work on multiprocessor real-time locking protocols has shown how to support fine-grained lock nesting with asymptotically optimal worst-case priority-inversion blocking (pi-blocking) bounds. However, *contention* for each resource has heretofore been considered an unconstrained variable. This paper presents the first fine-grained multiprocessor real-time locking protocol with contention-sensitive worst-case pi-blocking bounds. Contention-sensitive pi-blocking is made possible by incorporating knowledge of maximum critical-section lengths—which must be known *a priori* for analysis anyway—into the lock logic.

1. INTRODUCTION

When designing software for a multiprocessor platform, the need for efficient synchronization looms large. In the domain of real-time systems, multiprocessor locking protocols exist that can be used to realize synchronization requirements, but protocols that can be flexibly applied and do not overly inhibit parallelism have remained somewhat elusive. This is mainly due to difficulties caused by nested lock requests.

There are two principal means for supporting nested locks: coarse-grained locking and fine-grained locking. Under *coarse-grained locking*, all resources that potentially could be locked simultaneously by any task are statically coalesced under one lock. This approach clearly inhibits parallelism. However, under the alternative of *fine-grained locking*, wherein resources are locked individually, correctness-related issues such as deadlock-avoidance become problematic.

Perhaps because of such difficulties, the first fine-grained multiprocessor real-time locking protocol was proposed only recently, in the form of the *real-time nested locking protocol* (*RNLP*) [12, 14]. When expressed in terms of the processor count, m, and the task count, n, the RNLP has worst-case priority-inversion blocking (pi-blocking) bounds that are asymptotically optimal. However, these bounds are actually not an improvement over those possible under coarse-grained locking. This may seem surprising, given the improved run-time parallelism afforded by fine-grained locking. A key motivation for designing a fine-grained protocol is to avoid the artificial inflation of *lock contention* caused by coarse-grained locking. Clearly, analysis that does not consider contention as a first-class parameter (like m and n) cannot demonstrate contention-related benefits. Similarly, protocols that are not designed to be contention sensitive *in the worst case* will not exhibit *worst-case* pi-blocking improvements.

Realizing this, we consider for the first time in this paper the issue of contention sensitivity in the design of multiprocessor real-time locking protocols. We show that such protocols do exist by presenting an asymptotically optimal variant of the RNLP that is contention sensitive. A number of challenges arise when attempting to design a contention-sensitive real-time locking protocol. We discuss some of these challenges below, and how they were addressed in the protocol proposed herein. To provide needed context, however, we first briefly review prior related work.

Prior related work. While the first multiprocessor real-time locking protocols were presented over two decades ago [9, 10], the first protocols to be asymptotically optimal with respect to pi-blocking began to appear only a few years ago [2, 3, 4], and a contention-sensitive protocol has never been presented. The aforementioned RNLP is the only multiprocessor real-time locking protocol proposed to date that supports fine-grained lock nesting [12, 14]. The RNLP is actually a family of protocols, where specific variants are obtained by determining how waiting is realized (spinning vs. suspending), the mechanism used to ensure that tasks make progress, and how pi-blocking is analyzed. (A more detailed description of the RNLP is given later in Sec. 2.) Worst-case pi-blocking under the RNLP is either $O(m)$ or $O(n)$ per resource request, depending on analysis assumptions. As noted above, these worst-case bounds are asymptotically optimal when such bounds are expressed in terms of m and n only. Intuitively, the RNLP achieves optimality by delaying the satisfaction of some lock requests to ensure that earlier-issued outermost requests are never blocked by later-issued ones. Wieder and Brandenburg [15] recently showed that the problem of obtaining *precise* worst-case pi-blocking bounds for FIFO- or priority-ordered nested locks is NP-hard. This result does not directly apply to a protocol such as the RNLP that can opt to delay the satisfaction of some requests.

*Work supported by NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors. The second author was supported by an NSF graduate research fellowship.

RTNS 2015, November 04 - 06, 2015, Lille, France

ISBN 978-1-4503-3591-1/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2834848.2834874

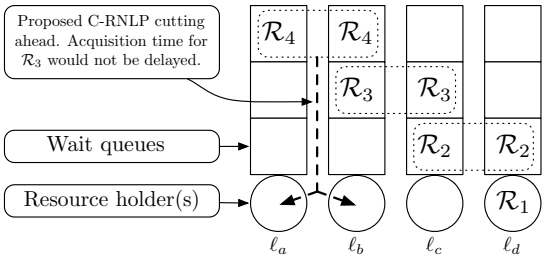

Figure 1: Illustration of the worst-case blocking behavior of the RNLP and the proposed improvement in the C-RNLP. $\mathcal{R}_4$ is transitively blocked by $\mathcal{R}_1$ even though they do not conflict unless $\mathcal{R}_4$ cuts ahead of $\mathcal{R}_3$.

Key problem: transitive blocking chains. The delaying of some requests in the RNLP gives rise to the possibility of long *transitive blocking chains*, which are a major complication in the design of a contention-sensitive locking protocol. Consider the example in Fig. 1, which depicts four requests $\mathcal{R}_1, \ldots, \mathcal{R}_4$ contending for four resources $\ell_a, \ldots, \ell_d$. Blocked requests wait within per-resource FIFO queues. In the depicted situation, $\mathcal{R}_1$ has acquired ℓ_d, $\mathcal{R}_2$ is waiting to acquire both ℓ_c and ℓ_d (and is enqueued on the waiting queue for both resources), $\mathcal{R}_3$ is waiting to acquire both ℓ_b and ℓ_c, and $\mathcal{R}_4$ is waiting to acquire both ℓ_a and ℓ_b. In this example, despite the fact that $\mathcal{R}_1$ and $\mathcal{R}_4$ do not *conflict*, or access a common resource, they are serialized due to requests $\mathcal{R}_2$ and $\mathcal{R}_3$, forming a *transitive blocking chain*. Such transitive blocking chains are the very reason why the current RNLP is not contention sensitive. Observe that the worst-case pi-blocking for $\mathcal{R}_4$ is not upper bounded by the worst-case contention for the resources it accesses, ℓ_a and ℓ_b, but is a function of the maximum number of queued requests.

Contributions. We present the C-RNLP, a contention-sensitive variant of the RNLP. Under the C-RNLP, per-request pi-blocking bounds are asymptotically optimal when such bounds are expressed as a function of m, n, *and lock contention*. In describing the C-RNLP, we mainly focus on a spin-based variant due to space constraints, but variants can be defined corresponding to all RNLP variants.

The key new idea in the C-RNLP is a "cutting ahead" mechanism that breaks long transitive blocking chains. In designing such a mechanism, care must be taken to ensure that requests that are cut ahead of do not experience increased worst-case blocking. The C-RNLP addresses this concern through the incorporation of critical-section length information into the lock and unlock logic. Such information is necessary for *a priori* worst-case blocking and schedulability analysis, so it is reasonable to assume lock requests could be annotated with such information.

The C-RNLP's lock/unlock logic is more complicated than the RNLP's. To address this concern, we implemented a spin-based variant of the C-RNLP and conducted an experimental evaluation on a dual-socket 36-core Intel machine. Our experimental results demonstrate a clear trade-off between lock/unlock overheads and the increased parallelism afforded by using a contention-sensitive locking protocol. These experiments show that the additional overheads can be worth incurring if nesting is common.

Organization. In the rest of the paper, we provide needed background (Sec. 2), describe the C-RNLP, first in a rather abstract way (Sec. 3) and then in a more implementation-oriented way (Sec. 4), present our experimental results (Sec. 5), and finally, conclude (Sec. 6).

2. BACKGROUND

In this section, we provide relevant background material.

Task model. We consider the classic sporadic real-time task model and assume familiarity with this model. Specifically, we consider a task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ scheduled on m processors. We denote an arbitrary job of τ_i as J_i. We limit our attention to job-level fixed-priority schedulers, such as static-priority and earliest-deadline-first (EDF) schedulers, under which the *base priority* of each job is fixed. As discussed below, a synchronization protocol may change the priority of a job.

Resource model. We extend the task model above by assuming that there are n_r shared resources to which tasks in Γ require synchronized access. We denote these resources as $\mathcal{L} = \{\ell_1, \ldots, \ell_{n_r}\}$. We assume that all resources are serially reusable, *i.e.*, no reader/writer sharing. We assume familiarity with synchronization-related terminology, such as *critical section*, *lock*, *unlock*, *outermost critical section*, *etc.* When a job J_i requires a resource ℓ_a, it *issues* a *request* $\mathcal{R}_i$ for ℓ_a.[1] $\mathcal{R}_i$ is *satisfied* as soon as J_i *holds* ℓ_a, and *completes* when J_i *releases* ℓ_a. Two requests *conflict* if they access a common resource. We define the number of issued and not-yet-completed requests that conflict with $\mathcal{R}_i$ when $\mathcal{R}_i$ is issued to be the *contention* that $\mathcal{R}_i$ faces, denoted c_i. The maximum critical-section length is denoted L_{max}.

Priority inversions. Locking protocols designed for real-time systems must enable bounds on *priority-inversion blocking* (or *pi-blocking*) to be determined; such bounds are applied in schedulability analysis. Intuitively, pi-blocking occurs when a job is delayed while lower-priority work is executing in its place. A job may experience pi-blocking while it waits for a resource it has requested—this is called *request blocking*—or as a result of a progress mechanism (described later)—this is called *progress-mechanism-related* (*PMR*) *blocking*.

Analysis assumptions. For asymptotic analysis, we consider critical-section lengths, as well as the number of requests per job to be constant. We consider m and n to be variables when discussing both the proposed C-RNLP and prior protocols, and each c_i for $\tau_i \in \Gamma$ to also be a variable when discussing the C-RNLP. All other parameters are considered to be variable unless stated otherwise.

Progress mechanisms. To ensure that pi-blocking times can be reasonably bounded, real-time locking protocols employ *progress mechanisms* that may temporarily raise a job's *effective priority*. In this paper, we limit attention (due to space constraints) to spin-based protocols that employ the progress mechanism of *priority boosting* [3, 6, 10, 11], wherein a resource-holding job's priority is unconditionally elevated above the highest-possible base (*i.e.*, non-boosted) priority. In suspension-based locks, other progress mechanisms are used such as *priority inheritance* [11] or *priority donation* [4].

Basic RNLP structure. The C-RNLP builds directly on the structure of the RNLP, so a basic understanding of that

[1] We let $\mathcal{R}_i$ denote an arbitrary request of J_i. We have no need to disambiguate different requests from J_i as our analysis focuses on individual outermost requests.

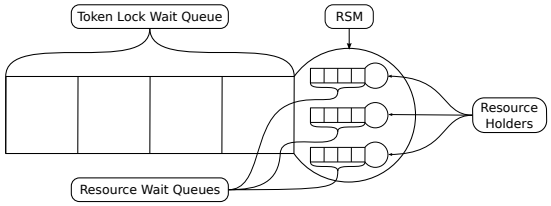

Figure 2: Architecture of the RNLP [12].

protocol is required to understand the results of this paper. As noted in Sec. 1, the RNLP is actually a family of protocols: a given instantiation is obtained by determining how waiting is realized (spinning or suspending), the progress mechanism to employ (priority inheritance, priority boosting, or priority donation), and potentially other factors (such as whether "long" critical sections are to be distinguished from "short" ones [12], and whether reader/writer sharing constraints exist [13]). All such instantiations share a common structure, which consists of two components: a *token lock* and a *request satisfaction mechanism* (*RSM*). This is illustrated in Fig. 2. When a job J_i requires a shared resource, it requests a token from the token lock. Once J_i has acquired a token, it can issue requests for different resources within the RSM. The RSM orders the satisfaction of such requests.

Each particular RNLP instantiation is defined by specifying the token-lock algorithm to use, the total number of tokens, and the RSM implementation to be employed. For the case of spin-based locking with priority boosting (our focus here), each task is assigned a token when it becomes non-preemptive, so the token lock is essentially obviated. For the spin-based variant, worst-case request and PMR blocking are both $O(m)$, which is asymptotically optimal. The other variants have asymptotically optimal pi-blocking bounds as well.

The RSM. When a job is granted a token, it is assigned a timestamp. Within the RSM, each resource has a timestamp-ordered wait queue of pending requests for that resource. When requests can be nested, deadlock is a potential concern. A simple way to obviate any potential deadlock is to impose a partial ordering on resources, and to require that nested locks are acquired according to this ordering [5]. This is an old technique that is commonly used in practice (*e.g.* in the Linux kernel).

In some instantiations of the RNLP, resource orderings can be avoided by using *dynamic group locks* (*DGLs*) [12]. Let D_i be the set of all resources that may be acquired within an outermost lock request $\mathcal{R}_i$. When DGLs are used, instead of issuing requests for resources in D_i in a piecemeal fashion as nested requests are encountered, all such resources are requested simultaneously at the time of token acquisition. (In a real-time system, the set of all such requests would have to be known for schedulability analysis.) In essence, a resource group is identified *dynamically* and coalesced under one lock. The disadvantage of using DGLs is that requests may be issued for resources that are not actually needed. For example, if after acquiring resource ℓ_1, one of ℓ_2 and ℓ_3 is acquired, then requests would be issued for all three resources even though only two are needed. The main advantages are that resource orderings are not required and runtime overheads tend to be lower. Also, worst-case pi-blocking bounds are not altered under our existing analysis techniques when DGLs are used. Note that the usage of DGLs is not the same as coarse-grained locking: coarse-grained locking groups are *statically* defined offline, while DGLs can be used to acquire a *subset* of the resources that are a part of such a group.

3. C-RNLP

In this paper, we extend the RNLP to the C-RNLP, the first fine-grained locking protocol with contention-sensitive worst-case blocking. We show that blocking under the C-RNLP is $O(\min(m, c_i))$ for each task τ_i. This is accomplished by allowing some lock requests to "cut ahead" of other queued requests, thereby limiting the length of transitive blocking chains. For ease of explanation, we assume that lock nesting is realized through the use of DGLs. Thus, each request $\mathcal{R}_i$ specifies a set of resources D_i to be locked.

We define the C-RNLP in two passes: first, we give an abstract description based on how the wait-for graph is updated as requests are issued and completed, similarly to Dijkstra's early work on the dining philosophers problem [5]; later, in Sec. 4, we describe a more concrete implementation that conforms to the abstract specification.

The ordering of requests is maintained in a directed, acyclic wait-for graph $G = (V, E)$, where vertices denote requests and edges denote waiting relationships, *i.e.*, $(\mathcal{R}_i, \mathcal{R}_j) \in E$ means that $\mathcal{R}_i$ is blocked by $\mathcal{R}_j$. Initially G is empty, with $V = \emptyset$ and $E = \emptyset$. When a request is issued, it is added to the graph. This addition of a request along with any associated edges is called an *insertion*. Likewise, when a request completes, the removal of it and all of its edges is a *removal*. We denote the graph that results from G after an insertion or a removal as $G' = (V', E')$. (We similarly use primes in referring to notation relevant to G'.) We say that the graph G' is *instantiated* when it results from applying an insertion or removal operation on G. For now, we assume that these operations are atomic and take zero time to apply. $\mathcal{R}_i$ is *satisfied* when it has no outgoing edges. A resource ℓ_a requested by $\mathcal{R}_i$ is *locked* by $\mathcal{R}_i$ when $\mathcal{R}_i$ is satisfied. A protocol is considered *safe* if at most one request can lock any one resource at a time. We assume that once $\mathcal{R}_i$ is satisfied, $\mathcal{R}_i$ completes within L_i time units. Later we will explore the implications of the violation of this assumption. We now use a series of examples to motivate the rules of the C-RNLP.

Safety. Our first example motivates the rules presented later that ensure that the C-RNLP is safe.

Ex. 1. *Consider the wait-for graph G shown in Fig. 3. Each request requires only ℓ_a. $\mathcal{R}_1$ is satisfied and holds ℓ_a and blocks $\mathcal{R}_2$, as shown by the directed arrow to $\mathcal{R}_1$. Now suppose that $\mathcal{R}_3$ is issued and requires ℓ_a. $\mathcal{R}_3$ is added to G, and we must consider which edges to add. Several positions for inserting $\mathcal{R}_3$ are displayed in Fig. 3, denoted as positions P_1–P_5. For now, it suffices to understand the notion of a position intuitively, but later we shall see that a formal definition is needed. Intuitively, when a request is inserted into G, it is implicitly* reserving *a position. (Actually, with DGLs, a* set *of positions is reserved, but we will ignore this additional complication for now.) We examine these positions below after presenting several definitions.*

Def. 1. *For any request $\mathcal{R}_i$, let $In(\mathcal{R}_i)$ denote its incoming edges, $In(\mathcal{R}_i) = \{(\mathcal{R}_j, \mathcal{R}_i) : (\mathcal{R}_j, \mathcal{R}_i) \in E\}$, and let $Out(\mathcal{R}_i)$ denote its outgoing edges, $Out(\mathcal{R}_i) = \{(\mathcal{R}_i, \mathcal{R}_j) : (\mathcal{R}_i, \mathcal{R}_j) \in E\}$.*

Def. 2. *Let S be the set of satisfied requests: $S = \{\mathcal{R}_i : Out(\mathcal{R}_i) = \emptyset\}$.*

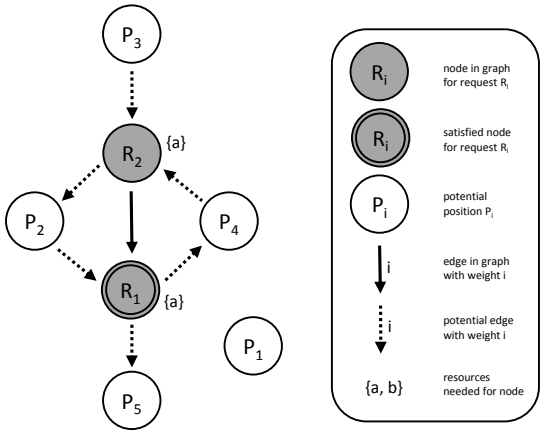

Figure 3: A wait-for graph G and several positions at which $\mathcal{R}_3$ could be inserted. (The legend also applies to subsequent figures. Note that edge weights are not used in this particular figure.)

DEF. 3. *A request $\mathcal{R}_i$* precedes *$\mathcal{R}_j$, denoted $\mathcal{R}_i \prec \mathcal{R}_j$, if there exists a directed path from $\mathcal{R}_j$ to $\mathcal{R}_i$.*

DEF. 4. *A request $\mathcal{R}_i$ has* cut ahead *of $\mathcal{R}_j \in V$ if G' is obtained by the insertion of $\mathcal{R}_i$ and $\mathcal{R}_i \prec \mathcal{R}_j$ is established.*

EX. 1 (CONTINUED). *We examine each of the positions P_1–P_5:*

- *P_1. Here $\mathcal{R}_3$ would have no edges, and thus would be satisfied. However, this would lead to ℓ_a being locked by both $\mathcal{R}_1$ and $\mathcal{R}_3$, which violates safety. Therefore, our protocol should not allow $\mathcal{R}_3$ to be placed at P_1.*
- *P_2. Here $\mathcal{R}_3$ would cut ahead of $\mathcal{R}_2$ and would be waiting for $\mathcal{R}_1$ to finish. This position ensures safety, i.e., ℓ_a will be locked by at most one request at a time.*
- *P_3. Here $\mathcal{R}_3$ would not cut ahead of any request, and would be waiting for $\mathcal{R}_2$. This position is also safe.*
- *P_4. Here $\mathcal{R}_3$ would cut ahead of $\mathcal{R}_1$ and wait for $\mathcal{R}_2$. This maintains safety, but creates deadlock in the system.*
- *P_5. Here $\mathcal{R}_3$ would be cutting ahead of $\mathcal{R}_1$, which should be disallowed since $\mathcal{R}_1$ is already satisfied.*

Motivated by the above example, we note that to ensure safety, there must be a single order in which each request for the same resource will be processed. A newly inserted request should also avoid cutting ahead of a request that has already been satisfied.

DEF. 5. *A set of requests $Q \subseteq V$ has a* unique ordering *if and only if for any two distinct requests $\mathcal{R}_i$ and $\mathcal{R}_j$ in Q, either $\mathcal{R}_i \prec \mathcal{R}_j$ or $\mathcal{R}_j \prec \mathcal{R}_i$.*

DEF. 6. *Let Q_a be the set of requests that require the resource ℓ_a: $Q_a = \{\mathcal{R}_i : \mathcal{R}_i \in V \wedge \ell_a \in D_i\}$.*

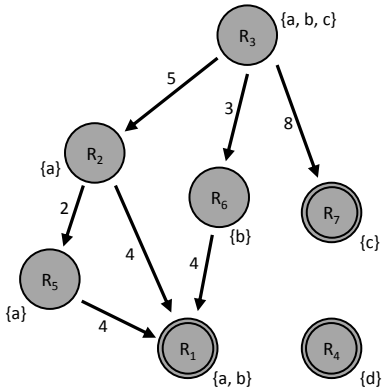

Figure 4: A wait-for graph G that includes seven requests.

DEF. 7. *An insertion into G resulting in G' is a* safe insertion *if: **(i)** for each resource ℓ_a, there is a unique ordering on the set Q'_a; and **(ii)** a new request $\mathcal{R}_i$ does not cut ahead of a satisfied request, i.e., for any $\mathcal{R}_j$ in V, $(\mathcal{R}_j, \mathcal{R}_i) \in E' \Rightarrow \mathcal{R}_j \notin S$. (Note that S is the set of satisfied requests in G.)*

Delay preservation. Now that we have determined which insertions are safe, we investigate which are "best." In order to do so, we add information to G about how long each request will run.

DEF. 8. *The* weight *of an edge $(\mathcal{R}_i, \mathcal{R}_j) \in E$ is given by $W(\mathcal{R}_i, \mathcal{R}_j) = L_j$.*

EX. 2. *Suppose we start with G containing nodes $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ and the edges depicted between them as shown in Fig. 4, after which, $\mathcal{R}_4$, $\mathcal{R}_5$, $\mathcal{R}_6$, and $\mathcal{R}_7$ are inserted in order into G. We examine each of those insertions.*

- *$\mathcal{R}_4$. The insertion of this request with no edges is clearly safe, as no other request in G requires ℓ_d.*
- *$\mathcal{R}_5$. This request cuts ahead of $\mathcal{R}_2$. While this is a safe insertion, it increases $\mathcal{R}_2$'s blocking time, as $\mathcal{R}_2$ must now wait for up to $L_1 + L_5 = 6$ time units to execute, as opposed to waiting for at most $L_1 = 4$ time units.*
- *$\mathcal{R}_6$. This request cuts ahead of $\mathcal{R}_3$ and waits for $\mathcal{R}_1$ to complete. Since $\mathcal{R}_3$ is already waiting for as much as $L_1 + L_2 = 9$ time units, $\mathcal{R}_6$ cutting ahead is acceptable, as it would cause at most $L_1 + L_6 = 7$ time units of waiting time.*
- *$\mathcal{R}_7$. This request cuts ahead of $\mathcal{R}_3$. Using the same reasoning as used with $\mathcal{R}_6$, we see that $L_7 < L_1 + L_2$. However, since $L_7 > L_2$, if $\mathcal{R}_1$ had already been running for close to L_1 time units, then inserting $\mathcal{R}_7$ in this position could delay $\mathcal{R}_3$.*

In order to reason about where to insert a request, we must know how long a satisfied request has been satisfied, as demonstrated by Ex. 2 above.

DEF. 9. *The* running time *of a request $\mathcal{R}_i$, denoted r_i, is the time for which $\mathcal{R}_i$ has been satisfied.*

Note that under our current assumption that $\mathcal{R}_i$ completes within L_i time units, $r_i < L_i$.

DEF. 10. *A* path *in G from $\mathcal{R}_i \in V$ to $\mathcal{R}_j \in V$ is denoted $\mathcal{R}_i \rightsquigarrow \mathcal{R}_j$. The length of this path is given by the sum of the weights of the included edges. This is denoted $|\mathcal{R}_i \rightsquigarrow \mathcal{R}_j|$.*

As seen in Ex. 2, to determine a request $\mathcal{R}_i$'s maximum blocking time, we are only concerned with the maximum distance traversed through G from $\mathcal{R}_i$'s node to a satisfied node, and we must take into account r_j for any $\mathcal{R}_j$ that is satisfied. $\mathcal{R}_i$'s maximum blocking time is given by the value $|A(\mathcal{R}_i)|$, defined next.

DEF. 11. *If $\mathcal{R}_i \notin S$, then let $A(\mathcal{R}_i)$ denote a path $\mathcal{R}_i \rightsquigarrow \mathcal{R}_j$ such that $\mathcal{R}_j \in S$ and $|\mathcal{R}_i \rightsquigarrow \mathcal{R}_j| - r_j$ is maximal; in this case, we define $|A(\mathcal{R}_i)| = |\mathcal{R}_i \rightsquigarrow \mathcal{R}_j| - r_j$. If $\mathcal{R}_i \in S$, then we define $|A(\mathcal{R}_i)| = 0$.*

As suggested by Ex. 2, we want to preserve the existing maximum delays for each request. Requiring preservation of maximum delays would also prevent the possibility of deadlock as shown in Ex. 1.

DEF. 12. *An insertion into G is* delay-preserving *if and only if $(\forall \mathcal{R}_i \in V :: |A'(\mathcal{R}_i)| \leq |A(\mathcal{R}_i)|)$.*

C-RNLP rules. Motivated by the examples above, our C-RNLP implementation must satisfy the following rules:

Rule 1: All requests wait until satisfied.

Rule 2: $\mathcal{R}_i$ is removed when $\mathcal{R}_i$ completes.

Rule 3: A node is inserted at a safe, delay-preserving position in G that gives the lowest $|A(R_i)|$.

Rule 4: Insertions into and removals from G are atomic.

Using the rules of the C-RNLP, we sometimes insert requests into G in different positions than if we were using the RNLP. As shown in Fig. 1, the RNLP orders requests in the order they are issued, with no cutting ahead. However, the C-RNLP would allow $\mathcal{R}_4$ in this example to cut ahead of $\mathcal{R}_3$, if this would not increase $\mathcal{R}_3$'s blocking time, thereby avoiding transitive blocking.

Refining Rule 3. We have presented the abstract rules that define the C-RNLP, but Rule 3 lacks sufficient information to guide an actual implementation. Therefore, we will refine Rule 3 after giving some necessary definitions. We first refine the notion of a position.

DEF. 13. *$\mathcal{R}_i$ and $\mathcal{R}_j$ are* consecutive with respect to ℓ_a *if $\{\mathcal{R}_i, \mathcal{R}_j\} \subseteq Q_a \wedge \mathcal{R}_i \prec \mathcal{R}_j \wedge \neg(\exists \mathcal{R}_l : \mathcal{R}_l \in Q_a :: \mathcal{R}_i \prec \mathcal{R}_l \prec \mathcal{R}_j)$.*

EX. 3. *Given the graph shown in Fig. 5, $\mathcal{R}_1$ and $\mathcal{R}_2$ are consecutive with respect to ℓ_a, and $\mathcal{R}_1$ and $\mathcal{R}_3$ are consecutive with respect to ℓ_b.*

DEF. 14. *A position P_k has at most one incoming edge and at most one outgoing edge, which in an abuse of previous notation, we denote $In(P_k)$ and $Out(P_k)$, respectively.*

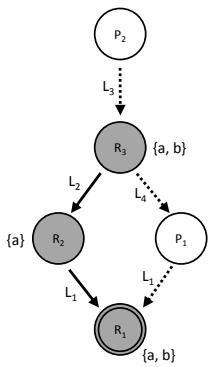

Figure 5: A wait-for graph G with two possible positions for $\mathcal{R}_4$.

- *If $In(P_k) = \emptyset$, then P_k is called a* top-most *position.*
- *If $Out(P_k) = \emptyset$, then P_k is called a* bottom-most *position.*
- *Otherwise, if $In(P_k) = \mathcal{R}_i$ and $Out(P_k) = \mathcal{R}_j$, then P_k is called an* inner *position. An inner position must have $Out(P_k) \prec In(P_k)$.*

*A position is said to be an ℓ_a-*position *if $In(P_k)$ and $Out(P_k)$ are each either $\emptyset$ or a request that includes ℓ_a.*

When a request $\mathcal{R}_l$ is inserted into a graph G, it *reserves* a set of positions X, and $In'(\mathcal{R}_l) = \bigcup_{P_k \in X} In(P_k)$ and $Out'(\mathcal{R}_l) = \bigcup_{P_k \in X} Out(P_k)$.

We now use Ex. 3 to motivate the discussion of a position's capacity, defined below.

EX. 3 (CONTINUED). *Suppose G is as shown in Fig. 5, with request $\mathcal{R}_4$ for resource ℓ_b about to be added to the graph at either position P_1 or position P_2. Both positions would yield a safe insertion. Whether $\mathcal{R}_4$'s insertion at P_1 is delay-preserving depends on the values of L_2 and L_4 and how much longer $\mathcal{R}_1$ will be executing in the worst case. For this insertion to be delay-preserving, $\mathcal{R}_4$ must finish at the latest when $\mathcal{R}_2$ would finish in the worst case (where each request takes exactly its stated maximum time to execute), as this would ensure that all later requests—only $\mathcal{R}_3$ in this scenario—would experience no additional worst-case blocking. To reason about what values of L_4 would meet this condition, we introduce the concept of position* capacity.

DEF. 15. *The* capacity *of a position P_k is defined as:*

$$cap(P_k) = \begin{cases} \infty & \text{if } In(P_k) = \emptyset \\ |A(\mathcal{R}_i)| & \text{if } In(P_k) = \mathcal{R}_i \wedge Out(P_k) = \emptyset \\ \omega - \beta & \text{otherwise} \end{cases} \quad (1)$$

where $\omega = |A(\mathcal{R}_i)| - |A(\mathcal{R}_j)|$ and $\beta = (L_j - r_j)$.

EX. 3 (CONTINUED). *We can now reason about the capacities of P_1 and P_2. For P_1 we obtain, $cap(P_1) = |A(\mathcal{R}_3)| - |A(\mathcal{R}_1)| - (L_1 - r_1) = (L_1 + L_2 - r_1) - 0 - L_1 + r_1 = L_2$. This value indicates how long $\mathcal{R}_3$ will be waiting in the worst case due solely to its blocking on $\mathcal{R}_2$ and taking into account that any request reserving position P_1 would also be waiting for $\mathcal{R}_1$ to finish. Therefore, if L_4 were at most this value, $\mathcal{R}_4$*

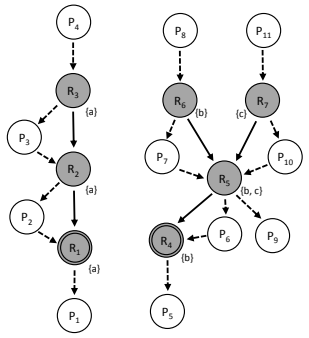

Figure 6: A wait-for graph G with all possible positions shown. P_1–P_4 are ℓ_a-positions, P_5–P_8 are ℓ_b-positions, and P_9–P_{11} are ℓ_c-positions.

could be inserted into P_1 and be delay-preserving. Because $In(P_2) = \emptyset$, $cap(P_2) = \infty$. Intuitively, a request of any length could be inserted into P_2.

DEF. 16. *Let P_{D_i} be a D_i* position set*, such that for each $\ell_a \in D_i$ there is an ℓ_a-position in P_{D_i}.*

When a request $\mathcal{R}_i$ is issued, it must reserve all positions in a D_i position set, P_{D_i}. In turn, $\mathcal{R}_i$ is inserted into G, with $In(\mathcal{R}_i) = \bigcup_{P_k \in P_{D_i}} \{In(P_k)\}$ and $Out(\mathcal{R}_i) = \bigcup_{P_k \in P_{D_i}} \{Out(P_k)\}$.

DEF. 17. *The* capacity *of the position set P_{D_i} is the smallest $|A(\mathcal{R}_i)|$ where $\mathcal{R}_i \in In(P_{D_i})$ (or ∞ if $In(P_{D_i}) = \emptyset$) minus the largest $|A(\mathcal{R}_j)| + L_j$ where $\mathcal{R}_j \in Out(P_{D_i})$ (or 0 if $Out(P_{D_i}) = \emptyset$)*

Note that the capacity of a position set is at most that of each individual position in the set.

Now that we have more fully developed relevant concepts pertaining to positions, we replace Rule 3 with Rule 3′, which upholds Rule 3 and refines how safe and delay-preserving positions are found.

Rule 3′: $\mathcal{R}_i$ is inserted by reserving all positions in P_{D_i} where $L_i \leq cap(P_{D_i})$ and $|A(\mathcal{R}_i)|$ is minimized.

EX. 4. *Suppose G is as shown in Fig. 6. (Note that all possible positions are shown.) Now that we have defined Rule 3′, we can easily examine all possible positions that a request $\mathcal{R}_i$ may reserve and determine which minimizes $|A(\mathcal{R}_i)|$. If $D_i = \{\ell_a\}$, then $\mathcal{R}_i$ must reserve one of the positions P_1–P_4. If instead $D_i = \{\ell_a, \ell_b\}$, then $\mathcal{R}_i$ must reserve both an ℓ_a position P_1–P_4 and an ℓ_b position P_5–P_8. Thus, $\mathcal{R}_i$ will have at least two edges.*

The fact that Rule 3′ upholds Rule 3 follows from the next three lemmas, the latter two of which are stated without proof, as they are straightforward.

LEMMA 1. *$\mathcal{R}_i$ inserted under Rule 3′ is delay-preserving.*

PROOF. Suppose to the contrary that $\mathcal{R}_i$ reserves a set of positions P_{D_i} and the insertion of $\mathcal{R}_i$ is not delay-preserving. Then, there is at least one node that obtains a new edge directed to $\mathcal{R}_i$ that experiences increased blocking. Let $\mathcal{R}_j$ denote such a node. Because $\mathcal{R}_j$'s blocking increases,

$$|A'(\mathcal{R}_j)| > |A(\mathcal{R}_j)|. \tag{2}$$

Let $\mathcal{R}_l$ denote a node to which an outgoing edge from $\mathcal{R}_i$ is directed due to the insertion such that the value $|A(\mathcal{R}_l)| + L_l$ is maximized. (If no such node $\mathcal{R}_l$ exists, then a slightly simpler version of the proof that follows can be applied. We omit this case due to space constraints.) Then, by Def. 11,

$$|A'(\mathcal{R}_i)| = |A'(\mathcal{R}_l)| + L_l - r_l. \tag{3}$$

Note that, after the insertion of $\mathcal{R}_i$, there can be no path from $\mathcal{R}_l$ to $\mathcal{R}_i$, for then a cycle would exist (and hence, deadlock). More technically, if such a cycle were caused, then the set of positions P_{D_i} would have zero or negative capacity by Def. 17, and thus $\mathcal{R}_i$ would not have been inserted by Rule 3′. Because the insertion of $\mathcal{R}_i$ does not result in any path from $\mathcal{R}_l$ to $\mathcal{R}_i$,

$$|A'(\mathcal{R}_l)| = |A(\mathcal{R}_l)|. \tag{4}$$

Because the insertion of $\mathcal{R}_i$ caused $\mathcal{R}_j$'s blocking to increase, $|A'(\mathcal{R}_j)|$ depends on $\mathcal{R}_i$, which upon insertion has not yet accessed any resource, as it waits on $\mathcal{R}_l$. Therefore, by Def. 11,

$$|A'(\mathcal{R}_j)| = |A'(\mathcal{R}_i)| + L_i. \tag{5}$$

The following reasoning establishes $cap(P_{D_i}) < L_i$.

$$\begin{aligned}
cap(P_{D_i}) &\leq \{\text{by Def. 17}\} \\
&\quad |A(\mathcal{R}_j)| - (|A(\mathcal{R}_l)| + L_l) \\
&< \{\text{by (2)}\} \\
&\quad |A'(\mathcal{R}_j)| - (|A(\mathcal{R}_l)| + L_l) \\
&< \{\text{by (4)}\} \\
&\quad |A'(\mathcal{R}_j)| - (|A'(\mathcal{R}_l)| + L_l) \\
&= \{\text{by (5)}\} \\
&\quad |A'(\mathcal{R}_i)| + L_i - (|A'(\mathcal{R}_l)| + L_l) \\
&= \{\text{by (3)}\} \\
&\quad |A'(\mathcal{R}_l)| + L_l - r_l + L_i - (|A'(\mathcal{R}_l)| + L_l) \\
&= \{\text{by simplification}\} \\
&\quad L_i - r_l \\
&\leq \{\text{because } r_l \geq 0\} \\
&\quad L_i
\end{aligned}$$

Therefore, for $\mathcal{R}_j$ to have experienced increased blocking upon the insertion of $\mathcal{R}_i$, $\mathcal{R}_i$ must have reserved a set of positions with $cap(P_{D_i}) < L_i$, which violates Rule 3′. Thus, such a set of reservations cannot occur. □

LEMMA 2. *$\mathcal{R}_i$ inserted under Rule 3′ is safe.*

LEMMA 3. *$\mathcal{R}_i$ is inserted by following Rule 3′ with the same $|A(\mathcal{R}_i)|$ as by following Rule 3.*

Establishing a bound. Based on the rules for the C-RNLP, we can bound the latest time at which a request could be satisfied.

LEMMA 4. *Suppose that G is instantiated at time t and G' is instantiated at time t'. If $|A(\mathcal{R}_i)| > 0$, then $|A'(\mathcal{R}_i)| \leq |A(\mathcal{R}_i)| - (t' - t)$.*

PROOF. The lemma follows because reservations are delay-preserving, and because all satisfied requests blocking $\mathcal{R}_i$ execute continuously between times t and t', due to priority boosting. □

DEF. 18. *Let $B_i = |A(\mathcal{R}_i)|$ at the time of $\mathcal{R}_i$'s insertion.*

LEMMA 5. *$\mathcal{R}_i$ is satisfied within B_i time units from its initial insertion.*

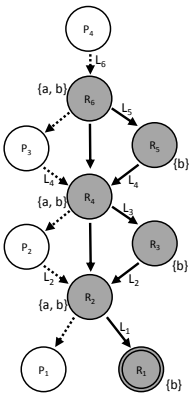

Figure 7: G with worst-case blocking for request $\mathcal{R}_i$ requiring ℓ_a, with L_i just greater than L_1, L_3, and L_5.

PROOF. Follows directly from Lem. 4. □

LEMMA 6. *$B_i < c_i(L_{max} + L_i)$.*

PROOF. Suppose $\mathcal{R}_i$ requires only one resource. By Rule 3′, $\mathcal{R}_i$ cannot be inserted into a position P_k where $cap(P_k) < L_i$. In the worst-case scenario, $\mathcal{R}_i$ would have to be inserted into a top-most position with all other relevant positions having capacity just less than L_i. This scenario is shown in Fig. 7, where $\mathcal{R}_i$ requires ℓ_a and L_1, L_3, and L_5 are all just less than L_i. In this case, $B_i < c_i L_{max} + c_i L_i$.

In the more general case, $\mathcal{R}_i$ requires a set of resources. Again, the worst-case scenario would only allow the insertion of $\mathcal{R}_i$ at a top-most position, with each other set of positions having capacity less than L_i. This scenario establishes the same bound. Therefore, $B_i < c_i(L_{max} + L_i)$. □

THEOREM 1. *A request $\mathcal{R}_i$ is satisfied within $c_i(L_{max} + L_i)$ time units.*

PROOF. Follows from Lem. 5 and Lem. 6. □

Uniform C-RNLP. Referring back to the proof of Lem. 6, when request $\mathcal{R}_i$ is inserted, call a potential position for it *problematic* if that position has a non-zero capacity that is less than L_i. In the bound established in Thm. 1, the term $c_i L_i$ arises because there can be up to c_i problematic positions when $\mathcal{R}_i$ is inserted. We now briefly discuss a second variant of the C-RNLP, which we call the *uniform C-RNLP*, in which problematic positions cannot occur.

In this variant, the time line is segmented into fixed-length *frames* of size L_{max}, wait-for graph modifications are allowed to occur only at frame boundaries, with all node removals occurring before any node insertions (we are still assuming that these graph modifications take zero time), and all satisfied requests are required to remain satisfied for exactly L_{max} time units. With these modifications, all parameters that affect the graph's basic structure are a multiple of L_{max} and it can be formally shown by examining (1) that each position has a capacity that is a multiple of L_{max}, or is 0, or is ∞. This implies that problematic positions cannot exist. As a result, the bound in Thm. 1 reduces to $c_i L_{max}$. However, because any request that occurs within a frame is delayed until the next frame boundary, this blocking bound must be increased by L_{max} (the frame size). From this discussion, we have the following theorem.

THEOREM 2. *Under the uniform C-RNLP, a request $\mathcal{R}_i$ is satisfied within $(c_i + 1)L_{max}$ time units.*

COROLLARY 1. *Under either variant of the C-RNLP, worst-case request blocking is $O(\min(m, c_i))$.*

PROOF. Follows from Thms. 1 and 2, and our analysis assumptions and assumed progress mechanism of priority boosting, which limits contention to at most m. □

Removal of assumptions. In the two variants of the C-RNLP just described, safety is maintained even if resource-holding times are longer than specified. In fact, our stated blocking bounds actually remain unaltered if a request $\mathcal{R}_i$ can hold a resource for longer than L_i time units, provided no request holds any resource for longer than L_{max} time units. However, if a request is allowed to hold a resource for longer than L_{max} time units, then our bounds no longer hold. From a practical point of view, this really is not a deficiency, because if reliable bounds on resource-holding times are not known, then it is pointless to attempt to conduct schedulability analysis.

So far, we have assumed that all modifications to a wait-for graph take zero time. In reality, of course, such modifications will entail some overhead. Such overheads can be factored into our blocking bounds using straightforward techniques. If these overheads are regarded as constants (which is something assumed in all prior work on real-time locking protocols known to us), then blocking times under both protocol variants remain contention sensitive.

4. IMPLEMENTATION

In the prior section, we described two variants of the C-RNLP at an abstract level. In this section, we present a concrete implementation of the uniform variant.

Data structures. Fig. 8 depicts the key data structures of our implementation. A request acquires resources by reserving a set of positions in a reservation table, *Table*, which is an array of bit masks. Each bit in a bit mask represents a resource that can be reserved or locked for a *frame* of time of length L_{max}. We use bit masks because modern processors provide fast register-level operations for manipulating them. A request reserves a set of positions by selecting a row in *Table* (wrapping if necessary) and by setting the bits corresponding to its needed resources in that row's bit mask. The arrays *Enabled* and *Blocked* are both indexed by frame. A request $\mathcal{R}_i$ that has reserved a position given by row k is satisfied when *Enabled*$[k] = 1$ holds; we say that such a row is *enabled*. The manner in which *Blocked* is used is explained later. Several other variables are used as well. *Head* indicates the currently enabled row of *Table*. *Pending_requests* records the number of pending (issued but not completed) requests. *Size* gives the size of each array, which must be at least the number of pending requests at any time.

Pseudo code. The lock and unlock routines of our implementation are shown separately in Alg. 1 and Alg. 2, respectively. Note that shared variables are capitalized, while variables specific to a request or a function call are lowercase.

In Fig. 8, we show how a set of requests in a wait-for graph G would be represented in our implementation. As in G, requests that will be fulfilled later appear higher in *Table*. $\mathcal{R}_1$ and $\mathcal{R}_2$ are satisfied, as is indicated by *Enabled*$[0] = 1$. $\mathcal{R}_3$

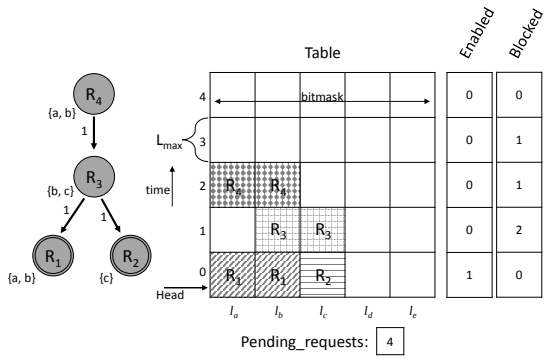

Figure 8: Illustration of G and *Table*.

has reserved ℓ_b and ℓ_c in *Table*[1]. The value 2 in *Blocked*[1] indicates that $\mathcal{R}_3$ is waiting for both $\mathcal{R}_1$ and $\mathcal{R}_2$ to finish. $\mathcal{R}_4$ has similarly reserved ℓ_a and ℓ_b in *Table*[2]. We now explain the lock and unlock routines by means of examples.

Ex. 5. *Suppose $\mathcal{R}_5$ for ℓ_b and ℓ_c is issued when the system state is as in Fig. 8. First, a lock on Sublock is obtained at Line 2 of Alg. 1 to serialize access to the protocol's shared state. For the request under consideration, the test in Line 3 would evaluate to true, so searching in Table for an available set of positions would begin. This search occurs in the loop at Lines 5 and 6, where the bit masks of different rows are checked. Upon termination of this search, the variable start indicates the corresponding row where the available positions were found. Following the search, next is set, Blocked[next] and Pending_requests are incremented, and the appropriate row of Table is updated at Lines 11–14. The lock on Sublock is then released at Line 15. The task issuing the request then spins on Enabled[start] at Lines 16 and 17.*

Ex. 6. *Now suppose that $\mathcal{R}_2$ completes. A lock on Sublock must first be obtained at Line 21 in Alg. 2. At Line 22, Table is cleared of $\mathcal{R}_2$ by bit-wise ANDing the negation of $\mathcal{R}_2$'s requested bitmask and its row in Table. Note that Head now points to the acquired row of $\mathcal{R}_2$. Blocked[next] and Pending_requests are then decremented at Lines 23 and 24. If there are no more requests blocking requests in the row indicated by next, i.e., Blocked[next] = 0 at Line 25, then that row is enabled and Head is updated at Lines 26–28. Finally, the lock on Sublock is released at Line 30.*

The above implementation limits the total number of resources to be at most the number of bits per bit mask, *e.g.*, on a 64-bit machine, there could be at most 64 resources in total. This restriction can be eased by applying results on the *renaming problem*, which is a classic problem in work on concurrent algorithms [1]. In this problem, tasks that have identifiers over a large name space are "renamed" by giving them identifiers over a small name space—such names can be both acquired and released [8]. A renaming algorithm could be applied in our context to assign a unique identifier to any resource while it is being used. As a result, we would merely need to limit the total number of *concurrently requested* resources to be at most the bit mask size. While renaming algorithms can be implemented with low overhead using appropriate atomic instructions, we have not yet fully explored their use. We note also that it is possible to extend our implementation by using several bit masks per row of *Table*, though this would increase lock/unlock overheads.

Algorithm 1 Uniform C-RNLP Lock

```
1: procedure C-RNLP Lock(requested)
2:     lock(Sublock)
3:     if Pending_requests > 0 then
4:         start ← (Head + 1) mod SIZE
5:         while (Table[start] & requested) ≠ 0 do
6:             start ← (start + 1) mod SIZE
7:         end while
8:     else
9:         start ← Head
10:    end if
11:    next ← (start + 1) mod SIZE
12:    Blocked[next] ← Blocked[next] + 1
13:    Pending_requests ← Pending_requests + 1
14:    Table[start] ← Table[start] | requested
15:    unlock(Sublock)
16:    while Enabled[start] ≠ 1 do
17:        /* null */
18:    end while
19: end procedure
```

Algorithm 2 Uniform C-RNLP Unlock

```
20: procedure C-RNLP Unlock(requested)
21:    lock(Sublock)
22:    Table[Head] ← Table[Head] & ∼requested
23:    Blocked[next] ← Blocked[next] − 1
24:    Pending_requests ← Pending_requests − 1
25:    if Blocked[next] = 0 then
26:        Enabled[Head] ← 0
27:        Head ← next
28:        Enabled[Head] ← 1
29:    end if
30:    unlock(Sublock)
31: end procedure
```

5. EXPERIMENTAL EVALUATION

To evaluate our C-RNLP implementation, we conducted a series of experiments, measuring lock/unlock overheads, as well as observed blocking and runtime performance.

We performed these experiments on a dual-socket, 18-cores-per-socket Intel Xeon E5-2699 platform. We compared the C-RNLP to the RNLP [14] and Mellor-Crummey and Scott's queue lock (the MCS lock) [7], applied as a coarse-grained lock, treating all resources as one resource group.

Measuring lock/unlock overheads. We measured lock and unlock overheads (*i.e.*, the time it takes to perform the lock and unlock calls of each protocol) as a function of the number of requested resources, *i.e.*, $|D_i|$, the total number of managed resources, n_r, and the number of contending tasks, n. Tasks were statically pinned one per core.[2] We considered $n \in \{2, 4, \ldots, 36\}$, $n_r \in \{1, 2, 4, 6, \ldots, 64\}$, and $|D_i| \in \{1, 2, 4, 8, 12, \ldots, 48\}$ where $|D_i| \leq n_r$. Each contending task executed lock and unlock calls in a tight loop 1,000 times, with a negligible critical section, so as to maximize contention for shared variables within the lock and unlock calls.

The C-RNLP lock and unlock calls themselves acquire a lock (which is true of the RNLP as well). Thus, blocking can occur within the lock/unlock logic. This blocking is part of the lock/unlock *overhead*, and we therefore refer to it as

[2]Tasks were pinned to cores on the same socket when possible.

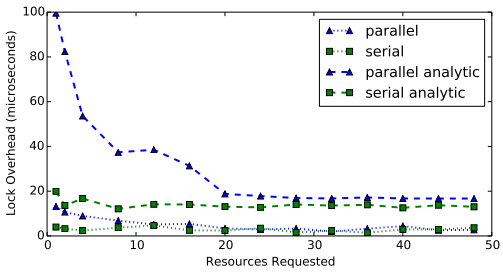

Figure 9: Measured C-RNLP lock overhead as a function of $|D_i|$ for $n = 36$ and $n_r = 64$.

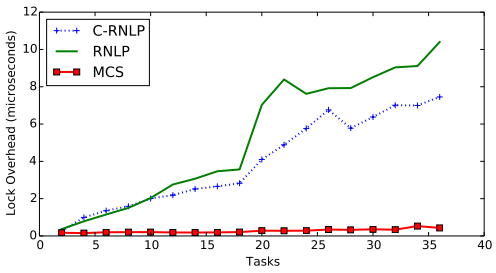

Figure 10: Lock overhead as a function of task count n for $n_r = 64$ and $|D_i| = 4$.

overhead blocking, to differentiate it from the *protocol blocking* (Lines 16 and 17 in the C-RNLP) experienced while waiting for C-RNLP-protected resources to be released. Therefore, when measuring lock/unlock overheads, we included overhead blocking in the measurement, but not protocol blocking. A similar methodology was applied to the RNLP and the MCS lock (the latter has no overhead blocking). We measured these overheads using the cycle counter, and report the 99^{th} percentile of observed lock/unlock overheads, so as to filter any spurious measurements. Due to space constraints, we focus only on lock overheads here; a similar story emerges when considering unlock overheads. Also, we can only present a subset of the data we collected.

Fig. 9 gives several curves pertaining to the lock-overhead data we collected for the C-RNLP. Each curve is plotted with respect to the number of requested resources, $|D_i|$. For the curve labeled *serial*, D_i was defined to be the same for all tasks (*i.e.*, resources are accessed serially). For the curve labeled *parallel*, D_i was determined at random (*i.e.*, resources can be accessed in parallel). These two curves include any overhead blocking. One might argue that it is better to account for such blocking analytically rather than by relying on measurement. To assess this possibility, we present two additional curves, *serial analytic* and *parallel analytic*, which were derived by measuring lock overheads with overhead blocking *excluded* and by inflating that measurement by accounting for overhead blocking analytically. Based on Fig. 9, we make the following two observations.

Obs. 1. *The complexity of the lock logic in the C-RNLP requires an analytical estimation of worst-case overhead blocking as opposed to a purely measurement-based approach.*

We observed a surprising overhead trend, supported by Fig. 9, in the *parallel* case in that, as the number of requested resources $|D_i|$ increased, the observed worst-case lock and unlock overheads *decreased*. In comparison to the *serial* case, the observed overheads were higher, which was also surprising given the potential that less of *Table* needed to be considered if more requests could be processed in parallel. Initially, we conjectured this was because our experimental process was not able to produce the worst-case overhead blocking. To address this, we considered overhead blocking analytically. The overheads using this analytical approach are indeed much higher, and therefore in a truly hard-real-time safety-critical system, an analytically rigorous approach must be taken to account for overhead blocking.

Interestingly, this observation has implications for other locking protocols that themselves employ a lock. In particular, suspension-based locks, which are implemented in the kernel, often acquire kernel-based spin locks. To truly bound the worst-case overheads of such protocols, a similar analytical approach should be taken to account for overhead blocking.

Obs. 2. *Runtime parallelism can increase worst-case lock and unlock overheads for the C-RNLP.*

This can be seen by comparing the curves for the serial cases in Fig. 9 to those for the parallel cases. We found the better performance in the serial cases quite surprising, because in these cases tasks do not "share" rows of *Table* and hence longer searches through *Table* are needed (indeed, we confirmed that less of *Table* was typically searched in the parallel case). However, at least in the context of our experimental framework, greater parallelism allows requests to be issued at a faster rate, since they experience less protocol blocking. This in turn increases contention for the shared variables in the C-RNLP implementation. We conjecture this increased contention resulted in cache invalidations and additional cohearance traffic that resulted in increased memory latency and therefore higher overheads.

The measurements discussed so far pertain only to the C-RNLP. In Fig. 10, we plot measured lock overheads for all three considered protocols as a function of the task count n (recall that in our experimental framework, $n \leq m$). Two observations are supported by this data.

Obs. 3. *For the C-RNLP, observed overheads increase dramatically when resources are shared across sockets.*

This observation applies to the RNLP as well. It can be confirmed by examining the sharp rise in the curves for both protocols between $n = 18$ and $n = 20$. This rise is due to increased memory latencies due to cross-socket interactions.

Obs. 4. *Fine-grained locking protocols have higher overhead than coarse-grained ones.*

This can be easily seen in Fig. 10. This result is not surprising, as coarse-grained protocols require far simpler lock/unlock logic. This exposes an interesting tradeoff: fine-grained protocols offer decreased blocking with higher overheads, while coarse-grained protocols offer decreased overheads at the expense of increased blocking.

Runtime performance. To examine this tradeoff, we measured the total lock and unlock overhead (including overhead blocking) and protocol blocking, which we hereafter simply refer to as *total blocking*. Specifically, we tested the

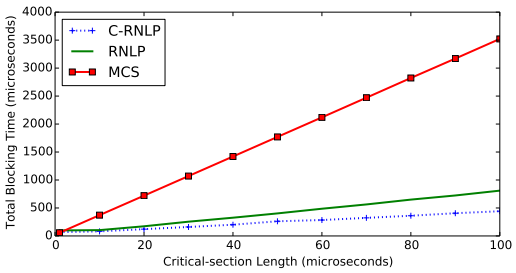

Figure 11: Total blocking time of lock call as a function of critical-section length for $n = 36$, $n_r = 64$, and $|D_i| = 2$.

same configuration parameters as in the previous experiments, and we also varied critical-section lengths within $\{1, 10, 20, \ldots, 100\}$ microseconds. Fig. 11 is a sample graph from this study, where $n = 36$, $n_r = 64$, and $|D_i| = 2$, though many others are available online.[3] Based on these results, we make the following observation.

Obs. 5. *When critical-section lengths are greater than several microseconds and some parallelism is possible, the C-RNLP has less total blocking than previous protocols.*

This can be seen quite dramatically in Fig. 11, where both the RNLP and C-RNLP substantially outperform the MCS lock, with the C-RNLP (because of the greater parallelism it affords) besting the RNLP. Fig. 11 was chosen to highlight the best-case scenario for the C-RNLP, *i.e.*, the case in which the most cutting ahead is possible. Obviously, for cases in which there is little if any cutting ahead (*e.g.*, the serial case described previously), the C-RNLP has inferior total blocking to the MCS lock as it results in the same request ordering, but with higher overheads. Additionally, in the graphs available online, there exist cases in which the overhead of the C-RNLP results in higher total blocking than the other protocols.

6. CONCLUSION

We have presented two variants of the C-RNLP, the first multiprocessor real-time locking protocol to be contention sensitive. The C-RNLP uses novel techniques, which incorporate knowledge of critical-section lengths, that enable requests to be ordered in a way that breaks long transitive-blocking chains. These techniques increase lock and unlock overheads. However, the experimental results presented herein suggest that these overheads may be worth incurring in some use cases.

Due to space constraints, we have limited our attention to spin-based variants of the C-RNLP in this paper. In a future expanded version of the paper, we will fully present C-RNLP variants corresponding to all RNLP variants and show that worst-case pi-blocking for each C-RNLP variant is asymptotically optimal when lock contention is considered.

To complement the measurement-based study presented herein, we plan in future work to examine overhead/parallelism tradeoffs in the context of a full overhead-aware schedulability study that considers several coarse- and fine-grained locking protocols. Such a study will require implementations of all C-RNLP variants. While the spin-based implementation presented herein entails only user-level code, suspension-based variants require kernel-level implementations. For all variants, we intend to consider the possibility of using a "lock server" as an option to reduce overheads. The idea here is to bind such a server to one processor and have tasks acquire and release resources by invoking the server. Such a server would tend to run "cache hot," which might significantly reduce overheads. We also intend to more fully investigate the potential of using renaming algorithms (see Sec. 4), as well as other techniques that might reduce overheads or increase flexibility.

[3] http://www.cs.unc.edu/~anderson/papers.html

References

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *FOCS '87*.

[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*.

[3] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*.

[4] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*.

[5] E. Dijkstra. Two starvation free solutions to a general exclusion problem. EWD 625.

[6] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS '09*.

[7] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization of shared-memory multiprocessors. *Transactions on Computer Systems*, 9(1):21–64, 1991.

[8] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.

[9] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS '90*.

[10] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS '88*.

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

[12] B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *RTNS '13*.

[13] B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *IPDPS '14*.

[14] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*.

[15] A. Wieder and B. Brandenburg. On the complexity of worst-case blocking analysis of nested critical sections. In *RTSS '14*.