A Contention-Sensitive Multi-Resource Locking Protocol for Multiprocessor Real-Time Systems

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

***Abstract*—Recent work on real-time multiprocessor synchronization has produced several locking-protocol variants that support fine-grained lock nesting with asymptotically optimal worst-case blocking bounds. However, in such work, *contention* for each resource has been considered an unconstrained variable. This paper presents a new fine-grained multiprocessor real-time locking protocol with contention-sensitive worst-case blocking, which dominates prior work analytically. This improvement is made possible by incorporating the maximum critical section length, which must be known a priori for analysis, into the lock logic. A spin-based variant of the proposed protocol is prototyped, and observed lock and unlock overheads are less than 0.5 μs on a quad-core machine. Such overheads are likely sufficiently small to be practical in many systems.**

I. Introduction

To realize the performance benefits of multiprocessor systems for real-time applications, efficient synchronization mechanisms are required that allow for synchronized tasks to execute in parallel whenever possible. However, until only recently, no multiprocessor real-time locking protocols supported *fine-grained* locking, *i.e.*, nested lock requests. Nesting lock requests, if not handled properly, can give rise to increased worst-case blocking bounds on account of notoriously difficult-to-analyze transitive blocking [5]. For this reason, historically in many multiprocessor real-time systems only *coarse-grained* locking protocols, or locking protocols that do not support nested requests, were employed. Under coarse-grained locking, resources that may be accessed concurrently must be *grouped* and treated as a single lockable entity.

Fine-grained locking allows for increased parallelism over coarse-grained locking as multiple non-conflicting requests may be concurrently satisfied for resources in the same group. However, existing fine-grained multiprocessor locking protocols have the same worst-case blocking bounds asymptotically as coarse-grained ones. Therefore, the parallelism afforded by fine-grained locking is not reflected in the blocking bounds asymptotically. While using more precise analysis yields improved worst-case blocking bounds (due to smaller constant factors) for fine-grained protocols [2], [3], it is disappointing that fine-grained locking has not yet proven to provide an asymptotic benefit. Perhaps the reason that no asymptotic benefits have been shown for fine-grained locking is due to incomplete analysis assumptions that do not reflect many of the nuances of fine-grained locking as compared to coarse-grained locking. In particular, *contention*, or the number of competing tasks for each individual resource, is not explicitly considered in previous work. Instead, previous analysis considered only the total number of requests for all resources within the group.

Existing multiprocessor real-time locking protocols have been shown to be asymptotically optimal under previous analysis assumptions that did not explicitly consider contention for each resource. In particular, protocols have been developed with $O(m)$ worst-case blocking, where m is the number of processors in the system. However, previous research suggests that nested resource requests are often uncommon [1]. Therefore, in many cases, contention for individual resources may be less than $O(m)$. We therefore endeavor to develop a *contention-sensitive* multiprocessor real-time locking protocol, *i.e.*, a locking protocol in which worst-case blocking is upper bounded by the contention for each of the potentially accessed individual resources.

II. The C-RNLP

In this work, we extend the *Real-time Nested Locking Protocol* (*RNLP*) [2], [4], to the contention-sensitive RNLP (C-RNLP). Before describing the C-RNLP, we briefly review the essential intuition of the RNLP.

The RNLP can be realized by coalescing all nested requests within an outermost critical section into a single non-nested atomic request for all potentially accessed resources within the original outermost critical section. In previous work [2], we called this abstraction *dynamic group locking*. Dynamic group locking is a form of fine-grained locking, as tasks may request only a small subset of the objects within a larger group, instead of locking the entire group, as in coarse-grained locking. Additionally, dynamic group locking has the same blocking bounds as true nested locking [4] in the RNLP [2]. In the RNLP, conflicting requests are satisfied in the order in which they were requested, though non-conflicting requests may be satisfied concurrently, unlike under coarse-grained locking.

To illustrate the RNLP, consider the example in Fig. 1, which depicts four requests denoted $\mathcal{R}_1, \ldots, \mathcal{R}_4$ contending for resources $\ell_a, \ldots, \ell_d$. Each queue is FIFO ordered, but the vertical position within the queues illustrates the blocking relationships, *i.e.*, $\mathcal{R}_3$ is blocked by $\mathcal{R}_2$, which is blocked by the satisfied request $\mathcal{R}_1$. Note that requests may be enqueued in multiple queues since multiple resources are atomically requested using dynamic group locking. 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 demonstrate why the RNLP is not contention sensitive; the worst-case blocking for $\mathcal{R}_4$ using the RNLP is not upper bounded by the worst-case contention for the resources it access, ℓ_a and ℓ_b. Instead, the worst-case blocking is a function

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

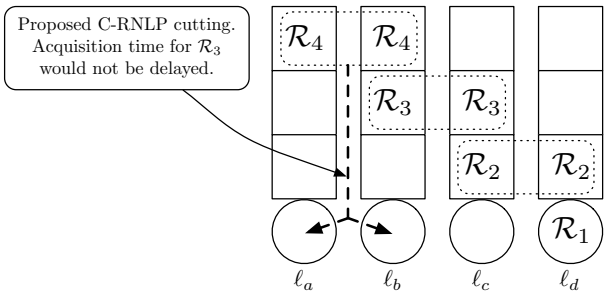

Fig. 1. Illustration of the worst-case blocking behavior of the RNLP and the proposed improvement in the C-RNLP. $\mathcal{R}_4$ is transitively blocked by $\mathcal{R}_1$ even though they do not conflict unless $\mathcal{R}_4$ is allowed to cut ahead of $\mathcal{R}_3$.

of the maximum number of queued requests, which for non-preemptive spinlocks is $O(m)$ where m is the number of processors.

If instead $\mathcal{R}_4$ had been issued before $\mathcal{R}_3$, it would have been satisfied concurrently with $\mathcal{R}_1$. Such a request sequence demonstrates the runtime parallelism afforded by the RNLP—parallelism that is not possible under coarse-grained locking. Regrettably, however, the potential runtime parallelism is often not reflected in the worst-case blocking, as an inopportune request order may serialize all requests.

To remedy this situation, the C-RNLP is augmented to allow some requests to *"cut ahead"* of other blocked requests. Specifically, a request $\mathcal{R}_i$ is allowed to cut ahead of $\mathcal{R}_j$ if doing so does not delay the worst-case satisfaction time of request $\mathcal{R}_j$, based upon the requests upon which it is already blocked. To determine if such cutting ahead is permissible, in the C-RNLP each lock request must specify its maximum critical section length so that the lock and unlock logic can determine the maximum duration of work a given request is queued behind. 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 in many real-time systems.

For example, suppose that in Fig. 1, each request has a critical section length of one time unit. Then $\mathcal{R}_3$ is blocked by both $\mathcal{R}_1$ and $\mathcal{R}_2$, or two time units total. If $\mathcal{R}_4$ cuts ahead of $\mathcal{R}_3$ as suggested in Fig. 1, it can be satisfied concurrently with either $\mathcal{R}_1$ or $\mathcal{R}_2$, and will therefore not cause any additional blocking for $\mathcal{R}_3$, assuming no request overruns its specified maximum critical section length. We note that if this assumption is violated, blocking bounds may be violated, and potentially deadlines may be missed. We claim this is the result of improper provisioning and analysis assumptions, and is therefore *not* a deficiency of the locking protocol itself.[1]

It can be shown that by allowing this type of "cutting ahead" in the wait queue, the worst-case blocking for each request is bounded by the sum of the critical section lengths for all conflicting requests. Intuitively, this is because a request can cut ahead of any requests that form a transitive blocking chain of length longer than the maximum contention for any of the requested resources. Thus, if c_i is the number of requests that conflict with $\mathcal{R}_i$, then the worst-case blocking under the C-RNLP is $O(\min(c_i, m))$, instead of $O(m)$, as in the original RNLP. Therefore, the C-RNLP has contention-sensitive worst-case blocking.

III. Experimental Results

To demonstrate the feasibility of the proposed cutting-ahead technique, we developed a preliminary spin-based prototype of the C-RNLP and measured runtime overheads. (We note that the cutting-ahead technique may also be applied in a suspension-based lock.) Our implementation is based on bitmasks;[2] each request specifies a bitmask of resources it may access, and such bitmasks can be quickly conjuncted to determine if two requests conflict. Both the lock and unlock logic, which compare these bitmasks and determine if and when requests can cut ahead, are largely contained within a critical section to an ordinary ticket lock. Furthermore, within the critical section corresponding to each lock and unlock call, the entire wait queue may be traversed while maintaining per-resource blocking metadata to determine if cutting ahead is permissible.

To measure the additional complexity and blocking associated with the lock and unlock logic for the C-RNLP, we evaluated our prototype implementation on a 4-core 2.6GHz Intel Core i7 processor. Approximately 500MB of overhead data was recorded and the 99^{th} percentile observed lock and unlock overheads were less than $0.5\,\mu$s (we report the 99^{th} percentile to filter effects of interrupts and other spurious behavior). While these overheads are certainly higher than a simple ticket lock, such overheads may be offset by the improved, contention-sensitive worst-case blocking behavior, particularly for longer critical section lengths.

IV. Conclusions and Future Work

We proposed and prototyped a contention-sensitive multiprocessor real-time locking protocol, which analytically dominates prior work. The improved blocking bounds are made possible by incorporating into the lock logic the length of each critical section. This adds complexity to the lock implementation, but our preliminary experimental results suggest this overhead is reasonable. In future work, we will conduct a full overhead-aware schedulability study of several coarse- and fine-grained locking protocols to evaluate this tradeoff between the analytical gains made possible by these fine-grained protocols, and the additional implementation complexity and therefore overheads they entail, particularly with respect to how the protocol scales to higher core counts. Additionally, we plan to implement and investigate the cutting-ahead technique in a suspension-based lock.

References

[1] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT '07*.

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

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

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

[5] A. Wieder and B. Brandenburg. On the complexity of worst-case blocking analysis of nested critical sections. In *RTSS '14*, (to appear).

[1] A C-RNLP implementation should, however, ensure lock safety, *i.e.*, mutual exclusion, even if critical section lengths are overrun.

[2] Our implementation could be extended to use resource renaming to support more resources than can be encoded in a single word.