New Approaches to Contention-Sensitive Nested Locking in Real-Time Systems*

Catherine E. Nemitz
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

Nested lock requests in multiprocessor real-time systems can be handled by only a handful of synchronization protocols. These protocols trade off overhead and blocking under varying analysis assumptions. In some systems, a fine-grained contention-sensitive protocol has significantly lower worst-case blocking compared to its non-contention-sensitive counterparts, which yields improved schedulability provided overheads are low enough. In this work, we summarize three key schemes for handling nested requests and briefly discuss existing protocols. We then propose three approaches to reduce the often interdependent overhead and blocking for a new contention-sensitive protocol.

CCS Concepts

•**Computer systems organization** → **Real-time systems;** *Embedded and cyber-physical systems; Embedded software;* •**Software and its engineering** → **Mutual exclusion; Real-time systems software; Synchronization;** *Scheduling; Process synchronization;*

Keywords

multiprocessor locking protocols, nested locks, priority-inversion blocking, real-time locking protocols, contention-sensitive blocking

1. INTRODUCTION

The progression of multicore technologies has allowed increasing numbers of real-time applications to be conceived. To allow these applications to become realities, we must maximize the use of current hardware. For example, the automotive industry is pushing toward autonomous vehicles, which require hardware with low weight, power consumption, and size that can perform complex computations on input data. In particular, sensing data such as images or video streams may be processed and modified by several tasks, requiring resource access control for that shared memory. In addition, images may be processed by some combination of GPUs, which in turn may be considered resources.

*Work supported by NSF grant CNS 1717589. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

For any real-time application, synchronization protocols are required to provide efficient resource-allocation. An efficient synchronization protocol results in schedulability improvements, which allow more effective use of the hardware.

We focus on systems in which nested resource requests are allowed. That is, a job may require multiple resources simultaneously and thus will request the resources in a nested fashion. Our goal is to guarantee contention-sensitive worst-case blocking for all jobs. A job is considered to experience contention-sensitive blocking if the amount of its blocking is dependent only on the number of other requests for the same resource. We will refine this notion in Sec. 2.

Contributions.

After covering background material, we will present three ideas about how we propose to move contention-sensitive locking protocols forward to achieve better schedulability.

2. BACKGROUND

We begin by giving an overview of the systems we are considering and how such systems are analyzed. Then we discuss three broad schemes that are used by various locking protocols to grant access to multiple resources.

2.1 Models

Task model.

We assume the classic sporadic task model. We consider a task system of n tasks denoted $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Tasks are scheduled on m processors using a job-level fixed priority scheduler. We often consider an arbitrary job J_i of task τ_i.

Resource model.

We consider systems with n_r resources. An arbitrary resource is denoted ℓ_a. Unless a locking protocol specifies otherwise, resources may be requested in any order. In this work, we focus on resources that require mutual exclusion.

Request model.

When a job J_i requires access to a resource ℓ_a, it *issues* a *request*, denoted $\mathcal{R}_{i,a}$. If J_i issues only one request, we shorten this to $\mathcal{R}_i$. We say the request is *satisfied* when the job *holds* the resource. While the request is satisfied, it is in a *critical section*. We denote the critical section length of $\mathcal{R}_i$ as L_i and the maximum critical section length of any request L_{max}. Once a request *completes*, the job *releases* the resource. If job holds a resource and then requires another, we say that the requests for these resources are *nested*.

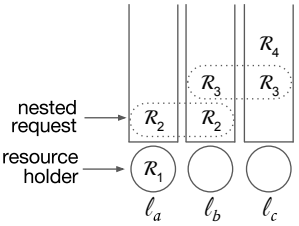

Figure 1: Example illustrating one potential ordering of requests $\mathcal{R}_1$ through $\mathcal{R}_4$. Here, $\mathcal{R}_1$ is satisfied and holds resource ℓ_a. The other requests are enqueued and waiting for access to their required resources.

(Later, we explore a technique that allows requests to be for multiple resources.) We denote the set of all resources J_i will require as D_i.

Nested requests occur in a variety of types of applications, and the depth of nested requests is typically between two and four (that is, between two and four resources are required simultaneously within the innermost critical section) [1, 3].

2.2 Analysis

We consider spin-based locking protocols and analyze such protocols on the basis of *priority-inversion blocking* (pi-blocking), which occurs when a job cannot execute because a lower priority job is holding a resource which the higher priority job requires. We consider the worst-case pi-blocking and take critical section lengths to be constant.

As mentioned above, we focus on contention-sensitive locking protocols. We denote the maximum amount of contention for a resource ℓ_a, that is, the highest possible number of active requests for that resource, as c_a. Note that this value is static for a given system. (This is in contrast to the dynamic concept of contention for ℓ_a, which is the number of active requests for that resource at a particular instance in time.) Given $\mathcal{C}_i = \max_{x \in D_i} c_x$ for a job J_i, we say that a locking protocol is contention-sensitive if the worst-case pi-blocking of any job J_i is bounded by $O(\mathcal{C}_i)$.

The key hindrance to contention-sensitive blocking is the possibility of *transitive blocking*, which occurs when a job experiences blocking because of job that it does not share any resources with. Such a situation is depicted in Fig. 1; request $\mathcal{R}_4$ conflicts only with $\mathcal{R}_3$, and yet neither of these requests are satisfied because of the chain of blocking caused by $\mathcal{R}_1$ and $\mathcal{R}_2$. This figure shows a system which allows resources to be requested together. $\mathcal{R}_1$ is satisfied and holds ℓ_a, depicted by the circle at the head of the queue. $\mathcal{R}_2$ has been enqueued for resources ℓ_a and ℓ_b. Likewise, $\mathcal{R}_3$ has been enqueued for resources ℓ_b and ℓ_c. Finally, $\mathcal{R}_4$ is enqueued and waiting for access to ℓ_c.

2.3 Handling nested requests

The following three schemes handle nested requests to provide mutually exclusive access to resources and prevent deadlock. Each method has its trade-offs, some of which affect analysis components such as the critical section length. To illustrate each approach, we use a short running example.

EXAMPLE 1. *Consider a job J_1 that requires resource ℓ_a. In addition, consider J_2 that will require access to ℓ_a and then nested access to ℓ_b. Suppose there is also a job J_3 that requires access to ℓ_b and a job J_4 that requires access to ℓ_c.*

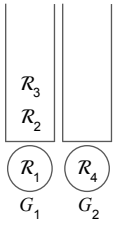

Figure 2: Example illustrating the effect of using static groups.

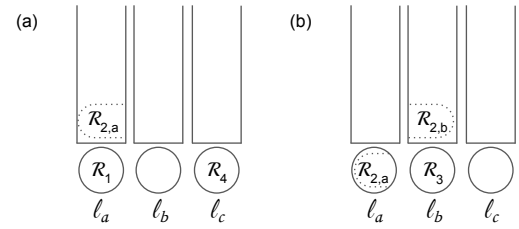

Figure 3: In (a), $\mathcal{R}_1$ holds ℓ_a, and $\mathcal{R}_{2,a}$ is waiting for ℓ_a. In (b), $\mathcal{R}_3$ holds ℓ_b and $\mathcal{R}_{2,a}$ holds ℓ_a, which allowed $\mathcal{R}_{2,b}$ to be issued for ℓ_b. The blocking of $\mathcal{R}_{2,b}$ thus contributes to the critical section length of $\mathcal{R}_{2,a}$, which in turn increases the worst-case blocking of requests for ℓ_a.

Static group locks.

This method requires static groups of resources to be formed by analyzing which resources are accessed in a nested fashion by some job. A job requiring any resource in the group must acquire the entire set. This approach allows a mutex to control access to any shared resource; each job will require only one group, so deadlock is impossible. The use of a mutex yields low overheads and inherently provides contention-sensitive blocking. However, it is important to distinguish that this notion of contention is relative to the created group of resources that a job requests. The static groups may be quite pessimistic, causing a job to contend with jobs that do not share resources but do share a group.

EXAMPLE 1 (CONT'D). *With the four jobs above, static groups $G_1 = \{\ell_a, \ell_b\}$ and $G_2 = \{\ell_c\}$ could be formed. Then jobs J_1, J_2, and J_3 will all issues requests for G_1, and job J_4 will issue a request for G_2, as shown in Fig. 2. Note that J_1 and J_3 now are considered to share a resource for the purposes of determining blocking, though they do not actually require access to the same resource. This is a small example of the increased pessimism that static group locks cause.*

Resource ordering.

In contrast to static group locks, resource ordering allows *fine-grained* locking; each job only acquires the resources it needs. In this approach, a total order on all resources is defined prior to system startup. Any nested requests must acquire resources in that order. This prevents deadlock and easily allows for a low overhead protocol within this scheme. However, this approach can easily inflate the critical section lengths beyond the given L_i, as illustrated by the following example. In fact, this inflation can be more than $m \cdot L_{max}$.

EXAMPLE 1 (CONT'D). *In this scenario, requests $\mathcal{R}_1$, $\mathcal{R}_{2,a}$, and $\mathcal{R}_4$ were released before $\mathcal{R}_3$. As shown in Fig. 3(a), $\mathcal{R}_1$ was immediately satisfied and holds ℓ_a. With the resource ordering imposed, J_2 must first issue a request for ℓ_a*

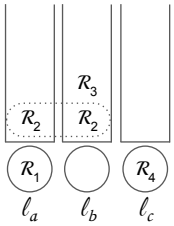

Figure 4: With DGLs, job J_2 issues a single request $\mathcal{R}_2$ for all resources it requires.

and then issue a separate request for ℓ_b, denoted $\mathcal{R}_{2,a}$ and $\mathcal{R}_{2,b}$, respectively. In Fig. 3(a), $\mathcal{R}_{2,a}$ is waiting for access to ℓ_a. $\mathcal{R}_4$ is satisfied and holds ℓ_c.

In Fig. 3(b), $\mathcal{R}_4$ has completed and $\mathcal{R}_3$ has been issued for ℓ_b and is satisfied. Later, $\mathcal{R}_1$ completed, and $\mathcal{R}_{2,a}$ became satisfied. J_2 then issued $\mathcal{R}_{2,b}$ and must wait for $\mathcal{R}_3$ to complete. It holds ℓ_a, so the time $\mathcal{R}_{2,b}$ blocks inflates the critical section length of $\mathcal{R}_{2,a}$.

Observe that resource ordering can cause huge amounts of blocking. A job requiring multiple resources may experience the following. Just before its first resource becomes available, requests may enqueue for its second resource. While it holds the first resource, it could experience the worst-case blocking for its second resource. This inflates the critical section, causing higher blocking than the expected L_i for any later request for that first resource. This build-up of blocking can be repeated for each nested resource access the job requires.

Dynamic group locks.

A third method for handling nested requests is by using *dynamic group locks* (DGLs). In this scheme, a job requiring nested resources issues a single request for all resources that it requires. This lengthens all inner critical section lengths to the length of the outermost access. Note that if a job conditionally acquires ℓ_a or ℓ_b but not both, under DGLs, it must request both resources. While holding both resources decreases potential runtime parallelism, it does not have an effect on the overall blocking; as discussed later as it pertains to static contention, we must consider the worst-case contention for each resource, and this job would be counted toward the contention of both resources regardless.

Several directions of work have been explored using the DGL scheme, as discussed in Sec. 3. One approach has moderate overheads and $O(m)$ blocking (that is, blocking bounded by the number of processors). Another approach has similar overheads and contention-sensitive blocking given certain analysis assumptions.

EXAMPLE 1 (CONT'D). *Returning to our four jobs, under DGLs, J_2 issues a single request for both ℓ_a and ℓ_b. (In order to prevent deadlock, protocols must ensure requests for multiple resources enqueue atomically into all required resource queues.) Fig. 4 shows one way in which the requests could enqueue. $\mathcal{R}_1$ is satisfied and holds ℓ_a. Thus, $\mathcal{R}_2$ is blocked. As depicted in Fig. 4, $\mathcal{R}_3$ was issued after $\mathcal{R}_2$ and must wait for access to ℓ_b. Regardless of when it is issued, $\mathcal{R}_4$ is satisfied immediately, as no other requests require ℓ_c.*

3. RELATED WORK

Standard mutex implementations, such as ticket locks and MCS locks function well with static groups locks and are inherently contention-sensitive [8].

Protocols that support fine-grained lock nesting by using resource ordering include the Multiprocessor Bandwidth Inheritance Protocol (M-BWI) [5], MrsP [4], and nested FIFO locks [2], the last of which has corresponding analysis that tractably bounds blocking.

The only protocols to use DGLs are those in the Real-time Nested Locking Protocol (RNLP) family [11, 10]. Two RNLP variants yield contention-sensitive blocking. The fast RW-RNLP provides contention-sensitive resource access only to read requests and non-nested write requests [9]. Nested write requests under the fast RW-RNLP are not contention-sensitive. Finally, the C-RNLP yields contention-sensitive blocking with the assumption that critical section lengths are the same for all resources [6].

4. NEW APPROACHES

We present three approaches to use with DGLs that we believe will be important in improving upon existing approaches toward nested lock requests. In particular, our goals are low overheads and contention-sensitive blocking for all requests, which will lead to better schedulability results.

4.1 Mutex usage

When a lock implementation requires maintenance of significant lock state, the simplest approach is to protect this state with a mutex that prevents concurrent lock calls from modifying the lock state simultaneously. This is the approach taken by the C-RNLP. While this is safe, it increases overheads by causing all requests to conflict on the lock-state mutex.

Therefore, our first approach to a new contention-sensitive locking protocol is to eliminate or reduce the usage of a lock-state mutex. Some lock structures allow this naturally or with only a slight addition of state-maintenance operations and thus overhead. For example, enqueing on multiple queues in a way that is seen as atomic simply requires that two requests for the same resources enqueue in the same order relative to each other for each resource. This is a condition that can be checked and preserved without requiring a mutex. Alternatively, even using a different mutex for each resource queue would reduce the amount of blocking that contributes to overhead, as only requests for the same resource, which already contend, would share a given mutex.

4.2 Static contention

Previous approaches to providing contention-sensitive resource access have focused on doing so with a dynamic view of contention; that is, a new request's worst-case blocking should be upper bounded by the number of active requests with overlapping resource requirements. However, this dynamic view of contention cannot be used in schedulability analysis. We must instead use the static measure of contention (the upper bound of the possible dynamic contention) in our analysis.

In light of this insight about the use of dynamic and static contention, we aim to explore the use of static contention in constructing a new contention-sensitive protocol.

A protocol designed around static contention may have less overhead; decisions regarding enqueuing for resources could be based on the static contention instead of computing the number of requests ahead of the current request. We are interested in exploring the trade-offs in such an ap-

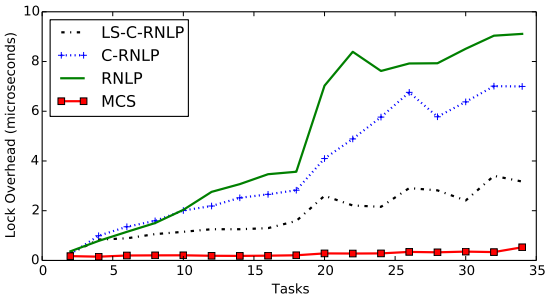

Figure 5: Lock overhead as a function of task count n for $n_r = 64$ and each job requesting four random resources from that set.

proach, which we expect would increase both schedulability and average blocking times (which could negatively impact non-real-time workloads running on the same platform as the real-time workload).

An alternate way to lower overheads could be to mix the ideas of DGLs and lock ordering. Jobs could issue requests under the DGL scheme, with all resources requested simultaneously. The lock state could then be updated with an ordered approach, in which resources are ordered by decreasing static contention. When enqueuing for these resources, requests could be required to wait until some threshold value is met before enqueuing in the subsequent queue. To clarify, a request might enqueue for ℓ_a, wait for $c_a - c_b$ time units, and then enqueue for ℓ_b, with the goal of becoming satisfied for all its resources at the same time.

4.3 Lock server

As with many ideas, this approach comes from a solution to a different problem. When some legacy applications are transfered to a multiprocessor context, a fundamental component that can slow its execution is the presence of lock requests. The idea of *remote core locking* improved performance; one core was dedicated to processing lock requests for one or more locks. This allowed the lock state and the memory locations protected by the lock to remain cache-hot. Requests were issued to this remote core by writing the lock identifier and the address of the critical section that needed to be executed to its shared cache space [7].

We propose a similar solution that we call a *lock server*. In contrast to remote core locking, a lock server maintains the lock state and executes the logic of lock and unlock calls but does not execute any of the critical section code on behalf of the request. This approach was motivated by previous work [6], in which we observed overhead trends that imply that the lock state bounces between different caches. In particular, notice the overheads presented in Fig. 5 of the RNLP and the C-RNLP. The tasks systems that generated these overheads (described in more detail below) was run on a 36 core machine with two sockets. Each task was pinned to a core, and while there were at most 18 tasks, only a single socket was used. However, once the second socket (with a separate cache) was in use, overheads drastically increased.

To do some preliminary testing of the hypothesis that a lock server would reduce overheads, we implemented the C-RNLP as a simple lock server (denoted LS-C-RNLP). Each request was issued to the lock server, which returned a location in memory on which to spin. The job then spun on its core until the value in that location in memory was set by the lock server to indicate that its request was satisfied.

We evaluated the LS-C-RNLP against the original C-RNLP, the RNLP, and the MCS on a dual-socket 18-cores-per-socket Intel Xeon E5-2699 platform. As mentioned above, each task was pinned to a core (using only a single socket when possible). These tasks repeatedly performed lock and unlock calls with a negligible critical section length in order to try to cause the worst-case overheads. Each task issued 1000 requests for a randomly chosen set of four resources of the available $n_r = 64$ resources. We report the 99th percentile of these overheads for varying numbers of tasks in the system in Fig. 5.

For the new contention-sensitive protocol we develop, we will test its overheads and the resulting schedulability of implementing it both with and without a lock server.

5. CONCLUSION

We explored three approaches to attaining a more universally applicable contention-sensitive protocol with increased schedulability results. In future work, we will explore these ideas. In particular, we are attempting to build the lock state data structures around the statically defined contention per resource. We are considering using DGLs for request issuance, but within the lock logic, employing a lock-enqueuing ordering based on decreasing static contention as a means of limiting the length of any transitive blocking chains.

6. ACKNOWLEDGMENTS

The author would like to thank Jim Anderson and Tanya Amert for their discussions and helpful feedback.

7. REFERENCES

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI 1998*.

[2] A. Biondi, B. Brandenburg, and A. Wieder. A blocking bound for nested FIFO spin locks. In *RTSS 2016*.

[3] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT 2007*.

[4] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS 2013*.

[5] D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6), 2012.

[6] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS 2015*.

[7] J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC'12*.

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

[9] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. In *RTNS 2017*.

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

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