Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts

Catherine E. Nemitz
The University of North Carolina at Chapel Hill, USA
nemitz@cs.unc.edu

Tanya Amert
The University of North Carolina at Chapel Hill, USA

James H. Anderson
The University of North Carolina at Chapel Hill, USA

Abstract

During the past decade, parallelism-related issues have been at the forefront of real-time systems research due to the advent of multicore technologies. In the coming years, such issues will loom ever larger due to increasing core counts. Having more cores means a greater potential exists for platform capacity loss when the available parallelism cannot be fully exploited. In this paper, such capacity loss is considered in the context of real-time locking protocols. In this context, lock nesting becomes a key concern as it can result in transitive blocking chains that force tasks to execute sequentially unnecessarily. Such chains can be quite long on a larger machine. Contention-sensitive real-time locking protocols have been proposed as a means of "breaking" transitive blocking chains, but such protocols tend to have high overhead due to more complicated lock/unlock logic. To ease such overhead, the usage of lock servers is considered herein. In particular, four specific lock-server paradigms are proposed and many nuances concerning their deployment are explored. Experiments are presented that show that, by executing cache hot, lock servers can enable reductions in lock/unlock overhead of up to 86%. Such reductions make contention-sensitive protocols a viable approach in practice.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Software and its engineering → Mutual exclusion, Software and its engineering → Real-time systems software, Software and its engineering → Synchronization, Software and its engineering → Process synchronization

Keywords and phrases multiprocess locking protocols, nested locks, priority-inversion blocking, reader/writer locks, real-time locking protocols

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.25

Supplement Material ECRTS Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.4.2.2`

Funding Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.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.

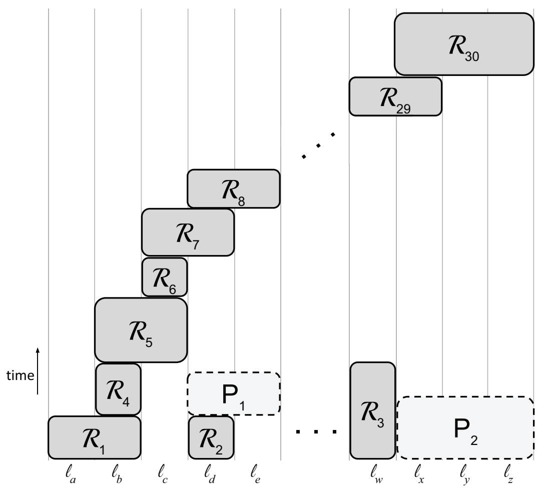


Figure 1 An illustration of transitive blocking.

1 Introduction

The evolution of multicore technologies over the past decade has shifted the focus of real-time systems research by making parallelism a paramount concern. During this time, the extent of parallelism available in commercially produced machines has steadily increased. Ten years ago, a quad-core machine was considered large. Today, machines with core counts of dozens or more are available. Looking forward, ever increasing core counts are likely to continue. The implication for real-time systems research is that resource-allocation methods shown to work well in the past may not scale as hardware platforms continue to evolve.

In this paper, we consider the issue of scale as it pertains to *real-time locking protocols*. For such a protocol to scale to large core counts, it must address intricate challenges posed by *nested lock requests*, which occur in a variety of applications [7, 13]. In particular, such requests can cause *transitive blocking chains* that cause tasks to unnecessarily execute sequentially. The potential for lost parallelism due to sequential execution increases with higher core counts. For example, assuming non-preemptive locks, if such a chain were to involve all cores of a quad-core machine, then 75% of the available processing capacity would be lost until the task at the head of the chain releases its acquired resources. On a much larger machine with, say, 32 cores, this percentage of loss would swell to nearly 97% if all cores were involved. Even if nested requests occur much less often than non-nested ones, they can still result in long blocking chains, particularly in the worst case, which would typically be considered in real-time schedulability analysis. We illustrate this point with an example chain of blocking that could occur, and thus must be accounted for in analysis.

▶ **Example 1.** Consider a set of 30 requests, some of which are shown in Fig. 1. Request $\mathcal{R}_1$ and requests $\mathcal{R}_4$ through $\mathcal{R}_{30}$ form a transitive blocking chain for the resources shown on the horizontal axis. The vertical axis shows time, with different box heights representing different critical-section lengths, and the placement along this axis representing when each request will be satisfied. Most of the blocking shown in Fig. 1 is avoidable. For example, $\mathcal{R}_8$ could move to position P_1, and $\mathcal{R}_{30}$ into P_2, greatly reducing their blocking. Note that moving $\mathcal{R}_8$ earlier reduces the blocking of later issued requests. Note also that even non-nested requests (*e.g.*, $\mathcal{R}_6$) can be transitively blocked.

Contention-sensitive real-time locking protocols guarantee the blocking time of each request is only proportional to the number of directly conflicting earlier requests by effectively "breaking" transitive blocking chains [43, 55]. Referring back to Ex. 1, enqueuing $\mathcal{R}_8$ as depicted is not contention sensitive as this queue ordering forces $\mathcal{R}_8$ to block on $\mathcal{R}_1$, $\mathcal{R}_4$, $\mathcal{R}_5$, and $\mathcal{R}_6$, none of which directly conflicts with $\mathcal{R}_8$ (they access different resources). In contrast, enqueuing $\mathcal{R}_8$ in position P_1 would ensure contention-sensitive blocking for it.

Unfortunately, the complex lock/unlock logic required to enable contention-sensitive enqueuings can result in higher overhead. To mitigate this issue, we explore herein the usage of lock servers to lessen such overhead. A *lock server* is a special process that sequentially performs all lock and unlock functions of a given protocol. The main advantage of using lock servers is that they can run cache hot (which is explained in the context of our platform in Sec. 3). The main disadvantage is the need to dedicate whole cores, or fractions of cores, to performing synchronization functions. However, on machines with high core counts, this may be a reasonable thing to do, as has been observed by others in other contexts [41, 50]. The main focus of this paper is to experimentally document the extent of overhead reduction lock servers enable when supporting contention-sensitive locking protocols. We show that such reductions can be *substantial*. We also examine various tradeoffs that arise with respect to how lock servers are deployed. We elaborate on these tradeoffs and our contributions below, after first presenting an overview of prior work to provide context.

Related work. The literature on real-time multiprocessor locking protocols is quite large [1, 2, 3, 4, 6, 8, 9, 12, 11, 10, 14, 15, 16, 18, 17, 19, 20, 21, 23, 24, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 37, 42, 43, 48, 49, 51, 53, 54, 55, 57, 58, 59, 60, 61, 62, 64, 65, 63, 67, 68, 71, 73]. Of the just-cited papers, we comment on several that are particularly relevant to our work.

A number of server-based locking protocols have been proposed previously that employ notions similar to a lock server but for a different purpose, namely, to ease the calculation of bounds on priority-inversion blocking (pi-blocking).[1] The first such protocol was the *distributed priority ceiling protocol (DPCP)* [57, 58, 59], which statically binds resources to cores and requires tasks to perform lock and unlock calls for a resource on the core assigned to that resource. Subsequently, a number of server-based protocols were proposed that follow a similar approach [21, 32, 33, 41, 42, 49, 73]. In contrast to these various server-based protocols, our focus in this paper is to preserve the blocking bounds of a given protocol while reducing its overhead. Also, our main concern is dealing with nested lock requests, which are actually precluded in most prior server-based protocols.

Only a few real-time multiprocessor locking protocols have been proposed that support nested lock requests. Among such protocols, only those in the *real-time nested locking protocol (RNLP)* family provide asymptotically optimal pi-blocking bounds [43, 55, 62, 64, 65, 63]. The RNLP family also includes the only proposed real-time locking protocols shown to be contention sensitive. We review these protocols in more detail later. Outside of the RNLP family, two other protocols have been proposed that directly support lock nesting, the *multiprocessor bandwidth inheritance protocol* [32, 33] and *MrsP* [21, 36, 73]. Neither is optimal, but both use creative techniques, like migration, to lessen blocking times.

Our work was partially inspired by work on a concept called *remote core locking* (*RCL*), which was directed at improving the performance of legacy *non-real-time* code when moving it from a uniprocessor system to a multiprocessor one [50]. In particular, RCL seeks to avoid

[1] Pi-blocking, which is more carefully considered in Sec. 2, is the primary basis on which different locking protocols are compared.

cache-line bouncing when a resource is accessed on different cores by requiring all resource accesses to occur on a designated core. In these sense, RCL is similar to the DPCP and related protocols, but the emphasis in work on RCL is to enable critical sections to run cache hot. In contrast, we want lock and unlock routines to run cache hot.

Contributions. We present an in-depth study of lock servers as a means for providing efficient implementations of contention-sensitive real-time locking protocols on large multicore machines. We restrict attention to protocols that use spinning (*i.e.*, busy waiting) to realize task blocking. We take our particular test platform as an exemplar of a "large" machine; this platform provides 36 cores split evenly across two sockets. We define lock servers in a way that does not fundamentally alter the blocking analysis of the locking protocol being supported. Thus, such analysis is not our major focus: *overhead* is.

We introduce lock servers by initially assuming a particular contention-sensitive locking protocol is to be supported that was designed assuming that all critical sections are uniformly of the same length. Using this protocol, we present four lock-server paradigms that are defined by specifying servers as either static or floating and either global or local. A *static* lock server is bound to a single core, while a *floating* one may migrate. A *global* lock server handles requests from all cores, while a *local* one handles requests from only its socket. Our test platform has two sockets, so in that context, the local case requires consideration of two lock servers, which require further arbitration. We do this by letting these lock servers alternate in phases, where the phase switching is controlled by a novel synchronization mechanism introduced here for the first time called a *phase-fair reader/reader lock*. After examining these various alternatives, we consider the ramifications of relaxing the uniformity requirement and allowing critical sections to be of different lengths.

To assess the efficacy of using lock servers, we conducted an extensive experimental evaluation on our test platform of all of the lock-server configurations mentioned above. In these experiments, the use of lock servers often reduced overhead dramatically. When supporting non-uniform critical sections, one of our lock-server paradigms reduced overhead by up to 72%. When supporting uniform critical sections, this decrease was as high as 86%.

Organization. In the rest of this paper, we provide needed background (Sec. 2), introduce static (Sec. 3) and floating (Sec. 4) lock servers assuming critical-section lengths are uniform, eliminate this uniformity assumption (Sec. 5), present our experimental evaluation (Sec. 6), and conclude (Sec. 7).

2 Background

In this section, we present relevant background material on task and resource models and provide further details concerning the locking protocols most relevant to our work.

Task Model. We consider a sporadic task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$. (We assume familiarity with the sporadic model.) These n tasks are scheduled on m processors by a job-level fixed-priority scheduler, such as one using earliest-deadline-first (EDF) priorities.

Resource Model. We focus on spin-based locking protocols invoked non-preemptively. We assume a set of n_r shared resources denoted $\mathcal{L} = \{\ell_1, \ldots, \ell_{n_r}\}$. When a job J requires access to one or more of these resources, it *issues* a request. We index requests in the order they are issued. An arbitrary request of J is denoted $\mathcal{R}_i$, and the set of resources it requires is

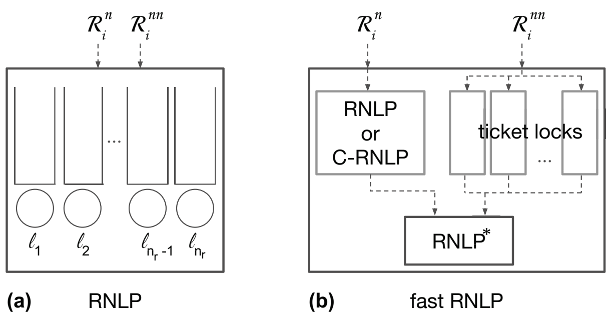


Figure 2 Important RNLP variants.

denoted D_i. $\mathcal{R}_i$ is said to be *satisfied* when J *holds* all resources in D_i. J then executes its *critical section* for L_i time units. When J *releases* all of the resources it held, $\mathcal{R}_i$ *completes*. $\mathcal{R}_i$ is considered to be *active* from the time it is issued until the time it completes.

Real-time locking protocols must have proven bounds on *priority-inversion blocking (pi-blocking)*. Pi-blocking occurs when a job cannot execute because of lower-priority work. In the context of non-preemptive spin-based protocols, a job is pi-blocked if it is spinning or if it cannot execute because some lower-priority job is executing non-preemptively.

Allowing requests to be issued for multiple resources at once as specified above provides a mechanism called a *dynamic group lock (DGL)* [63]. With DGLs, lock nesting is supported by requiring a job to issue one request for all of its needed resources, instead of issuing a separate request for each resource. The dynamic nature of DGLs allows groups of requested resources to be determined as required at runtime. This is in contrast to static group locks [9], which require resource groups to be determined offline and remain fixed.

We use DGLs to prevent deadlock. Another common approach is to define a resource ordering and require that resources be requested in that order [29, 38]. If conditional code exists, DGLs require the acquisition of resources that may not actually be needed. However, the use of DGLs and the use of a resource ordering result in the same pi-blocking bounds [62].

In stating such bounds, we assume that the maximum critical-section length, L_{max}, is constant. Additionally, we refer to the *contention* c_i experienced by a request $\mathcal{R}_i$: c_i is defined to be the number of requests that are active while $\mathcal{R}_i$ is active and that require one or more of the same resources as $\mathcal{R}_i$. A non-preemptive spin-based locking protocol is *contention sensitive* if it ensures a pi-blocking bound of $O(\min(m, c_i))$ per request.

In comparing different locking protocols, we care about overhead in addition to pi-blocking bounds. If a request $\mathcal{R}_i$ is issued at time t, and t' is the earliest time it either starts spinning or is satisfied, then the *lock overhead* $\mathcal{R}_i$ experiences is $t' - t$. Similarly, the *unlock overhead* $\mathcal{R}_i$ experiences is the total time needed to release all of its acquired resources. When we use the term *overhead* without qualification, we mean total lock plus unlock overhead.

The RNLP. The *RNLP (real-time nested locking protocol)* was the first real-time locking protocol to support nested lock requests with asymptotically optimal worst-case pi-blocking bounds [63]. The RNLP is actually a suite of protocols: both spin- and suspension-based variants exist and deadlock avoidance can be achieved by using either resource orderings or DGLs. We focus here on the spin-based DGL variant. At a high level, this variant is quite simple. As shown in Fig. 2(a), per-resource FIFO spin queues are used, and when a request for a set of resources is issued by some task, that resource is atomically enqueued onto the queues of all requested resources. Note that this atomic enqueueing requires the usage of an

Protocol	Pi-blocking	Overhead
RNLP	O(m)	moderate
C-RNLP	O(min(m,c))	high
fast RNLP (with RNLP)	O(m) nested	moderate (nested)
	O(min(m,c)) non-nested	low (non-nested)
fast RNLP (with C-RNLP)	O(min(m,c)) nested	high (nested)
	O(min(m,c)) non-nested	low (non-nested)

Figure 3 Important RNLP variants.

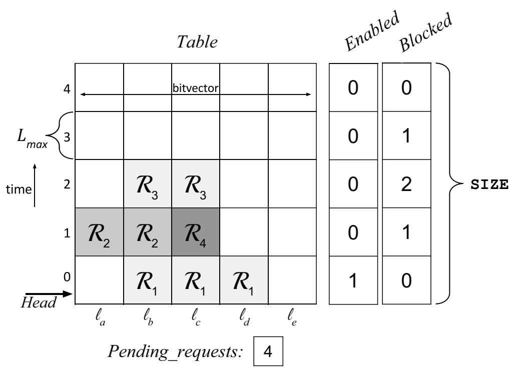


Table
Enabled
Blocked
bitvector
L_{max}
time
Head
4
3
2
1
0
$\mathcal{R}_3$ $\mathcal{R}_3$
$\mathcal{R}_2$ $\mathcal{R}_2$ $\mathcal{R}_4$
$\mathcal{R}_1$ $\mathcal{R}_1$ $\mathcal{R}_1$
ℓ_a ℓ_b ℓ_c ℓ_d ℓ_e
0 0
0 1
0 2
0 1
1 0
SIZE
Pending_requests: 4

Figure 4 Variables used in the U-C-RNLP (a similar depiction appears in [43]).

underlying mutex lock, which results in moderate lock overhead. Also, the RNLP provides no mechanism for reducing transitive blocking. For example, all of the transitive blocking shown in Fig. 1 can occur if requests are atomically enqueued as in the RNLP.

Contention-sensitive variants. The *C-RNLP (contention-sensitive RNLP)* was proposed to eliminate the long transitive blocking chains that can occur under the RNLP [43]. It does this by using a *cutting ahead* mechanism that enables contention-sensitive pi-blocking at the expense of higher overhead. A fairly detailed overview of the C-RNLP is provided later in Sec. 3 in discussing lock servers, so we refrain from providing further details now.

The *fast RNLP*[2] was proposed to achieve contention sensitivity and low overhead for non-nested requests, which are likely the common case in practice [55]. Nested requests can be made either contention sensitive at the expense of relatively high overhead for them, or non-contention sensitive, which entails much lower overhead. This functionality is achieved by employing a modular structure, as shown in Fig. 2(b). Each non-nested request acquires a simple ticket lock associated with its resource, while each nested request competes within either the RNLP (if contention sensitivity is not provided for such requests) or the C-RNLP (if it is). The RNLP* is a low-overhead version of the RNLP that must merely arbitrate between at most one non-nested request and at most one nested request per resource.

The RNLP variants just overviewed are summarized in Table 3.

3 Static Lock Servers

In this section, we consider the use of static lock servers to implement the C-RNLP. The C-RNLP is described in [43] in an abstract rule-based way. These rules can be realized in different ways in an actual implementation. For ease of exposition, we limit attention for now to the *uniform* implementation of the C-RNLP given in [43], which was designed assuming that all critical sections are the same length. Later, in Sec. 5, we will relax this assumption. In order to understand how to implement the uniform C-RNLP using lock servers, a basic understanding of it is required.

[2] Actually, the fast RNLP was proposed as the fast RW-RNLP because it provides reader/writer sharing. For simplicity, we ignore read requests in this paper and focus only on mutex sharing.

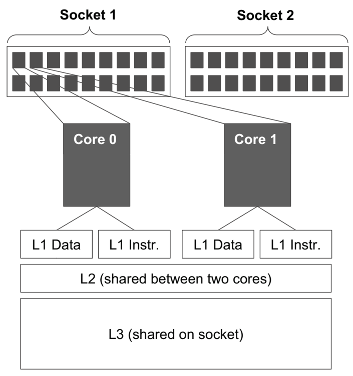


Figure 5 Test platform architecture.

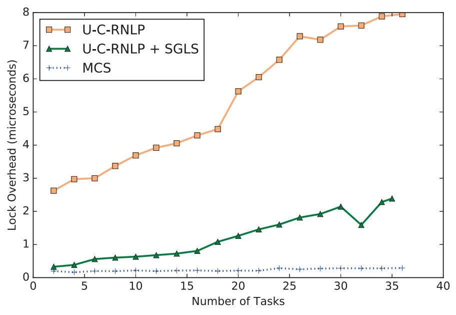


Figure 6 Lock overhead under the U-C-RNLP with and without a lock server.

Uniform C-RNLP. Under the uniform C-RNLP, denoted U-C-RNLP, each request $\mathcal{R}_i$ is satisfied within $\min(m, (c_i + 1))L_{max}$ time units, which meets the definition of contention sensitivity. This bound is realized by using a *Table* of possible satisfaction times. Each row of *Table* stores one or more bit vectors and represents a single start time, with each bit in that row representing one resource, as depicted in Fig. 4 with four requests. In the simplest implementation on a 64-bit machine, one bit vector is used, allowing 64 resources to be managed. The corresponding arrays *Enabled* and *Blocked* track which set of requests is satisfied and how many requests are immediately blocking a row in *Table*, respectively. For example, in Fig. 4, all requests in Row 0 of *Table*—here just request $\mathcal{R}_1$ for $D_1 = \{\ell_b, \ell_c, \ell_d\}$—are currently satisfied, as indicated by *Enabled*$[0] = 1$. The requests in the other rows are not currently satisfied. Requests in Row 1 are immediately waiting for one request, namely $\mathcal{R}_1$, to complete, as recorded by *Blocked*$[1] = 1$. Requests in Row 2 are waiting for two requests immediately preceding them to complete, as indicated by *Blocked*$[2] = 2$.

Platform description. In order to describe the lock-server paradigms considered in this paper more concretely, we specifically focus on our particular test platform, which is a dual-socket, 18-cores-per-socket Intel Xeon E5-2699. This platform provides significant per-socket parallelism while allowing issues on a multi-socket machine to be explored. As depicted in Fig. 5, each core has a 32KB L1 data cache and a 32KB L1 instruction cache. Pairs of cores share a unified 256KB L2 cache, and all cores on a socket share a unified 45MB L3 cache. We refer to lock state as *cache hot* if it maintains cache affinity in the lowest-level cache shared among all cores on which the server may execute.

The problem. Before delving into some of the nuances of using lock servers, let us examine the problem that they are intended to solve. Fig. 6 plots lock overhead as a function of core count (and thus number of requests) for three possibilities: the U-C-RNLP as originally presented in [43]; the same protocol but implemented using a single global lock server (denoted U-C-RNLP + SGLS); and an implementation in which all resources are coalesced under one lock using Mellor-Crummey and Scott's queue lock (denoted MCS) [52]. We take the latter as the gold standard for low overhead. We will carefully examine many such graphs in Sec. 6, so we will not bother to describe this particular one in any more detail now. However, notice the wide gap between the lock overhead for the U-C-RNLP compared to that for MCS. Our objective in this paper is to narrow this gap, hopefully considerably.

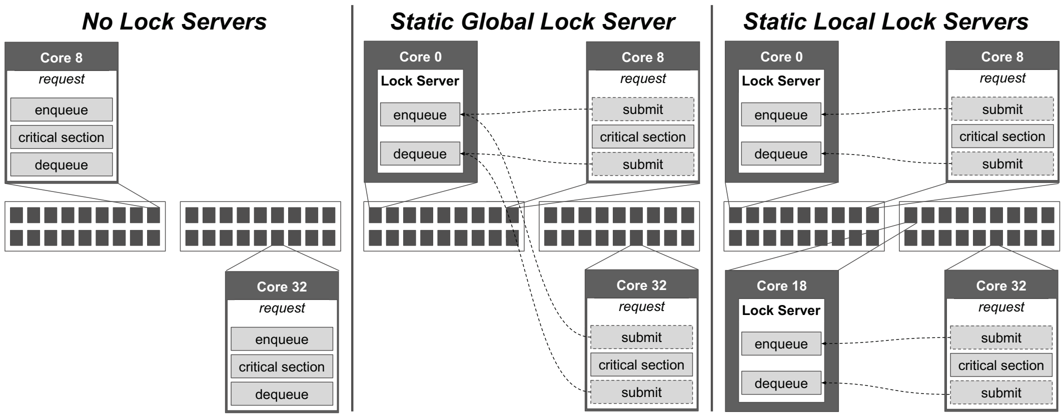


Figure 7 Three options: no lock servers (left), a single static global lock server (middle), and two per-socket static local lock servers (right).

Algorithm 1 Static Global Lock Server.

```
1: procedure SGLS(core: array of ptr to core_data)
2:     var k: unsigned int
3:     while (TRUE):
4:         if core[k]→service = LOCK_SERVICE:
5:             ^core[k]→spin_location := LS-Lock(core[k]→requested)
                                      ▷ Non-blocking LS-Lock returns spin location
6:             core[k]→service := NULL
7:         else if core[k]→service = UNLOCK_SERVICE:
8:             LS-Unlock(core[k]→requested)
9:             core[k]→service := NULL
10:        k := k + 1  mod NR_CPUS
```

Lock servers. Recall that our focus in this section is static lock servers that are pinned to dedicated cores. We consider two variations of this idea: using a *global* lock server that services requests from all cores, and using (on our platform) two *local* lock servers, each servicing requests coming from one socket. Fig. 7 depicts these two possibilities in comparison to a conventional locking protocol implementation that does not use lock servers. The potential value of lock servers can be seen by comparing the curve for U-C-RNLP + SGLS to the U-C-RNLP curve in Fig. 6. (Again, we consider graphs like this in detail later.)

3.1 A Static Global Lock Server

The simplest way to employ a lock server is to dedicate a single core to servicing all lock requests. The server uses a special version of a given protocol's Lock call, denoted LS-Lock, that updates the lock state to add a given request and then, instead of waiting by spinning to be satisfied, returns the location of a variable on which to spin. Similarly, a special version of Unlock, denoted LS-Unlock, is used. Note that these routines require no underlying mutex, as no task other than the lock server will ever access the lock state.

The behavior of the lock server is as specified in Alg. 1. It is continually active (Line 3), looping through each core (Line 10). Because our focus is non-preemptive, spin-based protocols, we know each core will have at most one active request at a given time. For a specific core k, the server checks if there is an active request that needs lock service (Line 4). If so, it uses LS-Lock to add the request to the lock state and determine the spin location for it (Line 5). In the case of the U-C-RNLP, this is the entry in *Enabled* that corresponds

Algorithm 2 New "Lock" and "Unlock" Submit Routines.

```
1: procedure SUBMIT-LOCK($c$: ptr to $core\_data$, $D$: set of resources)
2:     $c \rightarrow requested := D$
3:     $c \rightarrow service :=$ LOCK_SERVICE
4:     await $c \rightarrow service =$ NULL
5:     await $c \rightarrow spin\_location =$ TRUE
6: procedure SUBMIT-UNLOCK($c$: ptr to $core\_data$, $D$: set of resources)
7:     $c \rightarrow requested := D$
8:     $c \rightarrow service :=$ UNLOCK_SERVICE
9:     await $c \rightarrow service =$ NULL
```

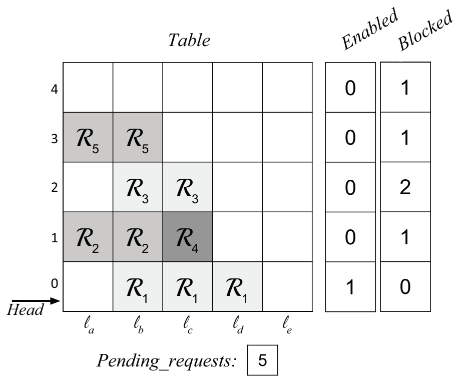


Figure 8 $\mathcal{R}_5$ is added to Row 3 of *Table*.

to the row in *Table* to which the request was added. The server then indicates that this core no longer requires service (Line 6). If instead, a request on core k requires unlock service (Line 7), the server removes it from the lock state by calling LS-UNLOCK (Line 8). It then updates the service variable indicating that core k no longer requires service (Line 9).

In the next example, we now turn our focus to the behavior of a requesting task.

▶ **Example 2.** Fig. 8 shows the result of processing a request $\mathcal{R}_5$ for $D_5 = \{\ell_a, \ell_b\}$ that is issued after requests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$ shown in Fig. 4. With a single global lock server, $\mathcal{R}_5$ executes SUBMIT-LOCK as shown in Alg. 2. It first sets *Requested* (Line 2) for its core and then indicates that it is awaiting lock service by the server (Line 3). After it has been serviced (Line 4), it spins on the location the server determined based on the other active requests (Line 5). As implied by Fig. 8, $\mathcal{R}_5$ spins on *Enabled*[3].

Using a global lock server in this manner has no impact on blocking; it simply changes the enqueuing and dequeuing portions of request processing in order to reduce overhead.

3.2 Static Local Lock Servers

In contrast to a global lock server, a local one is allowed to handle resource requests from only one socket. Our test platform has two sockets, so two lock servers are required to handle all requests; we denote them as $\mathcal{LS}_1$ and $\mathcal{LS}_2$. In this section, we assume that these lock servers are static, which means that each lock server is pinned to a specific core on its socket. The advantage of having two lock servers is that each must handle requests from only half the cores, and thus should execute with lower overhead. The disadvantage is that some arbitration mechanism is needed to mediate conflicting requests managed by the two servers. We illustrate the nature of the needed mediation with an example.

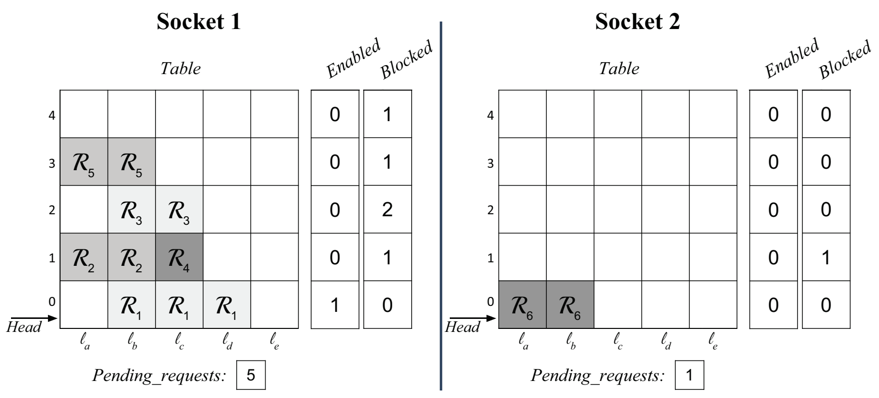


Figure 9 $\mathcal{R}_6$ is added to *Table* of Socket 2.

Algorithm 3 Static Local Lock Server.

```
1: procedure SLLS(core: array of ptr to core_data, s: socket identifier)
2:     Service lock and unlock requests like in Alg. 1, but with the following changes:
3:         Only requests from the local socket $s$ are handled
4:         Coordinate Phase with other lock server
5:         Set spin_location := TRUE for requests that are eligible to be satisfied while $Phase = s$
```

▶ **Example 2** (continued). Suppose that the requests in Fig. 8 were actually issued on Socket 1. Suppose now a request $\mathcal{R}_6$ for $D_6 = \{\ell_a, \ell_b\}$ is issued on Socket 2. This results in the two lock states shown in Fig. 9. Though $\mathcal{R}_6$ is the only request in $\mathcal{LS}_2$'s lock state, it should not be satisfied, as it conflicts with request $\mathcal{R}_1$ for resource ℓ_b. Thus, it must wait.

To mediate requests from the two lock servers, we propose to let them alternate execution in phases. In App. A, we present a phase-management protocol to coordinate these phases. In the U-C-RNLP, a natural way to define which requests belong to a certain phase is to let each row of *Table* indicate a phase. As shown in App. A, when defining and managing phases in this way, the blocking experienced by request $\mathcal{R}_i$ is at most $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units, where $c_{i,s}$ is the contention $\mathcal{R}_i$ experiences on Socket s and $L_{max,s}$ is the maximum critical-section length on Socket s. In Alg. 3, this boundary and change between phases is coordinated in Line 4 and the current phase is stored in the variable *Phase*. The coordination must ensure bounded time before a change of *Phase* when requests are waiting on the other socket. Thus, in Line 5, a request must be *able* to be satisfied (*e.g.*, it is in the active row of *Table* in the U-C-RNLP) and the phase must be set to the local socket before the request can be *marked* as satisfied by updating its spin location.

4 Floating Lock Servers

In the prior section, we implicitly assumed that static lock server(s) are to be supported by devoting full core(s) to them. While this may be reasonable on a large platform, we could instead allow other work to execute on the core(s) assigned to static lock servers(s) as long as that work executes at a lower priority. The impact lock servers have on such work could be assessed similarly to how interrupt accounting is done.

In this section, we explore a simpler alternative: floating lock servers. When using static lock servers, every request executes a spin loop for each server interaction in order to wait for a response. When using floating lock servers, the processor time wasted during these spin

Algorithm 4 Floating Global/Local Lock Server

```
 1: global var Server_exists: boolean initially FALSE

 2: procedure FLOATING-LOCK(c: ptr to core_data, D: set of resources)
 3:     var i_am_server: boolean initially FALSE
 4:     c→requested := D
 5:     c→service := LOCK_SERVICE
 6:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
 7:     if (i_am_server = FALSE):
 8:         i_am_server := WAIT-UNTIL(^(c→spin_location), TRUE)
 9:     if (i_am_server = TRUE):
10:         while (c→service ≠ NULL) or (c→spin_location ≠ TRUE):    ▷ Until satisfied, be server
11:             Perform lock server functionality
12:         Server_exists := FALSE

13: procedure FLOATING-UNLOCK(c: ptr to core_data, D: set of resources)
14:     var i_am_server: boolean initially FALSE
15:     c→requested := D
16:     c→service := UNLOCK_SERVICE
17:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
18:     if (i_am_server = TRUE):
19:         if c→service ≠ NULL:                          ▷ This request has not been serviced
20:             Perform unlock for this request
21:         Server_exists := FALSE

22: procedure WAIT-UNTIL(location: ptr, value)
23:     var t: unsigned int
24:     t := TestAndSet(&Server_exists)
25:     while (t = TRUE) and (*location ≠ value):
26:         if (Server_exists = FALSE):
27:             t := TestAndSet(&Server_exists)      ▷ TestAndSet return value of FALSE means ...
28:     return (t = FALSE)                         ▷ ... Server_exists was FALSE so I am now server
```

loops is reclaimed to execute lock-server code. This approach is tantamount to employing a *helping mechanism* [39], but unlike the traditional sense of helping, where one request may help another to complete a *critical section*, a request here performs only *lock logic* on behalf of another request. We describe the floating lock-server paradigm more fully below by first considering global servers and then local ones.

4.1 A Floating Global Lock Server

In this section, we more carefully describe the notion of a floating global lock server. Unlike static lock servers, in floating ones, request code and lock-server code are inextricably linked. Thus, we specify how a floating global lock server works via one code listing in Alg. 4.

In Alg. 4, a request in its lock call performs the same logic as it would using a static server (marking itself as requiring service, waiting for a location on which to spin, and then spinning), with intermediate checks to ensure that some request is acting as the lock server. The existence of a lock server is maintained in the global variable *Server_exists*. The helper method WAIT-UNTIL waits until a designated location holds a desired value, with the waiting terminated if the caller becomes the server (as determined in a test-and-test-and-set manner). The return value of this method indicates whether the caller is now the server.

Examining the FLOATING-LOCK routine in a bit more detail, a request first marks that it is ready to be serviced (Line 5). Then it waits to be serviced (Line 6). If it is not the lock server, then it spins on *spin_location* (Line 8). If it becomes the lock server, then it performs the lock server functionality until it is satisfied (Lines 10-11). Notice that whenever a request functions as the lock server here it would have been spinning in the global static lock server paradigm waiting for a server response.

The Floating-Unlock routine is similar, except that a request that becomes the lock server only services itself (Line 20). This is because an unlock does not involve blocking, so servicing other requests would not replace useless spinning, but would just slow the unlock.

4.2 Floating Local Lock Servers

While a floating global lock server has the benefit over static lock server(s) of not requiring dedicated core(s), it also would be expected to suffer higher overhead due to eroded cache affinity when lock state moves between sockets. Fortunately, there is a quick fix to keep lock state in cache: implement a floating *local* lock server. In this paradigm, a request can only perform the functions of the lock server for the socket from which it was issued. By restricting to a single socket, L3 cache affinity can be maintained. A floating local lock server uses the structure found in Alg. 4, but with the server logic in Lines 11 and 20 being that of a local lock server (with phase arbitration).

5 Handling Non-Uniform Requests

Recall from Sec. 3 that the C-RNLP is defined in an abstract rule-based way and that the U-C-RNLP is just one implementation of it [43]. The U-C-RNLP can be used to handle non-uniform requests by pessimistically viewing all critical sections as L_{max}. However, this changes the worst-case blocking bound of the general version from $\min(mL_{max}, c_i(L_{max} + L_i))$ to $\min(m, (c_i{+}1))L_{max}$ [43]. In this section, we discuss an alternate non-uniform implementation, denoted as the G-C-RNLP, that maintains the original bound.

The G-C-RNLP uses $|D_i|$ nodes to represent a request $\mathcal{R}_i$, one corresponding to each resource in D_i. A separate queue is maintained for each resource in the system. When $\mathcal{R}_i$ is processed, a satisfaction time is recorded for it by considering the satisfaction times for other requests and the critical-section length of each. Then, the queue for each resource in D_i is updated by inserting a node for $\mathcal{R}_i$ at a position that ensures that $\mathcal{R}_i$ will be at the head of its respective queues by its recorded satisfaction time. This protocol would likely give rise to prohibitively high overhead if the tasks themselves were to execute the queuing logic concurrently. In particular, when enqueuing a request $\mathcal{R}_i$, $|D_i|$ queues must be checked for the satisfaction times of existing requests and $|D_i|$ nodes must be inserted (sometimes in the middle of queues). However, if this protocol is implemented using lock servers,[3] then the overhead becomes quite reasonable, as we show in Sec. 6.

Using global lock servers (Secs. 3.1 and 4.1) to implement the G-C-RNLP is straightforward: we merely use the G-C-RNLP instead of the uniform C-RNLP in the LS-Lock and LS-Unlock routines. On the other hand, using local lock servers (Secs. 3.2 and 4.2) is more problematic due to the phase management such servers require. We show why by considering two examples. For the time being, we assume that a basic phase-management protocol called *Greedy Satisfaction* is used that allows only requests that can be satisfied at the start of a phase to be satisfied during that phase.

[3] Although not reflected in the pseudocode given in this paper, our lock-server implementations have been carefully honed using bit-vector operations and other techniques to improve efficiency. All of our code is publicly available online [56].

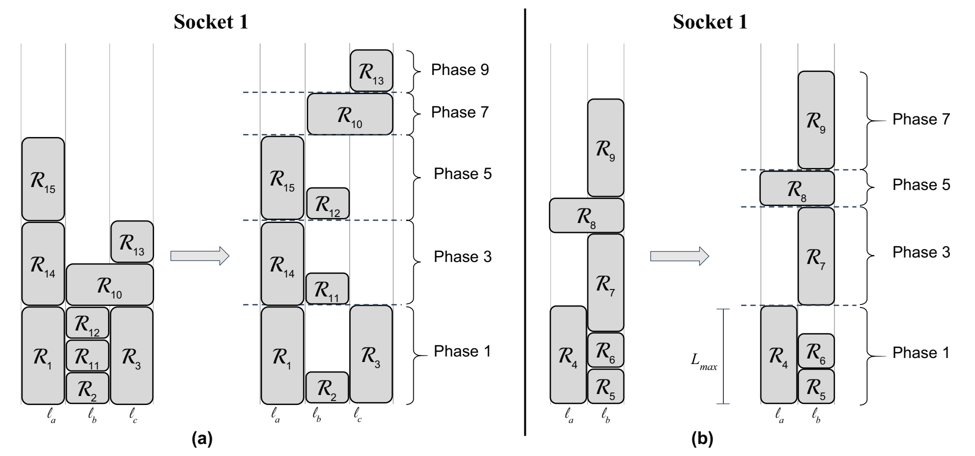


Figure 10 Scenarios with complicated phase management.

▶ **Example 3.** Consider the requests shown in Fig. 10(a), all issued on Socket 1. $\mathcal{R}_2$, $\mathcal{R}_{11}$, and $\mathcal{R}_{12}$ are "short" requests for resource ℓ_b and most of the other requests (for various resources) are longer. Under Greedy Satisfaction, requests would be satisfied in phases as shown in the right half of Fig. 10(a), with dashed lines indicating phase boundaries. Observe that, under this policy, only $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are satisfied in the first phase. $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ are satisfied later. Notice that all of the phases have odd indicies. This is because Socket 2 executes during even-indexed phases.

Ex. 3 shows that Greedy Satisfaction can unnecessarily delay requests: $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ both could have completed by the time $\mathcal{R}_3$ completed. Instead, they are moved to two later phases. We call this the *Long-Short Problem*: when requests vary in length, shorter requests can be delayed, further delaying other requests. In this example, $\mathcal{R}_{13}$ in particular is delayed substantially by requests with which it does not conflict.

Ex. 3 highlights the fact that, for some protocols, Greedy Satisfaction is inadequate. A better solution is a policy we call *Timed Satisfaction*, which allows requests that can finish within L_{max} time units to be satisfied in the same phase.

▶ **Example 4.** In Fig. 10(b), we apply Timed Satisfaction to a different set of requests on Socket 1. On the left, the requests are shown as they are ordered by the G-C-RNLP. On the right, the requests are shifted to occupy the phases the lock server would enforce. $\mathcal{R}_4$ and $\mathcal{R}_5$ are satisfied at the start of Phase 1. After $\mathcal{R}_5$ completes, $\mathcal{R}_6$ is also satisfied in this phase. However, after $\mathcal{R}_6$ completes, $\mathcal{R}_7$ cannot be satisfied, as it cannot be guaranteed to complete within L_{max} time units from the start of the phase. Therefore, $\mathcal{R}_7$ must wait until Socket 2 is allowed another phase, namely, Phase 3.

Ex. 4 illustrates another source of added blocking: $\mathcal{R}_7$ is forced to delay until the start of the next phase to be satisfied. Even if we were to increase the time window, the problem could arise again: another request could be issued that cannot complete within the window. We call this difficulty the *Overlap Problem*. A phase must end at some point to prevent the starvation of requests on the other socket. Whatever value we choose, we may have requests that would overlap a phase boundary and need to be delayed. The Overlap Problem can force a request that could otherwise be satisfied to be delayed until the current phase of its lock server completes followed by a full phase of the other lock server before being satisfied.

When considering the effect of local lock servers on blocking with the G-C-RNLP, we assume Timed Satisfaction is the phase-management policy used. (Again, the issues just discussed are unique to local servers.) As seen in Ex. 4, Timed Satisfaction is susceptible to the Overlap Problem. This is the reason why the worst-case blocking bounds presented in App. A for the G-C-RNLP are worse than those for the U-C-RNLP.

6 Evaluation

Our primary reason for exploring lock servers is to minimize overhead by keeping all lock state cache hot. For static global and static local servers, cache hot means the lock state should maintain L1 cache affinity on our platform, whereas a floating local server should tend to execute out of its socket's L3 cache. On the other hand, a floating global server will likely not be able to maintain much cache affinity if tasks execute on more than one socket.

Given these expectations, a number of questions arise. How do the different lock-server paradigms presented previously differ with respect to overhead, and do these differences match the above expectations? To what extent do lock servers lower overhead compared to not using lock servers? Are the overhead improvements enough to make contention-sensitive locking practical? How do lock servers scale with increasing core counts?

To answer these questions, we conducted an experimental study. Before covering the results revealed by our study, we first describe our experimental setup.

Experimental setup. Recall from Sec. 3 that our test platform is a dual-socket, 18-cores-per-socket platform. We used this platform to evaluate the lock-server paradigms discussed previously by conducting experiments involving tasks that repeatedly issue lock and unlock calls for random resources. We varied the number of tasks, n, number of resources, n_r, nesting depth (which defines the number of resources required for request R_i), $\mathbb{D} = |D_i|$, and critical-section length, L_i, to evaluate each parameter's effect on overhead and blocking. We define a *scenario* as an assignment of values to three of these parameters while varying the fourth. We considered the following parameter ranges: $n \in \{2, 4, ..., 36\}$, $n_r \in \{16, 32, 64\}$, $\mathbb{D} \in \{1, 2, 4, ..., 10\}$, and $L_i \in \{1\mu s, 20\mu s, 40\mu s, ..., 100\mu s\}$. In our experiments, all requests in a scenario have the same nesting depth. Unless stated otherwise, they also all have the same critical-section length L_i.

We recorded overhead and blocking times at user level, with one task pinned to each core. This setup ensures that requests execute non-preemptively. For a given scenario, we configured each task to perform 10,000 lock and unlock calls, with critical sections simulated by spinning for a duration of L_i. For task systems running on at most 18 cores, we used only the cores on one socket. When using more than 18 cores, all cores on Socket 1 were used with the remainder on Socket 2. Our workload is comprised solely of tasks making lock and unlock calls as described above. Thus, our evaluation focuses on cache affinity losses inherent to running a protocol and ignores potential evictions from other tasks; there exist techniques to keep cache affinity in some systems [5, 22, 25, 40, 46, 47, 66, 69, 70, 72].

In the graphs that follow, we plot 99th percentile measurements as worst-case values to filter out any spurious measurements caused by performing measurements at user level.[4] Across over 150 scenarios, we generated approximately 1,000 graphs. The graphs shown in this section were chosen as examples of trends seen across the entire collection of graphs. The full set can be found in an online appendix [56].

[4] This filtering does not guarantee smoothness of all curves.

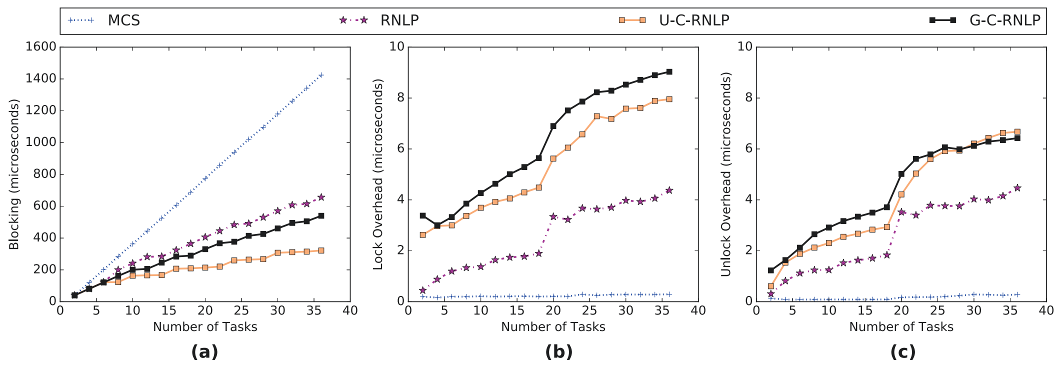


Figure 11 Blocking and lock/unlock overhead when no lock servers are used. For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

Overhead and blocking without lock servers. Before delving into results pertaining to lock-server paradigms, we examine a range of server-less implementation options. To gauge the tradeoffs involved in supporting lock nesting, we experimentally evaluated two contention-sensitive options, the U-C-RNLP and the G-C-RNLP, both implemented without lock servers, and the RNLP, which supports nesting but is not contention sensitive. As a baseline, we evaluated coalescing all resources under one MCS queue lock [52]. We conducted experiments in which these options were compared on the basis of blocking and overhead.

We now state several observations that follow from the full range of scenarios considered in these experiments. We illustrate these observations using the graphs in Fig. 11.[5] In this figure, we present lock and unlock overhead separately to demonstrate their relative scale: enqueuing takes slightly longer than dequeuing, but both operations require manipulating lock state, and thus both contribute to overhead. In later figures, we will combine lock and unlock overhead to yield one overhead graph.

▶ **Obs. 1.** *Without using lock servers, both C-RNLP variants have dramatically higher overhead than MCS.*

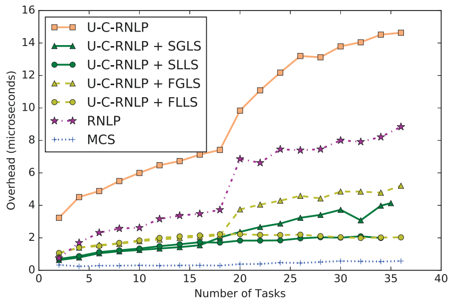
This is expected behavior, as MCS implements just a single spin queue. As shown in insets (b) and (c) of Fig. 11, the U-C-RNLP has lock and unlock overhead up to 27.4 and 23.9 times that of MCS, respectively. For the G-C-RNLP, these values are similarly high: up to 31.1 and 22.9 times, respectively.

▶ **Obs. 2.** *Compared to MCS, contention-sensitive protocols demonstrate significantly better blocking bounds as the number of requests increases.*

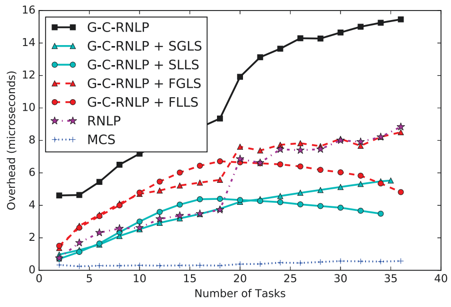
The low overhead of MCS (Obs. 1) comes at the expense of unscalable blocking. As shown in Fig. 11(a), worst-case blocking under MCS grows up to 5.3 and 2.9 times faster than that under the U-C-RNLP and G-C-RNLP, respectively.

Considering the RNLP is instructive because it provides some insights into the extra cost of providing contention sensitivity in addition to handling lock nesting. As shown in insets (b) and (c) of Fig. 11, lock and unlock overhead under the U-C-RNLP (resp., G-C-RNLP) are up to 1.8 and 2.1 (resp., 1.5 and 1.4) times that under the RNLP, respectively.

[5] In every such figure that we consider, the applicable scenario is stated in the figure's caption. Note that not all curves extend to $n = 36$. This is because up to two cores are reserved for lock servers and this number is scheme-dependent.



(a) U-C-RNLP overhead.



(b) G-C-RNLP overhead.

Figure 12 For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

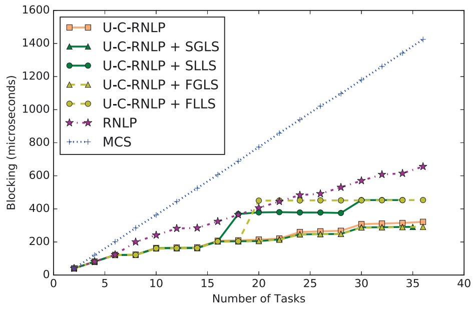


Figure 13 Worst-case blocking for the scenario in Fig. 12(a).

Applying lock servers. In Secs. 3 and 4, we presented four lock-server paradigms, each of which can be applied to any locking protocol. We conducted experiments to explore how these paradigms differ when used to implement the U-C-RNLP and the G-C-RNLP. We now state several observations that follow from the full range of scenarios considered in these experiments. We illustrate these observations using the graphs in Figs. 12 and 13. In Fig. 12(a), we compare the four possible lock-server variants of the U-C-RNLP against the baselines of MCS, the RNLP, and the U-C-RNLP without lock servers. Fig. 12(b) is similar, but is directed at the G-C-RNLP instead of the U-C-RNLP. (We abbreviate lock-server paradigms in figure captions, *e.g.*, static global lock server is SGLS.)

▶ **Obs. 3.** *Using lock server(s) results in significantly lower overhead.*

This can be seen both in Fig. 12(a) for the U-C-RNLP and in Fig. 12(b) for the G-C-RNLP. Observe that using lock server(s) usually resulted in overhead even lower than that of the RNLP. In fact, using local lock servers in this scenario reduced the overhead of the U-C-RNLP and the G-C-RNLP by up to 86% and 77%, respectively.

▶ **Obs. 4.** *When there are requests on only one socket, static lock servers result in the largest overhead reduction.*

This trend appears consistently in our results, and matches our intuition, as a static lock server can maintain L1 cache affinity. In Fig. 12, only one socket is used when $n < 18$ (it is strictly less because the lock server uses one core).

▶ **Obs. 5.** *When considering requests on two sockets, as the number of tasks increases, the overhead of local lock servers scales better than that of a global lock server.*

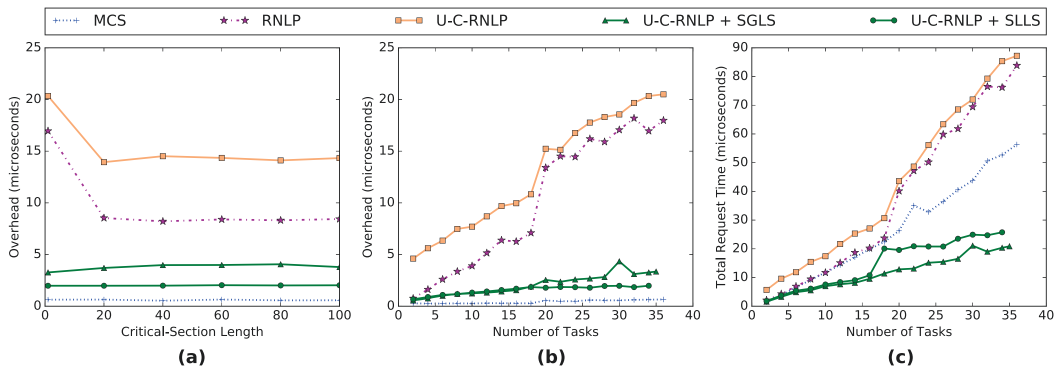


Figure 14 **(a)** Overhead as a function of critical-section length, for $n = 34, n_r = 64$, and $\mathbb{D} = 4$. **(b)** Overhead and **(c)** blocking as a function of n, for $n_r = 64, \mathbb{D} = 4$, and $L_i = 1\mu s$ for all i.

For example, in Fig. 12, the overhead of the U-C-RNLP (resp., G-C-RNLP) with floating local lock servers is up to 61% (resp., 43%) lower than with a floating global lock server.

▶ **Obs. 6.** *Floating global lock servers scale the poorest of the four lock-server paradigms.*

This observation is entirely expected and clearly evident in Fig. 12. Note that a floating global lock server still reduces overhead to be comparable to or better than the RNLP.

In Fig. 13, worst-case blocking under the U-C-RNLP is plotted for each lock-server paradigm for the same scenario presented in Fig. 12.

▶ **Obs. 7.** *Moving from one socket to two can negatively impact blocking of local lock servers.*

This observation is evident in Fig. 13. Two local lock servers are required if $n \geq 18$. The extra blocking is due to phase management and request imbalances between the two sockets. For example, for $n = 18$ there are 17 requests on Socket 1 and one request on Socket 2. The request on Socket 2 will have very low blocking, but requests on Socket 1 can experience twice as much blocking as when only one socket is in use. Without the mitigation in App. A, blocking scales poorly with increasing socket counts (*e.g.*, a four-socket platform [56]).

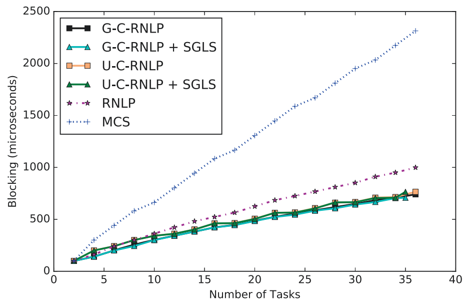
Requests with short critical sections. Inset (a) of Fig. 14 plots overhead as a function of critical-section length, while insets (b) and (c) provide data for a scenario with a short critical section of $1\mu s$. (The G-C-RNLP variants are omitted from this figure for clarity; overhead for them is higher than their U-C-RNLP counterparts but follows similar trends.) Such short critical sections result in overhead being a higher proportion of total request time (overhead plus blocking). Note that the blocking time of a request includes the overhead of any request upon which it must wait, so reducing overhead additionally reduces blocking.

▶ **Obs. 8.** *Overhead is (mostly) constant for all U-C-RNLP variants with respect to L_i.*

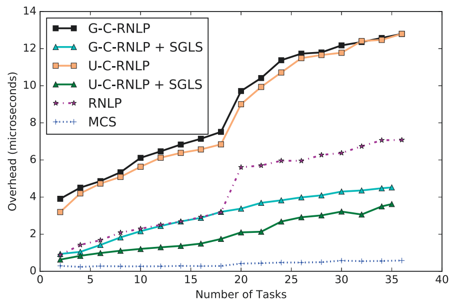
This is demonstrated in Fig. 14(a). Note that, when static lock servers are used, overhead remains low even for small L_i.

▶ **Obs. 9.** *When critical sections are short, lock servers greatly reduce the impact of overhead on total request time.*

The data in insets (b) and (c) of Fig. 14 indicates that, under the U-C-RNLP, requests with $1\mu s$ critical sections can experience worst-case overhead that is up to 23.4% of the total request time. When using a static local lock server, this is reduced to 9.6%.



(a) Blocking.



(b) Overhead.

Figure 15 For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for 75% of requests and $L_i = 100\mu s$ for the remaining 25% of requests.

	U-C-RNLP	U-C-RNLP + SGLS	U-C-RNLP + SLLS	U-C-RNLP + FGLS	G-C-RNLP + SGLS
Total Firsts	0	92	0	23	12
Total Seconds	1	26	18	70	4
Total Thirds	68	2	17	20	8
Total	69	120	35	113	24

Figure 16 Results of total request time comparison.

A case where the G-C-RNLP wins. From the results presented thus far, it is tempting to discount the G-C-RNLP entirely. In cases where all critical sections are of the same duration, the G-C-RNLP suffers worse overhead and blocking than the U-C-RNLP. We now explore scenarios in which the G-C-RNLP has very competitive worst-case blocking; this occurs when a large fraction of requests have critical-section lengths much less than L_{max}. Such a scenario is depicted in Fig. 15.

▶ **Obs. 10.** *When most requests have critical sections much shorter than L_{max}, the G-C-RNLP and U-C-RNLP have similar performance when both use a static global lock server.*

In Fig. 15, the G-C-RNLP with a static global lock server has lower blocking and only slightly higher overhead than the U-C-RNLP with the same lock-server setup.

Overall winner. Judging the lock-server paradigms should be done with a specific workload, but to make a general summary, we determined the "best" paradigm to the extent possible in our experimental framework as follows. For each considered scenario,[6] we calculated a single "total request time" score (blocking plus overhead) for each protocol variant by approximating the area under its curve using a midpoint Riemann sum. We then ranked the protocol variants for that scenario. Fig. 16 gives the total number of first-, second-, and third-place finishes for each protocol variant. The U-C-RNLP with a static global lock server was the overall winner. However, our experimental setup mostly generates scenarios in which critical sections are uniform, which tends to make the G-C-RNLP variants less competitive. Still, these results show there is value in using lock servers.

[6] We filtered out scenarios with $\mathbb{D} \in \{8, 10\}$, as they require nearly coalescing all resources under a single lock, which has non-contention-sensitive blocking.

7 Conclusion

In this paper, we have considered for the first time the use of lock servers on large multicore platforms to lessen overhead associated with contention-sensitive real-time locking protocols, without modifying the associated pi-blocking bounds. We proposed four specific lock-server paradigms and presented an experimental study in which the overhead reductions enabled by these paradigms was assessed. This study showed that such reductions can be dramatic. For example, the paradigm that generally performed best, static global lock servers, typically exhibited overhead reductions in the range 25%–75% compared to not using lock servers.

This paper is certainly not the last word on lock servers. Indeed, we hope that our work sparks further interest by others in this topic and more broadly raises an appreciation for investigating scalability issues affecting real-time resource-allocation methods as core counts continue to climb. With respect to lock servers themselves, a number of avenues for further research come to mind. First, while we have limited attention to spin-based locking protocols, the very notion of a lock server lends itself to an operating-system-based implementation. In that setting, suspension-based protocols warrant detailed consideration. Second, we have focused on one particular large multicore platform as an exemplar. Other platforms, including manycore platforms with different interconnects, warrant further study. Third, it would be interesting to apply the ideas in this paper to support transactions in a real-time database. In fact, a contention-sensitive real-time locking protocol together with lock server(s) can be thought of as a lock-based variant of software transactional memory that targets real-time applications. Fourth, we have focused herein on the extent to which lock servers can lower overhead. In the future, we will assess the *schedulability*-related impacts of different lock-server deployments, which will require investigating lock server behavior in the context of more complex workloads and exploring task balancing among lock servers. Finally, in a hard real-time system, it might be necessary to *provably* ensure that lock servers always execute in cache. Such assurances could be provided by integrating lock servers with cache-isolation techniques explored elsewhere [5, 22, 25, 40, 46, 47, 66, 69, 70, 72].

References

1 S. Afshar, M. Behnam, R. Bril, and T. Nolte. Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In *SIES '14*, pages 41–51. IEEE, 2014. `doi:10.1109/SIES.2014.6871185`.

2 S. Afshar, M. Behnam, R. Bril, and T. Nolte. An optimal spin-lock priority assignment algorithm for real-time multi-core systems. In *RTCSA '17*, pages 1–11. IEEE Computer Society, 2017. `doi:10.1109/RTCSA.2017.8046310`.

3 S. Afshar, M. Behnam, R. Bril, and T. Nolte. Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems*, 4(2), 2017. `doi:10.4230/LITES-v004-i002-a003`.

4 S. Afshar, F. Nemati, and T. Nolte. Towards resource sharing under multiprocessor semi-partitioned scheduling. In *SIES '12*, pages 315–318. IEEE, 2012. `doi:10.1109/SIES.2012.6356605`.

5 S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS '14*, pages 15–26. IEEE Computer Society, 2014. `doi:10.1109/ECRTS.2014.11`.

6 B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.

7 D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI '98*, pages 258–268, 1998. `doi:10.1145/277650.277734`.

8 A. Biondi and B. Brandenburg. Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors. In *ECRTS '16*, pages 39–49. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.30.

9 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56. IEEE Computer Society, 2007. doi:10.1109/RTCSA.2007.8.

10 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

11 B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS '13*, pages 141–152. IEEE Computer Society, 2013. doi:10.1109/RTAS.2013.6531087.

12 B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS '14*, pages 61–71. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.26.

13 B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT '07*, 2007.

14 B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *OPODIS '08*, pages 105–124, 2008. doi:10.1007/978-3-540-92221-6_9.

15 B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *RTCSA '08*, pages 185–194, 2008. doi:10.1109/RTCSA.2008.13.

16 B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, pages 49–60. IEEE Computer Society, 2010. doi:10.1109/RTSS.2010.17.

17 B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1), 2010.

18 B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*, pages 69–78. ACM, 2011. doi:10.1145/2038642.2038655.

19 B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2013.

20 B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS '08*, pages 342–353. IEEE Computer Society, 2008. doi:10.1109/RTAS.2008.27.

21 A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS '13*, pages 282–291. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.37.

22 M. Campoy, A.P. Ivars, and J.V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop '01*, 2001.

23 Y. Chang, R. Davis, and A. Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *RTNS '10*, 2010.

24 C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Dept. of Computer Science, Univ. of Maryland. Technical report, CS-TR-3252, April, 1994.

25 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS '15*, pages 305–316. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.36.

26 T. Craig. Queuing spin lock algorithms to support timing predictability. In *RTSS '93*, pages 148–157. IEEE Computer Society, 1993. doi:10.1109/REAL.1993.393505.

27 R. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS '06*, pages 257–270. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.42.

28 U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *ECRTS '06*, pages 75–84. IEEE Computer Society, 2006. `doi:10.1109/ECRTS.2006.10`.

29 E. Dijkstra. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

30 A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS '09*, pages 377–386, 2009. `doi:10.1109/RTSS.2009.37`.

31 G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.

32 D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS '10*, pages 90–99, 2010. `doi:10.1109/ECRTS.2010.19`.

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

34 P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *RTAS '03*, page 189, 2003. `doi:10.1109/RTTAS.2003.1203051`.

35 P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*, pages 73–83. IEEE Computer Society, 2001. `doi:10.1109/REAL.2001.990598`.

36 J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies '17*, volume 10300 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2017. `doi:10.1007/978-3-319-60588-3_5`.

37 J. Han, D. Zhu, X. Wu, L. Yang, and H. Jin. Multiprocessor real-time systems with shared resources: Utilization bound and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

38 J. Havender. Avoiding deadlock in multitasking systems. *IBM systems journal*, 7(2):74–84, 1968.

39 M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

40 J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS '11*, pages 23–32. IEEE Computer Society, 2011. `doi:10.1109/ECRTS.2011.11`.

41 P. Hsiu, D. Lee, and T. Kuo. Task synchronization and allocation for many-core real-time systems. In *EMSOFT '11*, pages 79–88. ACM, 2011. `doi:10.1145/2038642.2038656`.

42 W. Huang, M. Yang, and J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *RTSS '16*, pages 111–122. IEEE Computer Society, 2016. `doi:10.1109/RTSS.2016.020`.

43 C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS '15*, pages 3–12. ACM, 2015. `doi:10.1145/2834848.2834874`.

44 Y. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

45 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99*, pages 23–32, 1999. `doi:10.1145/301308.301319`.

46 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS '13*, pages 80–89. IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.19`.

47 D. Kirk and J. Strosnider. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *RTSS '90*, pages 322–330. IEEE Computer Society, 1990. doi:10.1109/REAL.1990.128764.

48 L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems (TOCS)*, 15(1):3–40, 1997.

49 K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS '09*, pages 469–478. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.51.

50 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*, pages 65–76. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

51 G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *ECRTS '11*, pages 262–271. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.32.

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

53 F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *ECRTS '11*, pages 251–261. IEEE Computer Society, 2011. doi:10.1109/ECRTS.2011.31.

54 F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *OPODIS '10*, volume 6490 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2010. doi:10.1007/978-3-642-17653-1_20.

55 C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. In *RTNS '17*, pages 38–47. ACM, 2017. doi:10.1145/3139258.3139262.

56 C. Nemitz, T. Amert, and J. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts (extended version), 2018. URL: http://www.cs.unc.edu/~anderson/papers.html.

57 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS '90*, pages 116–123. IEEE Computer Society, 1990. URL: https://doi.org/10.1109/ICDCS.1990.89257, doi:10.1109/ICDCS.1990.89257.

58 R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

59 R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS '88*, pages 259–269, 1988. doi:10.1109/REAL.1988.51121.

60 H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *RTCSA '95*, pages 160–167, 1995. URL: https://doi.org/10.1109/RTCSA.1995.528766, doi:10.1109/RTCSA.1995.528766.

61 C. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *ISPAN '96*, pages 70–76. IEEE Computer Society, 1996. doi:10.1109/ISPAN.1996.508963.

62 B. Ward. *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems.* PhD thesis, University of North Carolina, Chapel Hill, NC, 2016.

63 B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*, pages 223–232. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.17.

64 B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *RTNS '13*, pages 67–76. ACM, 2013. doi:10.1145/2516821.2516843.

65 B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *IPDPS '14*, pages 177–186. IEEE Computer Society, 2014. `doi:10.1109/IPDPS.2014.29`.

66 B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*, pages 157–167. IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.26`.

67 A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS '13*, pages 45–56. IEEE Computer Society, 2013. `doi:10.1109/RTSS.2013.13`.

68 A. Wieder and B. Brandenburg. On the complexity of worst-case blocking analysis of nested critical sections. In *RTSS '14*, pages 106–117. IEEE Computer Society, 2014. `doi:10.1109/RTSS.2014.34`.

69 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*, 2016.

70 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *RTAS '17*, pages 211–222, 2017. `doi:10.1109/RTAS.2017.15`.

71 M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *RTSS '15*, pages 1–12. IEEE Computer Society, 2015. `doi:10.1109/RTSS.2015.8`.

72 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS '14*, pages 155–166. IEEE Computer Society, 2014. `doi:10.1109/RTAS.2014.6925999`.

73 S. Zhao, J. Garrido, A. Burns, and A. Wellings. New schedulability analysis for MrsP. In *RTCSA '17*, pages 1–10. IEEE Computer Society, 2017. `doi:10.1109/RTCSA.2017.8046311`.

A Local Lock Server Phase Management and Blocking Bounds

In this appendix, we provide additional details concerning the phase-management protocol needed for the local lock servers described in Secs. 3.2 and 4.2. Such a server must determine which requests will execute in each of its phases in addition to managing phase changes.

Request selection. We restrict phases on Socket s to execute for at most the maximum critical-section length on that socket, denoted $L_{max,s}$. For the U-C-RNLP, the requests in a phase are determined by selecting the row in *Table* pointed to by *Head*. For the G-C-RNLP, Timed Satisfaction (recall Sec. 4.2) is used instead.

Phase coordination. Because all requests that can be satisfied simultaneously under C-RNLP rules can run concurrently relative to each other, they may be processed like read requests. With this in mind, the synchronization mechanism we need can be obtained by building on the idea of a phase-fair reader/writer lock [17]. Such a lock supports two kinds of requests, *reads* and *writes*, which execute in phases that alternate if both kinds of requests are present, where any number of reads can occur during a read phase but only one write during a write phase. The synchronization mechanism we desire similarly needs to support two kinds of requests that execute in alternating phases, but in our case, any number of requests can execute in a given phase. That is, we need a *reader/reader* lock. To our knowledge, such locks have not been studied in the context of real-time systems, so we present a new phase-fair reader/reader locking algorithm with corresponding blocking bounds in an online appendix [56]. (The phase-fair reader/reader problem is similar to the group mutual exclusion problem [44, 45] except that we require $O(1)$ pi-blocking bounds.)

Using this reader/reader lock, it is straightforward to support phase management in a way that satisfies the following general properties.

- Each lock server is either *active* or *passive* and at most one lock server is active at any time. A maximal interval of time when a lock server is active is called a *phase*.
- A request can be satisfied only if its lock server is active and if it can be satisfied under the variant of the C-RNLP being used by that server.
- A passive lock server with unsatisfied requests becomes active within L_{max} time units.
- All requests satisfied in a phase finish by the end of that phase.

Based on these properties, we prove the worst-case acquisition-delay bounds stated below in an online appendix [56]. In stating these bounds, recall that $\mathcal{LS}_s$ denotes the local lock server on Socket s. Also, we denote the contention a request $\mathcal{R}_i$ experiences on Socket s as $c_{i,s}$. We call such a request *entitled* if it could be satisfied under the C-RNLP.

▶ **Theorem 5.** *A request $\mathcal{R}_i$ on socket s that is serviced by a local lock server running the U-C-RNLP will be satisfied within $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units.*

▶ **Theorem 6.** *A request $\mathcal{R}_i$ on Socket 1 (resp., Socket 2) that is serviced by a local lock server running the G-C-RNLP will be satisfied within $c_{i,1}(3L_{max,1}+2L_{max,2}+L_i)$ (resp., $c_{i,2}(2L_{max,1}+3L_{max,2}+L_i)$) time units.*

These bounds have implications regarding how to partition a workload under schedulers that assign tasks to execute on specific cores or clusters of cores. We illustrate this point in the context of the U-C-RNLP.

To begin, suppose that the requests for each resource can be evenly split between sockets such that $L_{max,1} = L_{max,2} = L_{max}$. Then, $c_{i,1} = c_{i,2} = \frac{1}{2}c_i$, and the blocking bound in Theorem 5 reduces to $(\frac{1}{2}c_i + 1)(2L_{max}) = (c_i + 2)L_{max}$, which is only one critical-section length longer than that for the original protocol.

While splitting contention evenly like this may be desirable, a system designer could instead choose to assign tasks so as to decrease $c_{i,1}$ at the expense of $c_{i,2}$, which may be a more effective strategy if critical sections of different lengths exist. To see this, suppose that a fraction α of all requests have critical sections of at most $\beta \cdot L_{max}$ time units, where $0 < \beta \leq 1$. If tasks can be assigned so that these shorter requests are all issued from Socket 1 and all others from Socket 2, then the bound from Theorem 5 becomes $(\alpha c_i + 1)(\beta L_{max} + L_{max}) = (\alpha c_i + 1)(\beta + 1)L_{max}$ when applied to Socket 1, and $((1-\alpha)c_i + 1)(\beta L_{max} + L_{max}) = ((1-\alpha)c_i + 1)(\beta+1)L_{max}$ for Socket 2. Depending on the system, such a task assignment could lower the bounds applicable to all requests, as seen in the following example.

▶ **Example 7.** Suppose $c_i = 10$, $L_{max} = 100\mu$s, $\alpha = \frac{1}{5}$, and $\beta = \frac{1}{10}$. With the partitioning of requests described above, the bound on Socket 1 is $(\frac{1}{5}\cdot 10 + 1)(\frac{1}{10}\cdot 100 + 100)\mu s = 330\mu s$, and the bound on Socket 2 is $990\mu s$, both of which are lower than the bound of $(c_i+1)L_{max} = (10+1)100 = 1100\mu s$ for a server-less system (recall the U-C-RNLP discussion in Sec. 3).

Note that the improvement in the above example holds for both sockets, not just the one with lower critical-section lengths.