A Preliminary Examination of Schedulability under Lock Servers*

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

ABSTRACT

Allowing nested resource access in a real-time system introduces several challenges. Addressing these challenges within a synchronization protocol often leads to significant protocol overhead. Recently, a protocol-independent method was developed that significantly reduces this overhead; lock servers manage the execution of complex protocols, largely independently from the tasks that require resource access. However, some lock server configurations change the blocking caused by the underlying protocol. This leaves an as-of-yet unanswered question: how does the use of lock servers impact schedulability? I present a preliminary examination of that question and briefly explore how the assignment of tasks to lock servers can impact schedulability.

Keywords

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

1. INTRODUCTION

Real-time systems, those which require timing guarantees as a component of system correctness, require an efficient synchronization protocol to enable safe resource sharing while meeting deadlines. A particular challenge to protocol efficiency is the presence of nested resource requests, which occur in real-world systems [1, 3, 5] when multiple resources must be held simultaneously.

Synchronization protocols are necessary to ensure safe resource sharing, but contribute two fundamental types of delay to task execution. *Blocking* occurs when a task must wait due to the protocol managing access to resources. Blocking varies between protocols based on how each protocol orders tasks to grant resource access. The other delay introduced by locking protocols is *overhead*—the time required to execute the protocol logic and determine which task(s) may be granted access to resources at each time.

Until recently, approaches taken by locking protocols to handle nested resource access either (i) artificially limit nesting [8, 14], (ii) may cause significant blocking [6, 7, 9, 14, 15], or (iii) cause significant overhead [10]. A recently developed method [12] allows the reduction of protocol overhead and, depending on the configuration, may change the computation of worst-case blocking. This reveals a need—to fully examine the tradeoffs of overhead and blocking under this new approach *in the context of schedulability.*

In light of this, I take the uniform variant of the contention-sensitive real-time nested locking protocol (U-C-RNLP), as a case study and explore the impacts of the various lock server configurations on schedulability. While one lock server configuration tends to outperform all others in this preliminary study, the results also show that with better accounting techniques and methods of task allocation, the other configurations may lead to higher schedulability in more scenarios.

Organization. I begin by giving necessary background on the system model and on related work Sec. 2. Next, I present preliminary schedulability results Sec. 3. Finally, I conclude and present directions for future work in Sec. 4.

2. BACKGROUND AND PRIOR WORK

I begin by describing the system model before giving relevant details about the different lock server configurations and the functionality of the U-C-RNLP.

System Model. In this paper, I assume the standard sporadic task model, in which an arbitrary task is denoted τ_i. As described in more detail in Sec. 2, clustered scheduling is required for some lock server configurations, so I assume Clustered Earliest-Deadline-First (C-EDF) scheduling.

When a task τ_i requires access to one or more resources, it issues a request $\mathcal{R}_i$. I focus on a spin-based locking protocol, in which τ_i busy waits until it is granted access to its resources, after which it executes non-preemptively until completing at most L_i time units later. The maximum critical-section length is denoted L_{max}.

Lock Servers. Lock servers [12] build on an idea fundamental to remote core locking [11] by isolating the execution of the lock logic to a few processes to better utilize the cache(s). This is the mechanism that allows such a drastic reduction in overhead. There are four fundamental lock server types, which are distinguished by *locality* and *mobility.*

Let us begin by assuming a single lock server and exploring the two types of mobility. A *static* lock server is pinned to a single core. In contrast, a *floating* lock server moves between cores. More specifically, when a task is busy-waiting for access to its required resource(s), it can instead assume the role of the lock server; until it is satisfied, it cannot continue its execution, but it can execute the lock logic on behalf of

*Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, 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.

Protocol	Worst-Case Acquisition Delay	Overhead (μs) on Platform 1	Overhead (μs) on Platform 2
U-C-RNLP	$(c_i + 1) \cdot L_{max}$	23.5	29.2
U-C-RNLP + SGLS	$(c_i + 1) \cdot L_{max}$	13.5	8.0
U-C-RNLP + SLLS	$(c_{i,s} + 1)(L_{max,1} + L_{max,2})$	8.7	2.8
U-C-RNLP + FGLS	$(c_i + 1) \cdot L_{max}$	11.5	9.1
U-C-RNLP + FLLS	$(c_{i,s} + 1)(L_{max,1} + L_{max,2})$	10.8	3.1

Table 1: Blocking bounds and overhead of each lock server configuration with the U-C-RNLP.

other tasks. In general, a static lock server will result in lower overhead than a floating lock server, as eliminating mobility can allow lock state to remain in the L1 cache.

Now let us consider the locality options. A *global* lock server executes the protocol logic on behalf of all tasks, while a *local* lock server handles only a subset of all tasks. The local lock server configurations originally presented [12] designate one lock server per CPU socket. A set of local lock servers tend to have lower overhead than a global lock server, as some level of cache affinity can be maintained. Combining both distinctions, a set of static local lock servers will have the lowest overhead of the four possible configurations. However, the use of local lock servers comes with a tradeoff: an additional synchronization mechanism is required in order to ensure tasks managed by different lock servers execute safely. This changes the worst-case blocking for the protocol used, which I describe in more detail after presenting the basic functionality of the locking protocol I consider.

U-C-RNLP. Though lock servers provide a means for reducing overhead that is protocol independent, they require use of some protocol. For this work, I focus on the Uniform C-RNLP (U-C-RNLP) variant of the C-RNLP [10].

The U-C-RNLP maintains a table of waiting and satisfied requests that indicates when each request will be satisfied. When a new request is issued, it is added to the first ("earliest") row in which there are no requests for an overlapping set of resources. Entire rows are satisfied concurrently; when a request completes, it is removed from the table and, if it was the last request in its row, it indicates that any requests in the subsequent row may become satisfied immediately.

The worst-case blocking for a request $\mathcal{R}_i$ handled by the U-C-RNLP is thus dependent on the number of other requests that conflict (require one or more of the same resources) with $\mathcal{R}_i$. This is the *contention* that $\mathcal{R}_i$ may experience and is denoted c_i. This leads to the bound of the U-C-RNLP (without any lock server) shown in Table 1. Essentially, there are at most c_i+1 rows of requests that will be satisfied before $\mathcal{R}_i$ is satisfied, and it may take up to L_{max} time units for all requests of a given row to complete.

When lock servers are used, the worst-case acquisition delay depends on the configuration. With a single lock server, as with the Floating Global Lock Server (FGLS) or the Static Global Lock Server (SGLS), the blocking remains unchanged from that of the basic U-C-RNLP protocol without lock servers. However, with multiple lock servers, like with the Floating Local (FLLS) and Static Local (SLLS) configurations, additional coordination is required between the different lock servers before resource access may be granted.

Refining the notion of a maximum critical-section length per local lock server allows a tighter bound to be computed. The maximum critical-section length of any request managed by Lock Server 1 (resp., Lock Server 2) is denoted $L_{max,1}$ (resp., $L_{max,2}$). Additionally, I refine the contention of a request $\mathcal{R}_i$ to specify the number of contending requests also served by Lock Server s, which is denoted $c_{i,s}$. The bounds on acquisition delay for the U-C-RNLP with each lock server configuration is shown in Table 1.

3. SCHEDULABILITY ANALYSIS

In this section, I describe my evaluation methodology and present the results. This evaluation is centered around the question of schedulability, but synchronization protocol overhead and blocking must first be accounted for. I computed both of these components and then incorporated them into the open-source schedulability toolkit SchedCAT [2].

3.1 Overhead

Synchronization protocol overhead is platform dependent. To accurately compare the tradeoffs of different lock server configurations, overhead must be computed on any test platform. For this preliminary investigation, I explore two platforms. Platform 1 is a dual-socket, 8-cores-per-socket Intel CPU platform. The second platform I consider, Platform 2, is a dual-socket, 18-cores-per-socket Intel CPU platform. Both platforms have three cache levels, with the lowest level shared across an entire socket, but not between sockets. Based on this two-socket structure, for the local lock server configurations, I use one local server per socket.

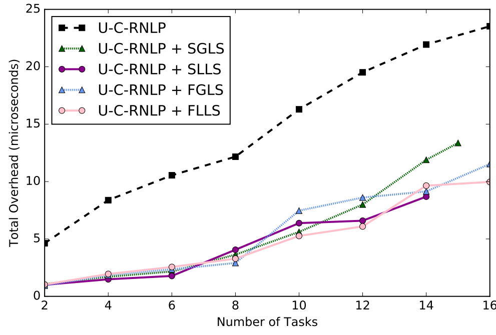
In order to measure the overhead of each configuration, I used the methodology described in [12]. I varied critical-section lengths, testing $L_i \in \{1, 15, 100\}\mu$s with nesting depth of either 2 or 4. The highest overhead for each configuration tended to be for nesting depth of 2 and $L_i = 1\mu$s, which is depicted in Fig. 1 for both platforms. The highest overhead values I measured are recorded in Table 1; these are the values I incorporated into the schedulability study by inflating the execution time of a task.

3.2 Blocking

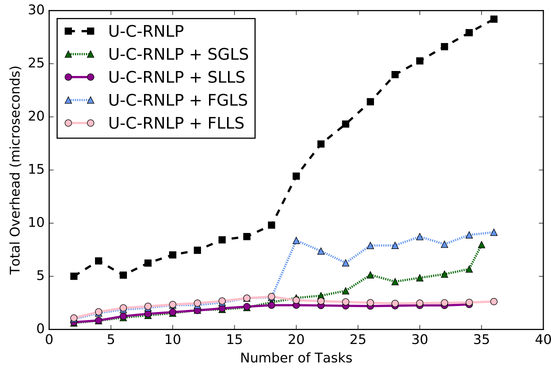
To compute blocking, I use the bounds presented in Table 1. In the implementation of these computations in the SchedCAT framework, I also made a refinement to tighten the analysis, by applying a period-based constraint [13], as it may not always take L_{max} for each row of requests to complete. Instead, I use a set of the largest critical-section lengths; I account for the number of times each critical section could delay $\mathcal{R}_i$ based on the relative periods of the tasks. Once I have computed the blocking a task may encounter, I inflate its execution time by this amount.

3.3 Schedulability

The taskset under analysis is divided into clusters by a Worst-Fit bin packing heuristic, which considers the utilization of a task to be its "weight" and the bin size to be $U = 1$.



(a) Platform 1



(b) Platform 2

Figure 1: Total protocol overhead.

Category	Name	Value
Critical-Section Length (μs)	Short	[1,15]
	Bimodal	[1,15] or [15,100]
	Moderate	[15,100]
Period (ms)	Short	[3,33]
	Moderate	[10,100]

Table 2: Named parameter distributions.

After accounting for the blocking and overhead a task may incur, I apply Baruah's G-EDF schedulability test [4] to each cluster individually. If all clusters pass this test, the taskset is deemed schedulable.

I examined a range of scenarios; the parameter values are summarized in Tables 2 and 3. I focused on nested requests; non-nested requests can be handled efficiently by other means [13]. Also, I assumed each task issues at most one request. For each value of system utilization considered, 100 tasksets were examined. To analyze the static lock server configurations, I assumed an entire core was dedicated to each server. I reevaluate this pessimistic decision below.

Platform 1. I begin with a series of observations from the 72 scenarios explored on Platform 1. Fig. 2 represents one such scenario, in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

Obs. 1. *The schedulability of the U-C-RNLP with the FGLS is always as good or better than that of the U-C-RNLP with no lock server.*

Category	Options
Task Utilization	[0.1,0.4]
Period	Short, Long
Percentage Issuing Requests	5%,10%,20%, 50%, 80%, 100%
Critical-Section Length	Short, Bimodal, Moderate
Number of Resources	64
Nested Probability	1.0
Nesting Depth	2, 4

Table 3: Schedulability study parameter choices. For each range of values, a value is selected uniformly at random.

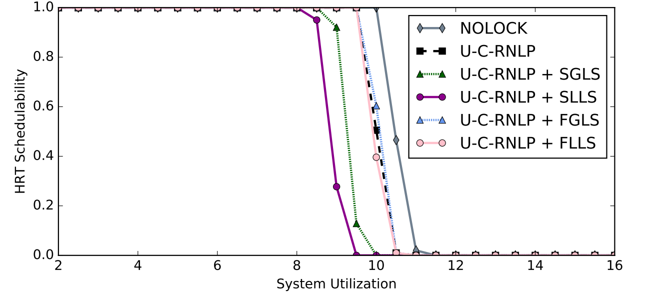


Figure 2: Schedulability on Platform 1 under a scenario in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

This is illustrated in Fig. 2, and is as expected; these two protocol configurations have the same blocking, but the FGLS leads to reduced overhead.

Obs. 2. *The performance of the FLLS relative to the baseline U-C-RNLP is dependent on the percentage of tasks that require resource access.*

If the percentage of tasks issuing requests is at most 10%, FLLS is as good or better than the baseline in 98.6% of scenarios. If instead 50% or more of the tasks issue requests, FLLS is worse than the baseline in 93.1% of scenarios.

Obs. 3. *The SLLS is always the worst option, and the SGLS is almost always the second worst.*

These poor results despite significantly lower overhead highlights the need to develop a better method of ensuring rapid lock server response to newly issued requests while also allowing an analysis method that is not overly pessimistic. For example, a high-priority task could be dedicated to this without requiring the dedication of an entire core.

Platform 2. Next I considered performance on Platform 2. The same scenario depicted in Fig. 2 for Platform 1 is shown in Fig. 3 for Platform 2.

Obs. 4. *Similar performance trends hold for Platform 2.*

Relative to the baseline, the FGLS and FLLS are more dominant on Platform 2 for systems with requests with short critical-section lengths; FGLS is better than the baseline in 95.8% of the scenarios and if at most 50% of tasks issue requests, the FLLS configuration is always better than the baseline. However, for other critical-section lengths, the

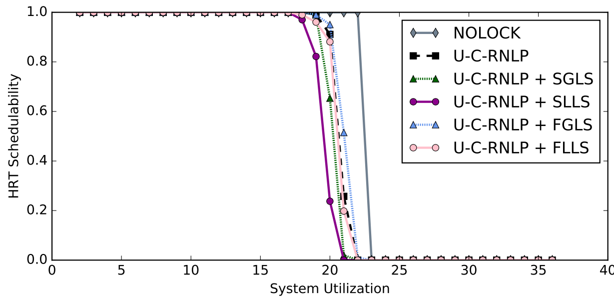


Figure 3: Schedulability on Platform 2 under a scenario in which tasks have short periods, 50% of tasks issue a request, and the nesting depth is 2.

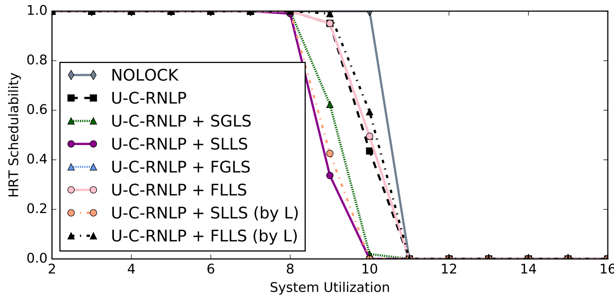


Figure 4: Schedulability on Platform 1 under a scenario in which tasks have short periods, 10% of tasks issue a request, and the nesting depth is 4. (Note: the U-C-RNLP + FGLS line is equivalent to that for the U-C-RNLP + FLLS.)

FLLS performs less well relative to the baseline on Platform 2, and is as good or better than the baseline only when 5% of tasks issue requests.

3.4 Methods for assigning tasks to clusters

I developed a basic heuristic to attempt to reduce blocking for local lock server configurations by leveraging the per-cluster definitions of maximum critical-section length. I explore this with a set of scenarios with bimodal critical-section lengths and for all percentages of tasks issuing requests except 100% (my assignment method performs poorly even at 80%). I assigned any τ_i with $\mathcal{R}_i$ with $L_i \leq 15\mu$s to Cluster 1 and τ_i with $\mathcal{R}_i$ with $L_i > 15\mu$s to Cluster 2. All tasks that did not issue requests were then assigned with Worst-Fit.

Obs. 5. *Different assignment techniques greatly impact the schedulability under local lock server configurations.*

For the FLLS configurations, switching to the assignment heuristic described above clearly improved schedulability in 40.0% of the 20 scenarios, tested on both Platform 1 and Platform 2. Similarly, the SLLS improved in 32.5% of scenarios. (These improvements ignore the 32.5% of scenarios for both platforms in which there was no significant change.)

Switching assignment heuristics also resulted in scenarios in which the FLLS with the new critical-section-length-dependent allocation method (labeled "by L" in Fig. 4) outperformed all other configurations. One such scenario is depicted in Fig. 4, in which tasks had short periods, 10% of tasks issued a request, and the nesting depth was 4.

4. CONCLUSIONS

I have explore the impact of lock server configurations on task system schedulability. While a single configuration, the Floating Global Lock Server, emerged as most effective in this preliminary study, the performance of other configurations can clearly be improved. With better methods for assigning tasks to clusters, schedulability under local lock servers improved. Additionally, more fine-grained methods for accounting for static lock servers would likely improve the results under those options significantly. For example, other tasks could be allowed to execute on the same cores and simply incur a penalty for each time the lock server may need to execute. In the future, I also plan to explore the tradeoffs of lock server configurations on four-socket systems; four local lock servers could be employed on such a platform, but doing so effectively will likely require further work on allocating tasks to clusters in order to reduce blocking.

5. REFERENCES

[1] AUTOSAR Release 4.4, Classic Platform, Specification of Operating System. `https://www.autosar.org/`, 2019.

[2] SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`, 2019. Accessed: 2019-02-07.

[3] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI '98*.

[4] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS '07*.

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

[6] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS '13*.

[7] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS '10*.

[8] 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*.

[9] J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies '17*.

[10] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS '15*.

[11] 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*.

[12] C. Nemitz, T. Amert, and J. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *ECRTS '18*.

[13] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: optimizing the common case. *Real-Time Systems*, 55(2), 2019.

[14] H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *RTCSA '95*.

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