Multiprocessor Real-Time Locking Protocols for Replicated Resources [*]

Catherine E. Nemitz[1], Kecheng Yang[1], Ming Yang[1], Pontus Ekberg[2], and James H. Anderson[1]
[1]Department of Computer Science, University of North Carolina at Chapel Hill
[2]Department of Information Technology, Uppsala University

Abstract

A real-time multiprocessor synchronization problem is studied herein that has not be extensively studied before, namely, the management of replicated resources where tasks may require multiple replicas to execute. In prior work on replicated resources, k-exclusion locks have been used, but this restricts tasks to lock only one replica at a time. To motivate the need for unrestricted replica sharing, two use cases are discussed that reveal an interesting tradeoff: in one of the use cases, blocking is the dominant lock-related factor impacting schedulability, while in the other, lock/unlock overheads are. Motivated by these use cases, three replica-allocation protocols are presented. In the first two, the lock/unlock logic is very simple, yielding low overheads, but blocking is not optimal. In the third, blocking is optimal (ignoring constant factors), but additional lock/unlock overhead is incurred to properly order lock requests. Experiments are presented that examine the overhead/blocking tradeoff motivated by these protocols in some detail.

1 Introduction

Most real-time locking protocols support only non-replicated resources. However, replicated resources arise in many practical settings. For example, on a multicore platform augmented with multiple graphics processing units (GPUs) as accelerators, the pool of available GPUs may be regarded as a replicated resource, with each GPU considered as a single replica. Assuming that a GPU-requesting task can use any available GPU, the problem of allocating GPUs to tasks is equivalent to that of allocating replicas to tasks.

In prior work on real-time resource sharing, k-exclusion locking protocols [2, 5, 17] have been devised that can be used to support replicated resources. A *k-exclusion lock* extends a mutual exclusion lock by allowing up to k simultaneous lock holders [7]. (Thus, mutual exclusion is equivalent to 1-exclusion.) A k-exclusion lock can ensure that each of k replicas is used by only one task. For example, GPU pools are managed in this way in GPUSync, a GPU-management framework that has been the subject of much recent work [3, 4, 6]. Unfortunately, when k-exclusion locks are used to support replication, a task can only request access to one replica at a time: if multiple replicas were to be allocated to the same task, then the number of simultaneous lock holders would need to be restricted to be less than k, and such a guarantee is not provided by a k-exclusion lock.

In this paper, we consider multiprocessor real-time locking protocols for allocating replicated resources where tasks may request multiple replicas. As we shall see, allowing multiple replicas to be requested creates new difficulties with respect to how resource requests are ordered. Before continuing, we illustrate the need for unrestricted replica sharing by means of two use-case scenarios.

First use case: enabling multiple GPU accesses. In domains such as visualization and signal and image processing [11, 12, 13], researchers have proposed harnessing the full parallelism in multi-GPU systems by supporting computations that access multiple GPUs simultaneously. If individual GPUs are viewed as replicas as discussed earlier, then such functionality would require enabling tasks to lock multiple replicas at once.

Second use case: controlling on-chip SRAM contention. In an ongoing industrial collaboration, we are attempting to support soft-real-time workflows on multicore machines where contention for on-chip SRAM memory must be kept below a specified limit. Such a limit can be enforced by defining a pool of "memory tokens," determining the number of such tokens each task needs to execute (based on analysis that indicates the SRAM contention it will cause), and then requiring each task to be allocated its needed tokens before commencing execution. The token pool can be viewed as a replicated resource, with each individual token corresponding to a single replica. Before commencing execution, a task must first lock its needed number of replicas.

Problem variations. Motivated by these use cases, we now discuss several variants of the problem considered in this paper. To begin, note that in the GPU use case, it is important for a task to know which individual GPUs it has been allocated, while in the SRAM use case, tokens are abstract entities that have no real identity. This observation motivates us to distinguish between the problem variants of *allocation* and *assignment*. In solving the replica allocation problem, a task must merely be allocated a specified *number* of replicas. In solving the replica assignment problem, the actual *identities* of the allocated replicas must be known.

[*] Work supported by NSF grants CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from both General Motors and FutureWei Corp.

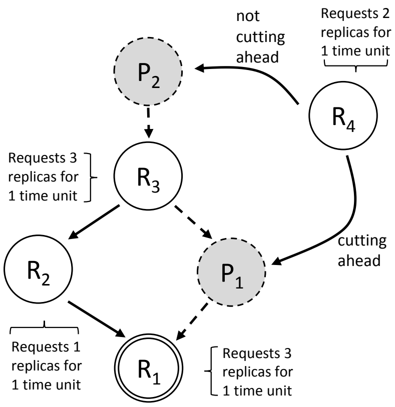


Figure 1: Requests $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ for three, one, and three replicas, respectively, of a resource with three replicas in total. $\mathcal{R}_1$ is currently satisfied, and $\mathcal{R}_2$ and $\mathcal{R}_3$ will be satisfied in increasing index order, as shown. $\mathcal{R}_4$ is newly issued and can cut ahead of $\mathcal{R}_3$ and be satisfied together with $\mathcal{R}_2$ as indicated by position P_1, or not cut ahead, as indicated by position P_2.

When locking protocols are used, tasks may experience delays due to both blocking and lock/unlock-call overheads. In the GPU use case, blocking delays will tend to be more impactful, while in the SRAM use case, the opposite is true. In particular, in the GPU use case, replicas can be expected to be rather scarce (two to eight GPUs might typically exist) and GPU critical sections can be rather long (GPU computation times can range from tens of milliseconds to several seconds, or even longer [3]). In contrast, in the SRAM use case, replicas are abundant (thousands of memory tokens might exist, with most tasks needing fewer than 100), so blocking should be uncommon. These observations motivate us to distinguish between *blocking-optimized* and *overhead-optimized* locking protocols.

Blocking times versus overheads. While it would be desirable to develop locking protocols that give rise to both low blocking and low overheads, doing so is problematic in the case of replicated resources. This is because optimizing blocking involves resolving *cutting-ahead* decisions, and this entails additional overhead. Consider, for example, the situation shown in Fig. 1. This figure depicts four requests, $\mathcal{R}_1, \ldots, \mathcal{R}_4$, for varying numbers of replicas of a resource that has three replicas in total. Blocking relationships are defined via a *wait-for graph*. In the depicted situation, $\mathcal{R}_1$ currently holds all three replicas, $\mathcal{R}_2$ has requested one replica and is blocked by $\mathcal{R}_1$, and $\mathcal{R}_3$ has requested all three replicas and is blocked by both $\mathcal{R}_1$ and $\mathcal{R}_2$. $\mathcal{R}_4$ is a newly issued request for one replica. It can be placed either such that it is blocked by each of the other three requests, or blocked by $\mathcal{R}_1$ and blocking $\mathcal{R}_3$, provided this choice does not increase the worst-case blocking of $\mathcal{R}_3$. In the latter case, we say that $\mathcal{R}_4$ is allowed to *cut ahead* of $\mathcal{R}_3$. Such cutting ahead would allow $\mathcal{R}_2$ and $\mathcal{R}_4$ to be satisfied simultaneously.

Contributions. This paper presents multiprocessor real-time locking protocols for replica allocation and replica assignment that are tailored to reduce either blocking or overheads for systems where tasks may lock multiple replicas simultaneously. Our specific contributions are fourfold. First, in Sec. 3, we present and prove correct a wait-free algorithm that can be applied to any replica allocation protocol to solve the replica assignment problem. The existence of this algorithm frees us from having to consider replica assignment further. Second, in Sec. 4, we present two simple overhead-optimized replica allocation protocols. These protocols have very low overheads but give rise to blocking times that are not optimal. Both protocols are good candidate solutions for the SRAM use case, in which blocking should be rare. To reap either protocol's benefits analytically, holistic blocking analysis is needed that does not grossly overestimate overall blocking within a considered schedulability-analysis interval. We present such holistic analysis in Sec. 5. Third, we present a blocking-optimized replica allocation protocol that employs a cutting-ahead mechanism for which blocking is optimal (assuming constant factors are ignored). We devised this algorithm as a candidate solution for the GPU use case. Finally, we present the results of runtime experiments in which the tradeoff between overhead and blocking motivated by the considered protocols is examined in some detail.

2 Background

In this section, we provide relevant background material.

Task model. We consider the classic sporadic real-time task model with a task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ scheduled on m processors. An arbitrary job of τ_i is denoted J_i and is scheduled using a job-level fixed-priority scheduler to set its *base priority*.

Resource model. We consider k replicas of a single resource. To access the replicas, a job *issues* a *request*. We consider an arbitrary request $\mathcal{R}_i$ of job J_i. This request requires D_i of the k replicas. $\mathcal{R}_i$ is *satisfied* when it *holds* D_i such replicas. It *releases* these replicas when it *completes*. A request $\mathcal{R}_i$ will hold its required number of replicas for at most L_i time units. We define $L_{max} = \max_{1 \leq i \leq n}\{L_i\}$. As noted in Sec. 1, we distinguish between the two problem variants of *replica allocation* and *replica assignment*. In both problem variants, D_i replicas must be allocated to $\mathcal{R}_i$. However, a solution to the allocation problem is not required to return the identities of the allocated replicas, while a solution to the assignment problem is.

Priority inversions. For ease of exposition, we limit our attention to synchronization protocols implemented via algorithms in which spinning (busy-waiting) is used to realize task blocking. Furthermore, such protocols are invoked non-preemptively: a job becomes non-preemptive just before requesting any replica and remains non-preemptive until just after it releases all of its requested replicas. All of our protocols can be converted to variants in which blocking is realized instead by suspending blocked tasks, but we do not have sufficient space to explain these protocol variants.

In the considered context, a job experiences *spin-based blocking* (*s-blocking*) whenever it is forced to busy-wait [1].

Additionally, non-preemptive execution can cause *priority-inversion blocking* (*pi-blocking*). In particular, a job is pi-blocked if its base priority is sufficient for it to be scheduled, but it cannot be scheduled because some lower-priority job is executing non-preemptively. Clearly, worst-case pi-blocking is dependent on worst-case s-blocking. We denote the worst-case s-blocking for a request $\mathcal{R}_i$ as s_i.

Cutting-ahead mechanisms. Fundamentally, all of the allocation algorithms considered in this paper function by determining the *order* in which requests will be satisfied. Generally speaking, such an order will be a partial order, because with a replicated resource, multiple requests potentially can be satisfied concurrently. Unnecessary s-blocking can be reduced by employing a *cutting-ahead mechanism* [8] that sometimes allows a newly issued request to be ordered before prior requests. By allowing a request $\mathcal{R}_i$ to cut ahead of others, the s-blocking $\mathcal{R}_i$ experiences is decreased. Regardless of how requests are ordered, we require two conditions to be upheld: safety and delay preservation.

Delay-preserving algorithms and optimality. The term *safety* merely means that the satisfication of requests must be done in a way that ensures that at most k replicas are allocated at any time. *Delay preservation* is defined with respect to request *insertions*: we say that a request is *inserted* when a shared data structure is updated on account of that request, and this update determines how this request will be ordered with respect to other requests. An insertion of a request is *delay preserving* if it does not increase the worst-case s-blocking time of any existing request. While the necessity of maintaining safety is essential to any system, delay preservation is vital to our analysis of s-blocking bounds. When considering allocation algorithms in this paper, we limit attention to algorithms that are delay-preserving. We define such an algorithm $\mathcal{A}$ to be *optimal* if and only if it minimizes worst-case s-blocking for any new request $\mathcal{R}_i$; that is, it is not possible to insert $\mathcal{R}_i$ into the partial order defined by existing requests in a way that results in less worst-case s-blocking than under $\mathcal{A}$ while still maintaining the requirement of being delay-preserving.

3 Allocation to Assignment

In this section, we show how to solve the assignment problem given a solution to the allocation problem. Specifically, assuming the latter solution supports two routines, ALLOCATE and UNALLOCATE, we show how to implement two corresponding routines for the former problem, ASSIGN and UNASSIGN. This implementation, which is shown in Alg. 1, uses a shared array,[1] *Replica*, of k test-and-set bits,[2] where each such bit represents a distinct replica. As seen in the code, a replica is acquired by performing a successful test-and-set operation on its corresponding bit, and is released by clearing that bit. Note that the code outside of the ALLOCATE and UNALLOCATE routines is entirely wait-free. (Depending on the application, alternative approaches such as a free list or a buddy system could be used that could potentially be more efficient. Our goal here is to provide an approach that is correct in *any* application context, given a solution to the allocation problem.)

The correctness of Alg. 1 can be shown via an invariant-based argument. The required invariant is given in Expression (1) below. The two terms $N[i]$ and $S[i]$ defined next are used in (1). Each request scans *Replica* only once, so $N[i]$ gives the number of available replicas that could still be assigned to $\mathcal{R}_i$. $S[i]$ is the set of (indices of) other requests currently scanning within the same suffix of *Replica* as $\mathcal{R}_i$. The quantification in (1) is assumed to be over those tasks i that are executing within Lines 4–8 in Alg. 1 (including the critical section between Lines 6 and 7).

$$N[i] = |\{p : p[i] \leq p < k :: \mathit{Replica}[p] = 0\}|$$

$$S[i] = \{j : j \neq i :: p[i] \leq p[j]\}$$

$$(\forall i :: (N[i] - \sum_{j \in S[i]} \mathit{rem}[j]) \geq \mathit{rem}[i]) \qquad (1)$$

The invariance of (1) can be shown via a simple inductive proof: it holds initially and is never falsified.[3] The correctness of Alg. 1 follows as a result, because (1) implies that, as a request $\mathcal{R}_i$ is scanning *Replica*, it is guaranteed that there are sufficiently many available replicas "upstream" of its current scan position to fulfill its remaining replica needs. Due to space constraints, we do not provide formal proofs of these properties here. Instead, we have opted to consider an example that illustrates some of the more interesting corner cases that arise when showing that (1) holds.

Ex. 1. We consider four requests, $\mathcal{R}_u$, $\mathcal{R}_v$, $\mathcal{R}_w$, and $\mathcal{R}_x$, in a system with $k = 10$ replicas, indexed from 0 to 9, where the number of replicas required per request is given by $D_u = 3$, $D_v = 3$, $D_w = 2$, and $D_x = 4$, respectively. Fig. 2 shows several snapshots of the *Replica* array, with each request's current position (given by its $p[-]$ variable) as it scans the array labeled above the array. Each array entry either is 0 or is labeled by the index of the request that has set it to 1. In discussing these snapshots, we refer to various *instantiations* of (1). (1) is actually a family of invariants, one per request, as given by the quantified variable i. When we refer to an instantiation for some request $\mathcal{R}_z$, we are referring to the quantified inner expression with $i \leftarrow z$.

In Fig. 2(a), $\mathcal{R}_x$ has reserved three replicas and has one more to reserve. $\mathcal{R}_w$ has finished executing ASSIGN$(w, 2)$ and is now in its critical section. $\mathcal{R}_v$ also requires one more replica. Finally, $\mathcal{R}_u$ has called ALLOCATE$(u, 3)$ and is waiting for that call to complete. Nine replicas are currently allo-

[1] For greater clarity, we capitalize shared variable names and denote private variable names in lowercase.

[2] The operation *test&set*(B) sets the bit B to be 1 and returns its original value. Such an operation is "successful" if the return value is 0.

[3] In formally reasoning about this code, we assume that each labeled statement is atomic. Each such statement references at most one shared variable, or is an invocation of a routine, the specification of which allows us to assume atomic execution. Thus, this assumption is easily realized in practice. Such atomicity assumptions are often made in reasoning about concurrent algorithms to simplify analysis.

Algorithm 1 Allocation to Assignment

```
Replica: array 0 to $k-1$ of 0..1 initially 0 /* shared */
$p[i]$, $rem[i]$: 0..$k$ /* private to $\mathcal{R}_i$ */
procedure ASSIGN($i$: 1..$n$, requested: 1..$k$)
1:  ALLOCATE($i$, requested)
2:  $p[i] \leftarrow 0$
3:  $rem[i] \leftarrow requested$
4:  while $rem[i] > 0$ do
5:      if test&set(Replica[$p[i]$]) = 0 then
            assign replica $p[i]$ to $\mathcal{R}_i$
            $rem[i] \leftarrow rem[i] - 1$
        end if
6:      $p[i] \leftarrow p[i] + 1$
    end while
end procedure
procedure UNASSIGN($i$: 1..$n$, requested: 1..$k$)
7:  for each $p[i]$ where replica $p[i]$ was assigned to $\mathcal{R}_i$ do
8:      Replica[$p[i]$] $\leftarrow 0$
    end for
9:  UNALLOCATE(i, requested)
end procedure
```

cated (though some are still unassigned), so this call will not return until one of the other requests unallocates its replicas. Note that all relevant instantiations of (1) hold.

In Fig. 2(b), $\mathcal{R}_w$ has freed all of its replicas by completing an invocation of UNASSIGN$(w, 2)$. (1) is no longer applicable to request $\mathcal{R}_w$ because it is no longer executing within Lines 4–8. For $\mathcal{R}_v$, $N[v]$ increases, clearly maintaining request $\mathcal{R}_v$'s instantiation of (1). The instantiation for request $\mathcal{R}_x$ is not affected in any way by the completion of $\mathcal{R}_w$. Additionally, $\mathcal{R}_x$ has reserved another replica, so *rem*$[x] = 0$ now holds and its invocation ASSIGN$(x, 4)$ is completed. $\mathcal{R}_u$ has now been allocated (but not yet assigned) its $D_u = 3$ resources. This did not invalidate request $\mathcal{R}_u$'s instantiation of (1). Additionally, $\mathcal{R}_u$ moved forward to position $p[u] = 1$ after determining that replica 0 was already reserved. This movement also did not invalidate request $\mathcal{R}_u$'s instantiation of (1).

In Fig. 2(c), $p[u]$ has been incremented to equal $p[v]$. Just before the increment, both $N[u] - rem[v] \geq rem[u]$ and $N[v] - rem[v] \geq rem[v]$ held, by (1) (note that *rem*$[w]$ and *rem*$[x]$ are both 0). Note also that immediately before the increment, $N[v] = N[u]$ held. Furthermore, neither $N[v]$ nor $N[u]$ was affected by the increment, nor were *rem*$[u]$ and *rem*$[v]$. Thus, all instantiations of (1) continue to hold.

In Fig. 2(d), $\mathcal{R}_u$ has acquired resource 4 by means of a successful *test&set* operation on position $p[u] = 4$. This does not affect request $\mathcal{R}_u$'s instantiation of (1) as the resource acquisition decrements both sides of the inequality for this instantiation. The instantiation for request $\mathcal{R}_v$ is also not invalidated, because both $N[v]$ and *rem*$[u]$ are decreased by one, resulting in no net change to the left side of the inequality.

In Fig. 2(e), $\mathcal{R}_u$ has reserved replica 5 and $\mathcal{R}_v$ has reserved replica 9. These reservations maintain all instantiations of (1).

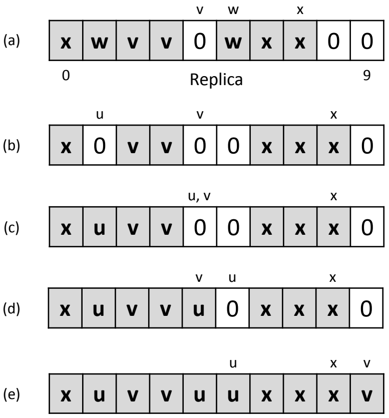


Figure 2: Several snapshots of the *Replica* array as requests $\mathcal{R}_u$, $\mathcal{R}_v$, $\mathcal{R}_w$, and $\mathcal{R}_x$ move through the code of Alg. 1.

4 Overhead-Optimized Allocation

Having just shown that the assignment problem can be solved given any solution to the allocation problem, we now turn our attention to the latter problem. In this section, we consider solutions to the allocation problem for which lock and unlock overheads are low. As we shall see, these solutions are not optimal. Later, in Sec. 6, we show how to eliminate unnecessary blocking and achieve optimality. As we shall see, this improvement comes at the expense of significantly higher overheads.

Our first solution, which is given in Alg. 2, is a variation of a ticket lock that uses two unbounded shared counters, *Num_requested* and *Num_released*. As their names suggest, these counters indicate the total number of replicas requested and released, respectively, up to the current time. These counters are updated atomically using *fetch&add* instructions.[4] As seen in the ALLOCATE routine, a request $\mathcal{R}_i$ must block[5] until enough replicas have been released to accommodate all replica requests prior to and including $\mathcal{R}_i$. Using this algorithm, a non-blocked request can acquire and release its needed resources by performing at most ten machine instructions. This low overhead comes at the expense of needing to use unbounded counters to preclude counter overflow, which could compromise safety. In many application domains, however, bounded counters would suffice because such counters can be guaranteed to not overflow during a system's lifetime. For example, a 64-bit counter that is incremented by 100 every ten nanoseconds will not overflow for 58.5 years.

If counter overflow is a potential concern, then our second solution, given in Alg. 3, can be used. This solution is actually nothing more than a counting semaphore implemented using busy-waiting. The shared variable *Num_available* gives the number of replicas currently available. The ALLOCATE and UNALLOCATE routines implement the semaphore P and V operations. These operations

[4] The operation *fetch&add*(X, v) performs the assignment $X \leftarrow X + v$ and returns the original value of X.

[5] We assume all **wait until** constructs are implemented by busy-waiting.

Algorithm 2 Overhead-Optimized Protocol: Unbounded

```
$Num\_requested : 0..\infty$ initially $0$ /* shared */
$Num\_released : 0..\infty$ initially $0$ /* shared */
$num\_reqd\_incl\_me: 0..\infty$ initially $0$ /* private to each request */
$num\_releases\_required: 0..\infty$ initially $0$ /* private to each request */
procedure ALLOCATE($needed : 1..k$)
  $num\_reqd\_incl\_me \leftarrow fetch\&add(Num\_requested, needed) + needed$
  $num\_releases\_required \leftarrow num\_reqd\_incl\_me - k$
  wait until $Num\_released \geq num\_releases\_required$
end procedure
procedure UNALLOCATE($needed : 1..k$)
  $fetch\&add(Num\_released, needed)$
end procedure
```

Algorithm 3 Overhead-Optimized Protocol: Bounded

```
$Num\_available : 0..k$ initially $0$ /* shared */
procedure ALLOCATE($needed : 1..k$)
  $Acquire(Queue\_lock)$
      wait until $Num\_available - needed \geq 0$
      $fetch\&add(Num\_available, -needed)$
  $Release(Queue\_lock)$
end procedure
procedure UNALLOCATE($needed : 1..k$)
  $fetch\&add(Num\_available, needed)$
end procedure
```

must be atomic. Atomicity is ensured in the ALLOCATE routine by using an underlying FIFO queue-based spin lock [9]. Atomicity is ensured in the UNALLOCATE routine by using an atomic synchronization primitive.

Theorem 1. *Neither Alg. 2 nor Alg. 3 is optimal.*

Proof. Consider requests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$ and $\mathcal{R}_5$, which require $D_1 = 6$, $D_2 = 5$, $D_3 = 6$, $D_4 = 5$, and $D_5 = 6$ replicas, respectively, in a system with $k = 10$ replicas in total. Suppose that each of these requests executes for at most one time unit ($L_i = 1$ for each i) and that they are issued in increasing index order. Then, under either Alg. 2 or Alg. 3, they would be ordered as in the wait-for graph shown in Fig. 3. Note that $\mathcal{R}_2$ cannot be satisfied until $\mathcal{R}_1$ completes, $\mathcal{R}_3$ cannot be satisfied until $\mathcal{R}_2$ completes, and so on. Now, consider a new request $\mathcal{R}_6$ that requires $D_6 = 5$ replicas for up to one time unit ($L_6 = 1$). Under either algorithm, $\mathcal{R}_6$ can be satisfied only after $\mathcal{R}_5$ completes, *i.e.* it would be inserted into the position denoted as P_2 in the wait-for graph. However, $\mathcal{R}_6$ could be satisfied simultaneously with $\mathcal{R}_2$ (*i.e.*, be inserted into position P_1) without violating the requirement of being delay-preserving. Thus, neither algorithm is optimal. □

Note that $\mathcal{R}_6$ cannot be satisfied simultaneously with $\mathcal{R}_2$ in the request sequence considered above because neither Alg. 2 nor Alg. 3 provides a cutting-ahead mechanism.

From a schedulability perspective, applying either Alg. 2 or Alg. 3 requires bounds on s-blocking times. Under either algorithm, a coarse bound of $s_i = (m-1) \cdot L_{max}$ applies, because a given request can be s-blocked by at most $m-1$ other requests (since requests execute non-preemptively), and in the worst case, these other requests execute without concurrency, for up to L_{max} time units each.

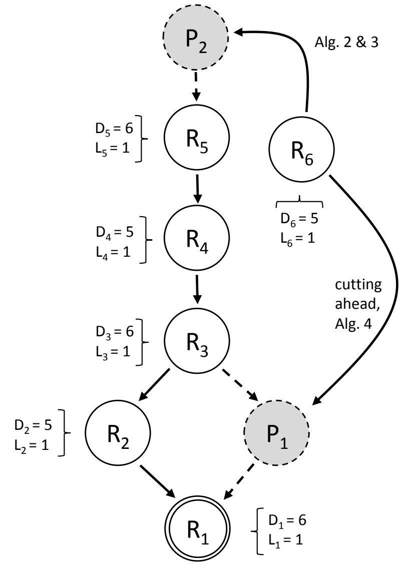


Figure 3: Existing requests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$ and $\mathcal{R}_5$ and possible positions P_1 and P_2 into which $\mathcal{R}_6$ could be inserted.

Generally speaking, obtaining tight blocking bounds for real-time synchronization protocols can be difficult. In an online appendix, we prove that the problem of computing tight s-blocking bounds for Algs. 2 and 3 is NP-hard in the strong sense [10]. (This problem has connections to non-preemptive gang scheduling.) If tight bounds are needed, then the following exponential-time method can be used. Given a specific ordered sequence of $m-1$ requests issued prior to a particular request $\mathcal{R}_i$, we can simulate the blocking sequence in pseudo-polynomial time to obtain a tight s-blocking bound for $\mathcal{R}_i$ *in this sequence*. To determine a general bound, however, we have to examine all possible ordered sequences of $m-1$ requests. This takes exponential time, because the number of possible permutations of $m-1$ requests taken from $n-1$ other tasks, with at most one taken from any job, is at least $\binom{n-1}{m-1} \cdot (m-1)!$ (and could be greater if jobs make multiple requests). If the number of permutations that need to be considered is not too large, then this method might be acceptable.

5 Holistic S-Blocking Analysis

We envision the overhead-optimized protocols just described to be of interest in settings like the SRAM use case where blocking should be rare. In such settings, overheads will tend to impact schedulability more than blocking, *provided* we are able to reap the benefits of rare blocking analytically. In particular, if *worst-case* s-blocking is pessimistically assumed for *every* request in schedulability analysis, then overall s-blocking will be vastly overestimated.

In this section, we present holistic s-blocking analysis that avoids such pessimism. (Our analysis was inspired by recent holistic blocking analysis presented by Ward [15], although the framework considered here is somewhat different.) Specifically, we show how to upper bound overall s-blocking for a sequence of requests, $R_1, R_2, \cdots, R_n$. Recall that each request R_i needs D_i replicas to be satisfied and will hold its needed replicas for at most L_i time units. Due to space constraints, we consider only Alg. 2, though similar analysis can be applied to Alg. 3.

Def. 1. (Aggregate Replica Execution Time) If a request executes for t time units while holding x replicas, then we say that it has an *Aggregate Replica Execution Time (ARET)* of $x \cdot t$.

Clearly, the ARET of each request R_i is at most $D_i \cdot L_i$. Thus, letting S denote the total ARET for the entire sequence of requests, we have

$$S \leq \sum_i D_i L_i. \tag{2}$$

Def. 2. (Direct and Indirect S-Blocking) In Alg. 2, a request experiences *direct s-blocking* if it is blocked at Line 3, and among those requests blocked at Line 3, the value of its private variable *num_releases_required* is the smallest. A request that is blocked at Line 3 for which the latter condition does not hold is said to experience *indirect s-blocking*.

Lemma 1. *At any time instant, if some request is s-blocked, then there must exist a unique request that experiences direct s-blocking.*

Proof. Follows from Def. 2. □

Let b_i be the total duration of time for which request R_i experiences direct s-blocking.

Lemma 2. *The total ARET over all time instants when R_i experiences direct s-blocking is at least $b_i(k - D_i + 1)$.*

Proof. When R_i experiences direct s-blocking, there are at most $(D_i - 1)$ replicas available. That is, at least $k - (D_i - 1) = (k - D_i + 1)$ replicas are held by other requests that are currently executing. Thus, the lemma follows. □

By Lemma 2,

$$S \geq \sum_i b_i(k - D_i + 1) \geq \sum_i b_i(k - D_{\text{max}} + 1), \tag{3}$$

where $D_{\text{max}} = \max_{1 \leq i \leq n}\{D_i\}$.

Lemma 3. *The total time duration when at least one request experiences s-blocking is $\sum_i b_i$. Furthermore,*

$$\sum_i b_i \leq \frac{\sum_i D_i L_i}{k - D_{\text{max}} + 1}. \tag{4}$$

Proof. The first part of the lemma follows from Lemma 1. The second part of the lemma follows from (2) and (3). □

Let $\mathfrak{D}_j$ denote the sum of j largest values in $\{D_i\}$. In the following lemma, q denotes the minimum number of requests that must hold replicas for other requests to potentially block; $q = m$ implies that no blocking will occur.

Lemma 4. *If $\mathfrak{D}_m \leq k$, then let $q = m$; otherwise, let q be the largest positive integer $(1 \leq q \leq m - 1)$ such that $\mathfrak{D}_q \leq k$. Then, at most $m - q$ requests can simultaneously busy-wait.*

Proof. To begin, note that $\mathfrak{D}_1 \leq k$ holds. Otherwise, there exists a request that will never be satisfied in any case. Thus, the two cases for q stated in the lemma are exhaustive. The first case, $\mathfrak{D}_m \leq k$, is straightforward, as no request would ever busy-wait in this case (*i.e.*, $q = m$), since there are only m processors. In the remainder of the proof, we focus on the second case. Suppose that there are at least $m - q + 1$ requests that are simultaneously busy-waiting and R_i is the one that experiences direct s-blocking. Because requests are assumed to execute non-preemptively, there are at most $m - (m - q + 1) = q - 1$ requests holding replicas. Because R_i experiences direct s-blocking, these at most $q - 1$ currently executing requests hold at least $k - D_i + 1$ replicas. Thus, including R_i, there are at most q requests that in total require $k + 1$ replicas. This contradicts the definition of q, which implies $\mathfrak{D}_q \leq k$. □

Theorem 2. *The total s-blocking experienced by the sequence of requests is at most*

$$(m - q)\sum_i b_i \leq \frac{(m - q)\sum_i D_i L_i}{k - D_{\text{max}} + 1},$$

where q is as defined in Lemma 4.

Proof. Follows directly from Lemmas 3 and 4. □

Ex. 2. Consider a sequence of n requests in a system with $k = m - 1$, where for each request R_i, $D_i = 1$ and $L_i = L$. Then, by Lemma 4, $q = m - 1$, and by Theorem. 2, total s-blocking is at most

$$\frac{(m - (m - 1)) \cdot nL}{m - 1 - 1 + 1} = \frac{nL}{m - 1}.$$

In contrast, if we were to charge each request a coarse per-request s-blocking bound of $(m - 1) \cdot L$, then this would yield a total s-blocking bound of $(m - 1) \cdot nL$. Note that a per-request bound of $(m-1) \cdot L$ does not take the potential of concurrent request satisfaction into account. Taking this into account yields a per-request bound of L. This yields a total s-blocking bound of nL for the entire sequence, which is still inferior to that derived above using our holistic analysis.

6 Blocking-Optimized Allocation

In considering the allocation problem so far, we have focused on solutions designed to yield low overheads. However, as we saw in Thm. 1, these solutions are not optimal and can result in unnecessary s-blocking. Excessive s-blocking can lead to poor schedulability in use cases (such as the GPU use case mentioned in Sec. 1) where tasks may block frequently and/or for relatively long durations. In this section, we present a blocking-optimized allocation algorithm that was designed for such use cases and employs a cutting-ahead mechanism. As we shall see, enabling lower s-blocking comes at the expense of higher overheads.

Both of our overhead-optimized algorithms (Algs. 2

and 3) are based on known synchronization constructs, namely, ticket locks and counting semaphores. The blocking-optimized algorithm presented in this section is also based on a known idea, namely, that of a *timing wheel*. Timing wheels were introduced to efficiently multiplex multiple timers within an operating system [14]. Ignoring certain pragmatic details, our blocking-optimized algorithm can be viewed as an approach that merely involves setting timers: instead of blocking until being released by some other request, each request instead sets a timer that expires when its needed replicas will be available. However, we cannot use the concept of a timing wheel directly to support the needed timers, because in our case, the manner in which a timer is set depends on previously set timers. Thus, we must modify the concept of a timing wheel for our purposes. So that our modifications are understandable, we first give a brief overview of timing wheels.

Timing wheels. Timing wheels enable multiple timers to be supported in a way that allows any timer to be started or stopped in $O(1)$ time. A timing wheel is a circular buffer of slots, each representing a discrete segment of time. For example, if the maintenance of a timing wheel occurs after an interrupt from a hardware clock that is issued every millisecond, then each timing wheel slot represents a $1ms$ time interval. While timing wheels can be optimized for various scenarios by the use of trees, linked lists, or additional queues, or by employing timing wheels hierarchically, the basic concept is quite simple: the granularity of time is given by the slot length, and timers that expire in the same slot are stored and processed together [14].

Basic allocation algorithm description. Ignoring certain technicalities for now, our blocking-optimized algorithm, shown in Alg. 4, functions as follows. A timing wheel, given by the array *Timing_wheel*, is used, where the slot length is given by the parameter *Slot_size*. The array's length (*i.e.*, total number of slots) is determined in a way that ensures that "wrapping" causes no problems, as explained below. Each *Timing_wheel* entry is actually a counter that indicates the number of available replicas in the corresponding slot.

In considering the ALLOCATE and UNALLOCATE routines, assume for now that these routines take zero time to execute. As before with Alg. 3, an underlying FIFO queue-based spin lock is used to ensure that these routines execute atomically (Lines 1, 8, 12, and 18). In Line 3 of ALLOCATE, a sequence of consecutive slots in *Timing_wheel* is found that can accommodate the given request $\mathcal{R}_i$. Here and later in the code, we use the notation *Slot_set*(s, n) as a shorthand for the set of n slots $\{s, s \oplus 1, s \oplus 2, ..., s \oplus (n-1)\}$ in *Timing_wheel*, where $\oplus$ represents addition modulo-*Size* (as explained later, *Size* is the size of *Timing_wheel*). After a suitable set of slots is acquired, their counters are decremented by D_i in Lines 5-6. The request $\mathcal{R}_i$ then s-blocks in Line 9 until its needed replicas are available by busy-waiting until its first acquired slot corresponds to the current time (ignoring the variable *Delta* for now). This busy-waiting corresponds to waiting for a timer expiration. In the UNALLOCATE routine, the acquired replicas are freed in Lines 13-14 by incrementing the corresponding per-slot counters by D_i.

Algorithm 4 Blocking-Optimized Protocol

```
Size : constant /* shared */
Slot_size : constant /* shared */
Timing_wheel : array 0 to Size−1 of $0..k$ initially $k$ /*shared*/
Delta : $0..\infty$ initially 0 /* shared */
Num_available : $0..k$ initially 0 /* shared */
Pending_requests : $0..m$ initially 0 /* shared */
start_time : $0..\infty$ initially 0 /* private to each request */
start_slot : 0..Size initially 0 /* private to each request */
$j$ : $0..\infty$ initially 0 /* private to each request */
$t$ : $0..\infty$ initially 0 /* private to each request */
num_slots : $0..\lceil \frac{L_{max}}{Slot\_size} \rceil$ initially 0 /* private to each request */
procedure ALLOCATE($i : 1..m$)
1:  Acquire(Queue_lock)
2:      num_slots ← $\lceil \frac{L_i}{Slot\_size} \rceil$
3:      start_time ← min $t$ s.t. $t \geq$ get_time() + Delta and
                    Timing_wheel[$j$] $\geq D_i$ for
                    $j \in$ Slot_set($t$, num_slots)
4:      start_slot ← $\lceil \frac{start\_time}{Slot\_size} \rceil$ mod Size
5:      for $j \in$ Slot_set(start_slot, num_slots) do
6:          Timing_wheel[$j$] ← Timing_wheel[$j$] − $D_i$
        end for
7:      Pending_reqeusts ← Pending_requests + 1
8:  Release(Queue_lock)
9:  wait until start_time − get_time() − Delta $\leq 0$
10: if fetch&add(Num_available, $-D_i$) − $D_i < 0$ then
11:     return − 1
    end if
end procedure
procedure UNALLOCATE($i : 1..m$)
12: Acquire(Queue_lock)
13:     for $j \in$ Slot_set(start_slot, num_slots) do
14:         Timing_wheel[$j$] ← Timing_wheel[$j$] + $D_i$
        end for
15:     Pending_reqeusts ← Pending_requests − 1
16:     Num_available ← Num_available + $D_i$
17:     Set_delta()
18: Release(Queue_lock)
end procedure
procedure Set_delta ()
19: if Pending_requests = 0 then
20:     Delta ← 0
21: else if Num_available = $k$ then
22:     Delta ← min $t$ s.t. $\exists$ a request with
                start_time − get_time() − $t = 0$
23: end if
end procedure
```

Having presented a high-level description of the algorithm, we now delve into various technicalities ignored so far, after which we present an example that illustrates how cutting ahead is supported by the algorithm.

Dealing with overrunning requests. In Alg. 4, a request blocks until its needed replicas are available by waiting for a timer to expire. The timer expiration time is determined based on the execution times of previous requests, specifically, the L_i value of each such request $\mathcal{R}_i$. If $\mathcal{R}_i$ were to overrun its specified worst-case lock-holding time of L_i, then safety could be compromised. Note that such an overrun indicates that flawed timing analysis was employed, and as a result, any corresponding schedulability analysis is flawed as well. Still, it would be unwise to allow such an

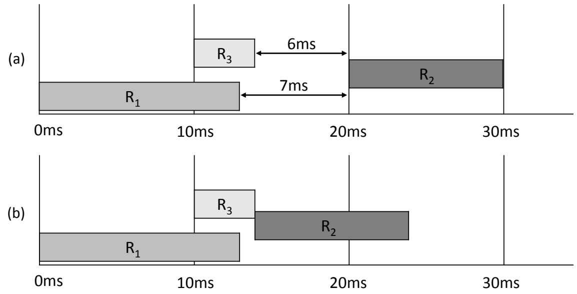


Figure 4: Depiction of the duration of $\mathcal{R}_1$ and $\mathcal{R}_3$ and the waiting of $\mathcal{R}_2$ with **(a)** worst-case blocking enforced and **(b)** the reduction of such blocking by incrementing *Delta*.

analysis flaw to be magnified by allowing the system's state to be corrupted. The check in Line 10 of ALLOCATE guards against such corruption. Specifically, if Line 11 is reached, then an overrun must have occurred, because some replica that should have been available at this point is still being held. In this case, an error code is returned, which allows the invoking task to abort its request.

Improving runtime performance. In the discussion above, we considered the possibility of a request $\mathcal{R}_i$ overrunning its L_i value. Given that these values represent *worst-case* lock-holding times, it is much more likely that these values are underrun, perhaps significantly. This possibility reveals a significant disadvantage of using a timer-based approach: a request may be forced to block for a worst-case amount of time even when the replicas it requires have already been released. Such a situation is depicted in Fig. 4(a), in which $\mathcal{R}_2$ waits to begin executing until time $20ms$, even though $\mathcal{R}_1$ and $\mathcal{R}_3$ have both completed. From a schedulability point of view, such unnecessary blocking is not problematic at all, but from the perspective of runtime performance, eliminating unnecessary blocking would be highly desirable.

In Alg. 4, such unnecessary blocking is ameliorated through the introduction of the variable *Delta*, which, when incremented, has the effect of allowing time to be instantaneously advanced. In particular, while *Timing_wheel* is indexed in *Slot_size* increments, each request actually waits to begin based on the current time plus *Delta*. If *Delta* increases, any waiting requests will be satisfied that much earlier than originally expected. Thus, incrementing *Delta* has the effect of shifting time ahead. With this modification, requests must be inserted into *Timing_wheel* based on the current time plus *Delta*.

The value of *Delta* is updated in the routine *Set_delta*. When invoked by a request, this routine will either increment *Delta*, reset it to zero, or leave it unchanged. If *Pending_requests* $= 0$, then *Delta* is reset to zero. Otherwise, if *Num_available* is less than k, then no change is made to *Delta*, as there is at least one request $\mathcal{R}_i$ that, if time were to advance, might be deemed to have overrun its L_i bound, regardless of whether it actually did so. Finally, if neither of these conditions holds, then *Delta* can be increased. In this case, it is increased by an amount that ensures that the current time plus *Delta* equals the smallest *start_time* value of any non-satisfied request. Fig. 4(b) demonstrates this, as $\mathcal{R}_3$ sets *Delta* $= 6$ and $\mathcal{R}_2$ becomes satisfied immediately.

Determining a safe bound on the size of *Timing_wheel*. We denote the total number of slots in *Timing_wheel* by the parameter *Size*. A safe upper bound on *Size* can be determined as follows. Because requests are executed non-preemptively, there can be at most m active requests at any time, each executing for at most L_{max} time units. Each such request can require at most $\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil$ consecutive slots. To determine the size of *Timing_wheel* required in the worst case, we can consider inserting a request $\mathcal{R}_i$ into *Timing_wheel* that requires $\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil$ consecutive slots with $m-1$ requests already inserted. In the worst case, the existing requests will each use $\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil$ slots and be separated by $\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil - 1$ unused slots. As a result, with only $(m-1)(2\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil - 1)$ slots it may be impossible to insert $\mathcal{R}_i$. However, adding a single additional slot enables $\mathcal{R}_i$ to be inserted, as then at least one "gap" between existing requests will have $\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil$ unoccupied slots. Thus, with at least $(m-1)(2\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil - 1) + 1$ slots in total, any new request can be inserted. Thus, *Size* can be set equal to $(m-1)(2\lceil \frac{L_{max}}{\mathit{Slot\_size}} \rceil - 1) + 1$.

Accounting for the fact that ALLOCATE and UNALLOCATE do not take zero time. In reality, the ALLOCATE and UNALLOCATE routines do not take zero time. However, the execution times of these routines can be easily accounted for by simply inflating each L_i term by the worst-case execution time of both ALLOCATE and UNALLOCATE.

Ignoring constant factors, using Alg. 4 instead of Alg. 2 or 3 can lessen s-blocking times. According to the following theorem, Alg. 4 is in fact optimal under certain ideal conditions. (Recall from Sec. 2 that we consider the issue of optimality only with respect to delay-preserving algorithms.)

Theorem 3. *If requests are initiated only at slot boundaries,* ALLOCATE *and* UNALLOCATE *take zero time, and Slot_size is defined so that every L_i is a multiple of Slot_size, then Alg. 4 is optimal.*

Proof. Suppose a request $\mathcal{R}_i$ is inserted into the partial order in a position that does not minimize its worst-case s-blocking. Then there exists an earlier slot boundary at which it could have been satisfied. However, since Alg. 4 begins searching for a safe, delay-preserving position into which $\mathcal{R}_i$ can be inserted starting with the first future slot boundary and then considering slots in increasing order, $\mathcal{R}_i$ would have been inserted at this earlier slot boundary. □

Corollary 1. If ALLOCATE and UNALLOCATE take zero time and *Slot_size* is selected to be arbitrarily small, then Alg. 4 is optimal.

In an online appendix, we prove that the problem of computing tight s-blocking bounds for Alg. 4 is NP-hard in the strong sense [10]. (As with Algs. 2 and 3, there are connections here to non-preemptive gang scheduling.) Such bounds can be computed in exponential time in a similar

Algorithm 5 Shortest Queue Protocol

```
procedure ALLOCATE(needed : $1..k$)
  Acquire(Queue_lock)
        enqueue $\mathcal{R}_i$ on shortest $D_i$ queues
  Release(Queue_lock)
        wait until $R_i$ is at the head of each of its chosen queues
end procedure
procedure UNALLOCATE(needed : $1..k$)
  Acquire(Queue_lock)
        dequeue $\mathcal{R}_i$ from all queues
  Release(Queue_lock)
end procedure
```

manner as explained previously in Sec. 4 for Algs. 2 and 3. However, the calculation of the s-blocking experienced by a request $\mathcal{R}_i$ given a fixed sequence of $m - 1$ prior requests (obviously) must be based on Alg. 4.

Under the ideal conditions expressed in Cor. 1, s-blocking under Alg. 4 can be no worse than under Algs. 2 and 3. As the following example shows, the cutting-ahead mechanism used in Alg. 4 can result in s-blocking bounds that are actually lower.

Ex. 3. Consider again the task system presented in the proof of Thm. 1 and assume the ideal conditions expressed in Thm. 3. Using either Alg. 2 or 3, $\mathcal{R}_6$ would experience 5 time units of s-blocking. If the pre-existing requests shown in Fig. 3 were ordered as shown, then under Alg. 4, $\mathcal{R}_6$ would be inserted into position P_2 and experience only 1 time unit of blocking. In fact, under Alg. 4, all of the even-indexed requests would have similarly been able to cut ahead and be satisfied together in pairs. By considering all possible request sequences, $\mathcal{R}_6$'s worst s-blocking under Alg. 4 can be shown to be 4 time units.

7 Experimental Evaluation

To evaluate Algs. 2, 3, and 4, we conducted a series of experiments in which we measured relevant overheads and blocking times. We performed these experiments on a dual-socket 18-cores-per-socket Intel Xeon E5-2699 platform. We also compared our algorithms to a pre-existing algorithm that was briefly sketched in a prior paper on hardware management [16]. This prior algorithm, which was not optimized for either overheads or blocking, is shown in Alg. 5.

Measuring overheads and blocking. We instrumented each ALLOCATE and UNALLOCATE routine to record the total spinning time and total execution time of each routine. The *s-blocking* experienced by a request is given by the total spinning time across both routines (recall that some of the UNALLOCATE routines require acquiring a spin lock). The *overhead* associated with the request is given by the total execution time of both routines minus the total spinning time of both routines. We measured s-blocking and overheads as a function of a number of parameters, including critical-section length (L_i), replica count (k), the number of contending tasks (n), and the number of cores (m). Tasks were statically pinned to cores (thus, $n = m$) and were assigned to the same socket for core counts that allowed such an assignment. (Tasking pinning is motivated by the assumption that requests execution non-preemptively.) Each task was defined to invoke ALLOCATE and then UNALLOCATE in a tight loop 1,000 times, with a critical section of a specified length executed between these calls. In each given experiment, we recorded the 99$^{\text{th}}$ percentile of the recorded s-blocking and overhead values to obtain worst-case results while filtering for any spurious behavior (*e.g.*, perturbations caused by interrupts).

Parameters. In our experiments, we varied various parameters, including those mentioned above (L_i, k, and n). Because worst-case critical section lengths (L_i) must be based on timing analysis, which can be pessimistic, we also included a parameter *cs_ratio* that indicates the proportion of L_i for which a request $\mathcal{R}_i$ actually executes.

Results. Due to space constraints, we only present graphs for a subset of our collected data here; the full set of graphs is available in an online appendix [10]. The following observation is supported by the range of data we collected.

Obs. 1. As expected, overheads under Algs. 2 and 3 were substantially lower than under Algs. 4 or 5.

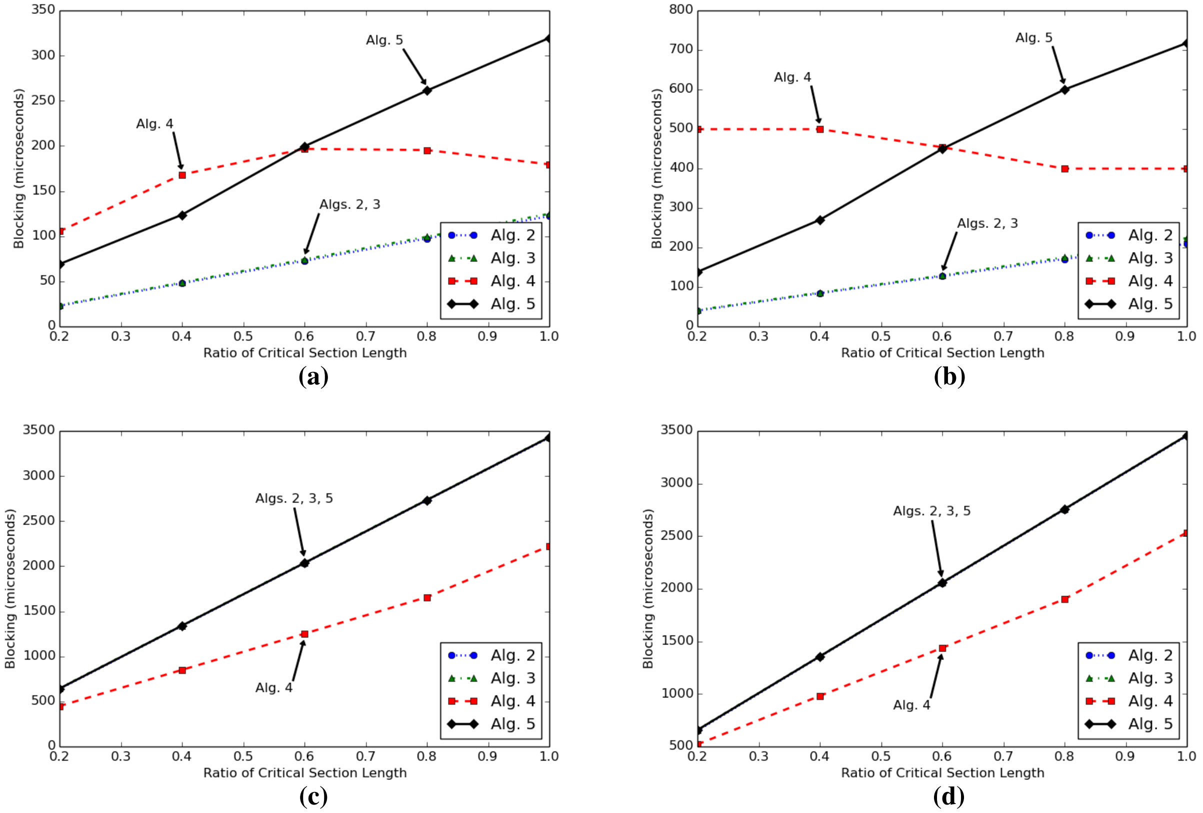
Across all of the experiments we conducted, overheads under Algs. 2 and 3 tended to be at most $0.35\mu s$ on average and at most $0.49\mu s$ in the worst case. In contrast, overheads under Alg. 4 tended to range within $[1.4, 8.8]\mu s$ on average and within $[2.2, 11.8]\mu s$ in the worst case, and under Alg. 5 within $[0.5, 36]\mu s$ on average and within $[1.7, 105]\mu s$ in the worst case. (The overheads of Algs. 4 and 5 could be potentially reduced with further work.)

Obs. 2. In most of the experimental scenarios that we examined, all four algorithms caused comparable s-blocking.

While Alg. 4 has a clear advantage in pathological cases, a typical release pattern of requests does not allow significant cutting ahead. Thus, the ordering of requests generated by Alg. 4 in our experimental framework was very similar to that generated by any of Algs. 2, 3, or 5.

Our remaining observations are supported by the graphs given in Fig. 5. This figure plots s-blocking data for each considered algorithm as a function of *cs_ratio*. This data comes from two experimental scenarios, referred to here as the *low-contention scenario* (reflective of the SRAM use case mentioned in Sec. 1) and the *high-contention scenario* (reflective of the GPU use case), respectively. These scenarios are primarily distinguished by the difference in the ratio of D_i to k. Further, in the latter scenario, we structured the release of requests based on D_i values so that cutting ahead could often occur and would be highly beneficial. In the low-contention scenario, we defined $n = 18$, $D_i \in [1, 9]$, and $k = 50$. In contrast, in the high-contention scenario, we defined $k = 10$ and $n = 18$ and set D_i to either 2 or 9. (These scenarios and further details concerning how our measurements were obtained are more fully described online [10].)

Obs. 3. In the low-contention scenario, s-blocking was substantially lower under Algs. 2 and 3 than under Algs. 4 and 5.



(a) (b) (c) (d)

Figure 5: Blocking times as a function of *cs_ratio*. The top-row (resp., bottom-row) graphs correspond to the low-contention (resp., high-contention) scenario. The left (resp., right) column gives average-case (resp., worst-case) value.

As seen in insets (a) and (b) of Fig. 5, Algs. 2 and 3 were superior both in the average case and in the worst case. This data suggests that the extra overhead incurred under Alg. 4 and the slot-oriented waiting used under it have a detrimental impact in this scenario. Alg. 5 also performs poorly as the maintenance of D_i queues adds significant overhead.

Obs. 4. In the high-contention scenario, s-blocking was generally lower under Alg. 4 than under Algs. 2, 3, and 5.

The data in insets (c) and (d) suggests that cutting ahead had a significant positive impact. However, to reap a schedulability benefit from cutting ahead, it must be possible to *prove* that cutting ahead happens in the worst case. In our experiments, this is not possible, because we have no notion of a periodic or sporadic task. Our tasks were instead designed to create resource stress, with no useful work between resource accesses.

8 Conclusion

In this paper, we have considered real-time locking protocols for replicated resources where individual tasks may acquire multiple replicas. We distinguished between two variants of the considered problem, namely, replica assignment and replica allocation, and gave algorithms for solving each variant. In the case of allocation, we explored some of the tradeoffs that exist between protocol-related overheads and blocking times, and presented algorithms that are optimized with respect to one or the other. Finally, we intestigated these tradeoffs experimentally.

There are numerous avenues for future work. First, our experimental results pertain to observed runtime performance. Such experiments need to be complemented by studies that assess schedulability differences. Second, our allocation algorithms were motivated by two use cases, as described in Sec. 1. We intend to investigate both use cases more fully, as well as a third, which involves controlling acccess to the indivual processing elements within a single GPU (which can be considerable in number). Third, it would be desirable to obtain an allocation algorithm that both exhibits low overhead and avoids unnecessary blocking; however, this seems quite difficult to do. Finally, as noted earlier in Sec. 2, all of our spin-based algorithms have suspension-based counterparts. These counterparts warrant full consideration, as they would likely be preferrable in many settings (*e.g.*, mediating lengthy GPU accesses).

References

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

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

[3] G. Elliott. *Real-Time Scheduling of GPUs, with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2015.

[4] G. Elliott and J. Anderson. Exploring the multitude of real-time

multi-GPU configurations. In *RTSS '14*.

[5] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In *RTNS '11*.

[6] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[7] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to process failure. In *FOCS '79*.

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

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

[10] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson. Multiprocessor real-time locking protocols for replicated resources (full version). http://www.cs.unc.edu/∼anderson/papers.html.

[11] J. Owens. Towards multi-GPU support for visualization. Journal of Physics: Conference Series 78, 2007.

[12] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. Computer graphics forum, 2007.

[13] S. Schaetz and M. Uecker. A multi-GPU programming library for real-time applications. Algorithms and Architectures for Parallel Processing: Lecture Notes in Computer Science 7439, 2012.

[14] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *SOSP '87*.

[15] B. Ward. Relaxing resource-sharing constraints for improved hardware management and schedulability. In *RTSS '15*.

[16] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

[17] M. Yang, H. Lei, Y. Liao, and F. Rabee. PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi-GPU sharing under partitioned scheduling. In *DASC '13*.