Taking One for the Team: Trading Overhead and Blocking for Optimal Critical-Section Granularity with a Shared GPU

Tanya Amert
Carleton College
Northfield, Minnesota, USA
tamert@carleton.edu

Catherine E. Nemitz
Davidson College
Davidson, North Carolina, USA
canemitz@davidson.edu

ABSTRACT

Conventional wisdom maintains that the interval of time mutually exclusive access is granted to a shared resource (*i.e.*, in a *critical section*) should be as short as possible. However, the arbitration of shared-resource accesses introduces overhead. As a result, there exist task systems for which schedulability can only be guaranteed when accesses are grouped together into one large critical section, as opposed to many smaller sections. Furthermore, when the shared resource is a graphics processing unit (GPU), the durations of some GPU accesses decrease when multiple such accesses compose a single critical section, compared to when they are kept separate.

In this paper, an extended resource model is presented in which each critical section can comprise several grouped accesses, rather than each access forming its own critical section. This model reveals a trade-off between overhead and blocking: a low-priority task with resource accesses grouped into fewer critical sections incurs less overhead at the expense of greater blocking suffered by higher-priority tasks. An optimal algorithm for resolving this trade-off for a shared GPU under uniprocessor fixed-priority scheduling is presented, along with an experimental evaluation that demonstrates its benefits compared to the extremes of critical-section granularity.

CCS CONCEPTS

• **Computer systems organization** → **Real-time systems**; *Heterogeneous (hybrid) systems*; • **Software and its engineering** → **Mutual exclusion**; **Scheduling**.

KEYWORDS

Real-time systems, shared resources, graphics processing units, fixed-priority scheduling, schedulability

ACM Reference Format:
Tanya Amert and Catherine E. Nemitz. 2024. Taking One for the Team: Trading Overhead and Blocking for Optimal Critical-Section Granularity with a Shared GPU. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 6–8, 2024, Porto, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3696355.3696368

This work was supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, and ONR grant N00014-20-1-2698.

RTNS 2024, November 6–8, 2024, Porto, Portugal

ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3696368

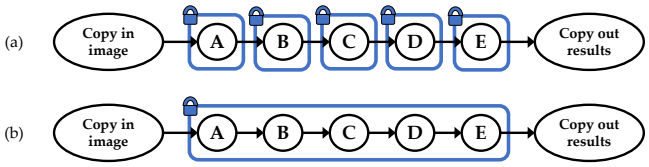

Figure 1: A GPU-based computer-vision algorithm comprising five kernels per image resolution, shown here for one resolution. The kernels can be treated as (a) individual critical sections, or (b) grouped into fewer, longer critical sections.

1 INTRODUCTION

A fundamental challenge in complex systems is the coordination of accesses to shared resources, such as in-memory data structures or shared hardware accelerators. When this coordination is performed via a mutual-exclusion synchronization protocol, if one task is accessing the shared resource then other tasks requiring access to that resource must wait. For this reason, conventional wisdom [4, 20] dictates that a *critical section*, *i.e.*, the computations performed while guaranteed sole use of a resource, should have as short of a duration as possible.

Following this guidance, multiple successive accesses to the same shared resource should each be treated as an individual critical section. Several successive accesses are common on heterogeneous platforms equipped with shared hardware accelerators; a typical access pattern for a computer vision (CV) application on a graphics processing unit (GPU) is to copy the input data to the GPU, execute a series of GPU computations called *kernels* on those data, and copy the results back to the CPU. An example is illustrated in Fig. 1 for a CV application comprising multiple kernels for each input image.

The grouping of kernels into critical sections impacts the duration of each individual kernel; the 99.9^{th}-percentile CPU-based measurements[1] of durations of kernels for the GPU application depicted in Fig. 1 are given in Tbl. 1 for two kernel-grouping configurations. If each kernel is treated as a critical section, as depicted in Fig. 1(a), then each kernel incurs both the synchronization protocol overhead as well as any GPU-related overhead, *e.g.*, due to cache affinity loss or GPU scheduling. If, however, multiple kernels are executed within the same critical section, as shown in Fig. 1(b), then durations of later kernels may be significantly reduced.[2]

[1]The full experiment is described in more detail in Sec. 5.1.
[2]Note that, as illustrated later in Fig. 5, kernel A is skipped for certain image resolutions. Thus, kernel B is the first kernel executed for some resolutions, and the 99.9^{th}-percentile measurements for kernel B do not decrease due to access grouping.

Table 1: Duration (in microseconds) of five GPU kernels, as 99.9^{th}-percentile measurements from the CPU.

Configuration	A	B	C	D	E
No grouping	150	152	191	150	182
All grouped	153	150	90	48	76

1.1 The Trade-Off: Overhead versus Blocking

The choice of critical section granularity thus exposes a trade-off. Separating discrete resource accesses, like GPU kernels, into unique critical sections leads to additional overhead. From the perspective of a single task, it is better to group accesses into longer critical sections: there is less time spent waiting to regain use of the resource and lower overhead for subsequent accesses.

Unfortunately, when one task is granted mutually exclusive use of a resource, another higher-priority task may suffer *priority-inversion blocking* (pi-blocking) if it must wait to use that resource. If resource accesses of one task are grouped into fewer, longer critical sections, higher-priority tasks may be subject to more pi-blocking, possibly making them unscheduleable due to missed deadlines.

However, for each task there is a (usually non-zero) amount of pi-blocking it can tolerate before becoming unscheduleable. If the added pi-blocking due to grouping lower-priority tasks' accesses into larger critical sections does not exceed this tolerance, then schedulability is not violated and lower-priority tasks experience less overhead and thus lower response times.

In this paper, we expose this trade-off via an extended task model that includes the durations of individual resource accesses rather than critical sections. Doing so enables schedulability to factor into the decision of how to group accesses into critical sections. We call such a grouping *valid* if schedulability can be guaranteed given that grouping, and an algorithm to group accesses into critical sections *optimal* if it is guaranteed to find a valid grouping if one exists. In this paper, we present such an optimal algorithm. Before detailing our contributions, we first introduce related work.

1.2 Related Work

A common approach to managing accesses to shared resources is via a synchronization protocol. For task systems scheduling based on fixed task priorities, the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) [22] provide mechanisms to ensure mutual exclusion, although the implementations require modifications to the kernel to support the inheritance of a blocked task's scheduling priority. Considering that some accesses are shorter than others, prior work has also sought to explore whether waiting by spinning or by suspending is more appropriate [10].

The possibility of incurred overhead from sources like synchronization protocols or lost cache affinity has led to *limited-preemptive scheduling*, which seeks a middle ground between the generally higher schedulability offered by fully preemptive systems and the predictability of non-preemptive systems. Under limited-preemptive scheduling, tasks may have regions of non-preemptive execution; if such regions are at least as long as any critical section, no synchronization protocol is needed to manage shared-resource use.

Previous limited-preemption approaches include arbitrating preemptions based on priority [16, 17, 21, 24] or allowing non-preemptive regions at either fixed locations (*e.g.*, deferring preemption until a given preemption point) [6, 11, 14] or floating locations for a known duration [4, 5, 25]; some prior work has explored a combination of models [12, 13, 15, 26]. Limited-preemptive scheduling has also been used to decrease energy usage [23]. Each of these prior works assumes that a task may execute non-preemptively for some duration, which may cause pi-blocking.

Unfortunately, non-preemptive execution may cause pi-blocking even for tasks that do not use any shared resources. Furthermore, non-preemptive execution on the CPU is not always the most appropriate approach. For example, while one task performs a computation on the GPU, another task may execute on the CPU.

In contrast, we consider fully preemptive tasks that are subject to the PIP. Unlike the non-preemptive regions of limited-preemption scheduling, under the PIP a resource-holding task is preemptable and pi-blocking is incurred only by some tasks in the system. The concept of a longest non-preemptive region from prior work [25, 26] provides a valuable starting point for determining the maximum duration of pi-blocking that each task can tolerate without an adverse impact on schedulability; our focus is on how to group resource accesses into critical sections of at most that duration.

1.3 Contributions

The contributions presented in this paper are threefold:

(1) We propose an extended task model that includes the duration of each shared-resource access and illustrate with simple examples how the choice of critical-section granularity impacts schedulability (Sec. 3).
(2) We leverage prior work to compute a per-task bound on critical-section durations such that schedulability is not violated and present an optimal algorithm for grouping accesses into critical sections (Sec. 4).
(3) We demonstrate via a schedulability study that our algorithm results in better schedulability than either grouping all accesses into a single critical section or performing each access within its own critical section (Sec. 5).

Although the work presented herein is motivated by GPU-using tasks, our approach is not specific to GPU-equipped platforms. Rather, our algorithm is applicable for any type of shared resource.

1.4 Organization

The remainder of this paper is organized as follows. In Sec. 2, we discuss needed background on fixed-priority scheduling and the PIP, as well as the results we leverage from prior work on limited-preemptive scheduling. We introduce our extended model in Sec. 3 and demonstrate the impact of critical-section granularity on schedulability via an example. In Sec. 4, we describe our access-grouping algorithm and prove its optimality. We detail our experimental evaluation in Sec. 5 before concluding in Sec. 6.

2 BACKGROUND

In this section, we discuss existing task and shared-resource models, along with the scheduling algorithm and analysis upon which we build our work.

2.1 Task Model

We consider a set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ of n tasks, where each task τ_i releases a (possibly infinite) sequence of jobs, $J_{i,1}, J_{i,2}, ...$; we refer to a generic job of task τ_i as $J_{i,j}$.

We consider *sporadic* tasks, *i.e.*, a period T_i specifies the lower bound on the separation between job releases of task τ_i. Each job $J_{i,j}$ must complete execution by a deadline $D_i \leq T_i$ time units after its release. In the absence of shared resources, we represent a task as the three-tuple $\tau_i = (T_i, C_i, D_i)$, where C_i is the worst-case execution time (WCET) of each job $J_{i,j}$. If $T_i = D_i$ (*i.e.*, for *implicit-deadline* tasks) we can simplify this to $\tau_i = (T_i, C_i)$.

2.2 Resource Model

Given GPU-equipped platforms as our motivation, we assume a single shared GPU as a shared resource.[3] When job $J_{i,j}$ requires use of the shared resource, it *issues* an *acquisition request* to a synchronization protocol managing access to that resource. Job $J_{i,j}$ *acquires* the resource once the request is satisfied, and then *holds* the resource (*i.e.*, is guaranteed mutually exclusive access to that resource) until it issues a *release request*.

The instructions executed while a job holds the resource comprise a *critical section*. As CV applications typically perform the same processing steps repeatedly on a sequence of input images, we thus represent a task as an alternating sequence of *unprotected sections* and *critical sections* of code. We let L_i^σ be the maximum duration of the σth critical section of job $J_{i,j}$, and define $L_i = \max_\sigma L_i^\sigma$.

Synchronization protocols incur overhead for both the *acquire* and *release* procedures. For simplicity, we define O as the total overhead added due to a single critical section. We assume that O contributes to blocking of other tasks and thus include it in L_i.

2.3 Fixed-Priority Scheduling

Given its simplicity and popularity [2], we focus on fixed-priority scheduling for uniprocessor platforms. For ease of notation, we assume that tasks are indexed in order of priority, *i.e.*, task τ_i has higher priority than task τ_{i+1}.

Let R_i be the worst-case response time for task τ_i, and let $hp(\tau_i)$ be the set of tasks with priority higher than that of task τ_i. Under fixed-priority scheduling, the execution requirement for tasks in $\{\tau_i\} \cup hp(\tau_i)$ in the time interval from 0 to t, $W_i(t)$, is given by

$$W_i(t) = C_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{t}{T_h} \right\rceil \cdot C_h. \tag{1}$$

Therefore, R_i is the smallest t such that $W_i(t) = t$ [18].

2.4 Limited-Preemptive Scheduling

Yao *et al.* [25, 26] used Eq. (1) to derive the extent to which a task could incur blocking without a deadline miss, termed the *blocking tolerance* of task τ_i and denoted β_i, as:

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i(t)\}, \tag{2}$$

where $\mathcal{TS}$ is the testing set of values of t up to $t = T_i$. This set can be reduced to integer multiples of periods [18] or further trimmed [7], still containing a pseudo-polynomial number of values to test.

[3]We leave to future work the extension to multiple resources.

The blocking tolerance can be used to determine highest possible pi-blocking that does not impact schedulability. In the context of limited-preemptive scheduling, pi-blocking is caused by non-preemptive regions of lower-priority tasks [25, 26]. When using a locking protocol to synchronize shared-resource usage, pi-blocking is caused by critical sections. We therefore let Q_i denote the upper bound on critical-section duration for task τ_i that maintains schedulability; we require $L_i \leq Q_i$. In Sec. 4.2 we detail how to compute Q_i given our extended task model.

2.5 Priority Inheritance Protocol

We utilize the Priority Inheritance Protocol (PIP) as our synchronization protocol [22].[4] The PIP enforces mutual exclusion by allowing a lower-priority resource-holding task to inherit the priority of a higher-priority task waiting to acquire the resource. In the absence of such a blocked task, higher-priority work that does not use the resource executes as normal, preempting the resource-holding task.

Using the PIP can cause two types of pi-blocking, described here in terms of a higher-priority task τ_i that may be blocked due to a lower-priority task.

Definition 2.1. Task τ_i experiences *direct blocking* while a lower-priority task holds the resource that τ_i requires.

Definition 2.2. Task τ_i experiences *push-through blocking* while a lower-priority task is executing with an inherited priority higher than that of τ_i from a task directly blocked on the resource.

Pi-blocking can be accounted for in schedulability analysis with a small update to Eq. (1) to include the worst-case pi-blocking, B_i, incurred by task τ_i [7]:

$$W_i(t) = B_i + C_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{t}{T_h} \right\rceil \cdot C_h. \tag{3}$$

We make use of the following properties of the PIP from Sha et al. [22], reworded to fit the terminology used in our work:

- **P1** A job $J_{i,j}$ can be blocked by a lower-priority job only if that job is executing within a critical section that could block $J_{i,j}$ (directly or with push-through blocking) when $J_{i,j}$ is released. (Lemma 1 in [22])
- **P2** A job $J_{i,j}$ can experience push-through blocking caused by priority inheritance for a specific resource only if that resource is used both by a job which has priority lower than that of $J_{i,j}$ and by a job which has or can inherit priority equal to or higher than that of $J_{i,j}$. (Lemma 4 in [22])
- **P3** A job $J_{i,j}$ can encounter blocking by at most one critical section per resource in the set of all the critical sections of lower or equal priority jobs which could block $J_{i,j}$ by using that resource (through either direct or push-through blocking). (Lemma 5 in [22])

We can simplify some of these properties for specific cases:

- **P4** A task cannot be blocked if no lower-priority task uses the resource.
- **P5** A task cannot be blocked if it does not use the resource and there is no higher-priority task that does use the resource.

[4]In a system with only a single resource, both the PIP and the PCP [22] perform identically.

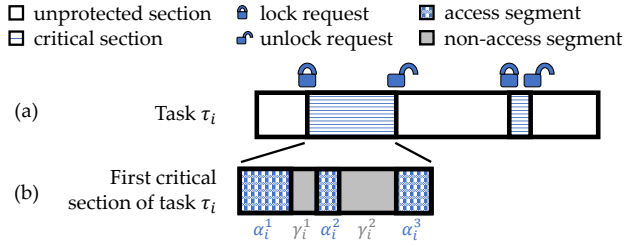

Figure 2: The decomposition of a task and a critical section.

In Sec. 4, we use these properties to calculate the bound on critical-section duration Q_i of each task τ_i. First, we introduce our extended task model that enables choice in critical-section granularities.

3 MODELING INDIVIDUAL RESOURCE ACCESSES

We now introduce our request model with the corresponding modifications to an extended task model. Then, we use example task sets to demonstrate the value of modeling resource accesses.

3.1 Request Model

The resource model discussed in Sec. 2.2 represents a job as an alternating sequence of unprotected sections and critical sections, as shown in Fig. 2(a). However, it is possible for a job to make multiple accesses to a shared resource while holding that resource. Therefore, we extend the model to expose the possibly multiple individual accesses that may occur during a single critical section.

We let α_i^v denote the vth resource access (for $v \geq 1$) made by a job $J_{i,j}$, and let $|\alpha_i^v|$ be the duration of α_i^v. We define γ_i^v as the corresponding vth non-access segment (with duration $|\gamma_i^v|$); γ_i^0 comprises instructions before α_i^1, and γ_i^v immediately follows α_i^v. A critical section is thus a contiguous sequence of access segments with interleaving non-access segments, as shown in Fig. 2(b).

3.2 Extended Task Model

Given this access-focused viewpoint, a task τ_i corresponds to the alternating sequence $\gamma_i^0, \alpha_i^1, \gamma_i^1, \alpha_i^2, \gamma_i^2, \ldots$. Any contiguous sequence of segments $\alpha_i^p, \gamma_i^p, \ldots, \alpha_i^q$ for $p \leq q$ can be grouped into a single critical section as long as no segment is part of two critical sections.

We now expand our task representation to the four-tuple $\tau_i = (T_i, \Gamma_i, A_i, D_i)$, where Γ_i is a list of the durations of non-access segments $\gamma_i^0, \gamma_i^1, \ldots$, and A_i is a list of the durations of access segments $\alpha_i^1, \alpha_i^2, \ldots$. If $T_i = D_i$, we can simplify our task representation to $\tau_i = (T_i, \Gamma_i, A_i)$. We let C_i be the summed durations of all segments in Γ_i and A_i, plus the overhead of each critical section, assuming a given grouping of accesses into critical sections. We denote by $|A_i|$ the number of access segments of task τ_i.

Example 3.1. Consider a task set comprising two tasks, which make one and three accesses, respectively, each for 10 time units: $\tau_1 = (140, [30, 30], [10])$ and $\tau_2 = (250, [20, 10, 20, 20], [10, 10, 10])$. There are four possible groupings of accesses by task τ_2 into critical sections.

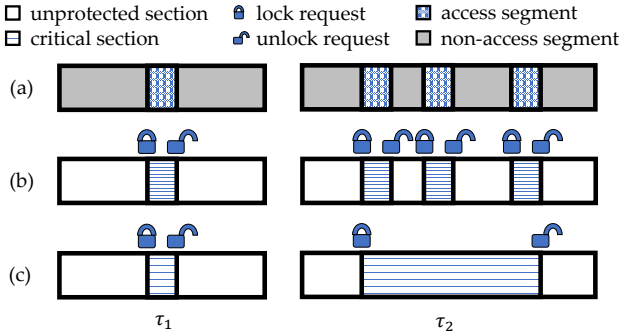

Figure 3: The task set described in Ex. 3.1: (a) divided into segments, (b) with one access per critical section, and (c) with all accesses grouped.

The task set from Ex. 3.1 is illustrated in Fig. 3, along with two possible groupings of each task's accesses into critical sections; we explore these two extremes of groupings next.

3.3 The Impact of Critical-Section Granularity on Schedulability

We now explore the impact of access grouping on schedulability using the example tasks introduced in Sec. 3.2. We will see that if each access composes its own critical section, then the schedulability test fails for this simple task set.

Example 3.2. Consider the same task set as in Ex. 3.1 and assume that no resource accesses are grouped, as illustrated in Fig. 3(b); each access composes its own critical section. For this system, we assume that the overhead is fixed, *i.e.*, does not depend on which task is acquiring or releasing the lock, with $O = 3$ time units.

Task τ_1 makes one access to the shared resource and thus has one critical section with duration $L_1 = O + |\alpha_1^1| = 3 + 10 = 13$ time units. Thus, $C_1 = |\gamma_1^0| + 13 + |\gamma_1^1| = 30 + 13 + 30 = 73$ time units. Similarly, each of the three critical sections of task τ_2 has duration $O + |\alpha_2^v| = 3 + 10 = 13$ time units and thus $C_2 = \sum_v |\gamma_2^v| + 3 \cdot 13 = (20 + 10 + 20 + 20) + 39 = 109$ time units.

Let task τ_1 have higher priority. Then, using Eq. (3) for task τ_1, we compute $W_1(t) = B_1 + C_1 = 13 + 73$, so $R_1 = 86$. As $R_1 < D_1 = 140$, all jobs of τ_1 meet their deadlines.

Task τ_2 has the lowest priority, so $B_2 = 0$. We solve $W_2(t) = 109 + \lceil \frac{t}{140} \rceil \cdot 73$ to get $R_2 = 255 > 250 = D_2$. The worst-case response time exceeds the deadline; the schedulability condition has been violated.

In Ex. 3.2, the overhead due to repeated acquisition and release requests results in a failed schedulability condition. Rather than task τ_2 having three critical sections, we could instead group these accesses into a single longer critical section.

Example 3.3. Consider the same task set as in Ex. 3.2, except that task τ_2 has all accesses grouped into a single critical section, as shown in Fig. 3(c). Including the time between accesses, this grouped critical section has duration 60 time units, plus overhead, for $L_2 = 63$ time units. Therefore, for task τ_1 we now have $W_1(t) =$

$B_1 + C_1 = 63 + 73$, so $R_1 = 136 < 140 = D_1$ and jobs of task τ_1 still meet their deadlines.

As task τ_2 has two fewer critical sections, it incurs less overhead, so we now have $C_2 = |\gamma_2^0| + L_2 + |\gamma_2^3| = 20 + 63 + 20 = 103$ time units. Again $B_2 = 0$, so we solve $W_2(t) = B_2 + C_2 + \left\lceil \frac{t}{T_i} \right\rceil \cdot C_1 = 0 + 103 + \left\lceil \frac{t}{140} \right\rceil \cdot 73$ to find $R_2 = 249 < 250 = D_2$; jobs of task τ_2 meet their deadlines, and the task set is therefore schedulable.

In Ex. 3.3, we showed that grouping accesses into a single critical section can make an otherwise unschedulable task set schedulable. We now consider an example that is only schedulable if accesses are *not* all grouped.

Example 3.4. Consider a task set comprised of two tasks, $\tau_1 = (130, [30, 30], [10])$ and $\tau_2 = (260, [20, 10, 20, 20], [10, 10, 10])$, with $O = 3$ time units. This is identical to the task set from Exs. 3.1–3.3 except that the tasks have different periods.

If all accesses are grouped, then $R_1 = 136 > 130 = D_1$; the task set is not schedulable. If no accesses are grouped, $R_1 = 86 < D_1$ and $R_2 = 255 < 260 = D_2$, and the task set is schedulable.

The above examples illustrate the challenge in the trade-off of overhead versus blocking incurred by grouping accesses into critical sections: simply always grouping or never grouping accesses will not be the best approach for all tasks systems.

Instead, we must base the decision of critical-section granularity on properties of a given task set. Like work for systems using limited-preemptive scheduling by Yao *et al.* [25, 26], we leverage the properties of fixed-priority scheduling to compute the longest possible critical section for each task, discussed next.

4 GROUPING ACCESSES INTO CRITICAL SECTIONS

We now introduce our approach for optimally grouping accesses into critical sections. We start in Sec. 4.1 by presenting our high-level algorithm that considers each task τ_i in priority order, using the bound Q_i to group accesses into critical sections. In Sec. 4.2 we extend previous work by Yao *et al.* [25, 26] to calculate each Q_i term, and in Sec. 4.3 we provide an optimal algorithm to group accesses for a given task τ_i. Finally, in Sec. 4.4, we show that the overall algorithm is optimal and discuss its computational complexity.

4.1 Algorithm Overview

Our goal is to combine accesses in a manner that maximally reduces overhead without causing any task to miss a deadline due to the pi-blocking caused by the (possibly larger) critical sections. Recall from the examples in Sec. 3.3 that changing the number of critical sections of τ_i (and thus the number of times overhead is incurred) also impacts C_i. We present in Alg. 1 an order for making the necessary computations, creating the critical sections, and updating the WCET values. We illustrate this with an example here before detailing specific computations in Secs. 4.2 and 4.3.

Example 4.1. Consider the same task set used in Exs. 3.1–3.3: $\tau_1 = (140, [30, 30], [10])$ and $\tau_2 = (250, [20, 10, 20, 20], [10, 10, 10])$, with $O = 3$. As both tasks make at least one access to the shared resource, the condition in Line 4 is true for both, and their accesses are grouped into critical sections.

Algorithm 1 Optimally grouping accesses into critical sections for each task τ_i, including the corresponding update to C_i.

```
1: procedure SetCriticalSections(τ: set of tasks ordered by decreasing priority,
                                 O: overhead)
2:     for i = 1 to n do                          ▷ Consider each task τ_i ∈ τ
3:         Compute Q_i using Thm. 4.5
4:         if A_i ≠ ∅ then
5:             criticalSections = GroupAccesses(α_i, γ_i, Q_i, O)
6:             if criticalSections == NULL then
7:                 return "Not schedulable"
8:             C_i += O · |criticalSections|
9:         Compute β_i using Eq. (1) and Eq. (2)
```

Alg. 1 starts with task τ_1, and computes $Q_1 = \infty$ (Line 3). Grouping accesses into critical sections occurs in Line 5; task τ_1 has a single access which is converted to a single critical section. The call to GroupAccesses is successful, so the check in Line 6 does not report the taskset unschedulable. The critical section choice results in $C_1 = 73$ (Line 8). The blocking τ_1 can tolerate is computed by Eq. (2) as $\beta_1 = \max_{t \in \mathcal{TS}(\tau_1)}\{t - W_1(t)\} = \max_{t \in \mathcal{TS}(\tau_1)}\{t - 73\} = 140 - 73 = 67$ (Line 9); this will be used when considering task τ_2.

For task τ_2, $Q_2 = 67$ (Line 3). The three accesses made by τ_2 can all be grouped into a single critical section with a duration of $L_2 = 63 < 67$ (Line 5). This results in $C_2 = 103$ (Line 8) and $\beta_2 = \max_{t \in \mathcal{TS}(\tau_2)}\{t - W_2(t)\} = \max_{t \in \mathcal{TS}(\tau_2)}\{t - (C_2 + \left\lceil \frac{t}{T_1} \right\rceil \cdot C_1)\} = 250 - (103 + \left\lceil \frac{250}{140} \right\rceil \cdot 73) = 1$ (Line 9).

As discussed in Ex. 3.3, this grouping of tasks' accesses into critical sections makes this task set schedulable.

Task WCETs depend on the number of critical sections, and are in turn used in the computations of Q_i and β_i. The order of these computations matters: accesses grouped using incorrect WCET values could result in missed deadlines or overly conservative critical sections. In Sec. 4.4, we return to this question of correctness and demonstrate that Alg. 1 performs calculations in the required order.

4.2 Computing the Longest Allowable Critical Sections

We leverage prior work on bounding the length of non-preemptive regions [25, 26]. In our context, however, tasks do not execute non-preemptively, so we instead account for sources of blocking (direct or push-through) that may be incurred when computing Q_i.

Let $\hbar$ (resp., ℓ) be the index of the highest-priority (resp., lowest-priority) task that accesses the resource. We first bound the maximum critical-section duration for each task with priority higher than or equal to that of $\tau_\hbar$ or lower than that of τ_ℓ.

Lemma 4.2. *The maximum critical-section duration for a task τ_i, $i \leq \hbar$, that maintains schedulability is unbounded. That is,*

$$\forall i, i \leq \hbar, Q_i = \infty.$$

Proof. By definition of $\hbar$, no task with higher priority than that of task τ_i uses the resource. Thus, by Rule P5, any task with priority higher than that of τ_i cannot experience pi-blocking. Therefore, for a task τ_i with $i \leq \hbar$, there is no constraint on Q_i, so $Q_i = \infty$. □

$k = \hbar$:	$L_{\hbar+1} \le \beta_\hbar$	$\wedge$	$L_{\hbar+2} \le \beta_\hbar$	$\wedge$	...	$\wedge$	$L_{\ell-1} \le \beta_\hbar$	$\wedge$	$L_\ell \le \beta_\hbar$
$k = \hbar + 1$:			$L_{\hbar+2} \le \beta_{\hbar+1}$	$\wedge$	...	$\wedge$	$L_{\ell-1} \le \beta_{\hbar+1}$	$\wedge$	$L_\ell \le \beta_{\hbar+1}$
...					...	$\wedge$	...	$\wedge$	...
$k = \ell - 2$:							$L_{\ell-1} \le \beta_{\ell-2}$	$\wedge$	$L_\ell \le \beta_{\ell-2}$
$k = \ell - 1$:									$L_\ell \le \beta_{\ell-1}$

Table 2: Expansion of expression $\bigwedge_{k<i\le\ell}(L_i \le \beta_k)$.

Lemma 4.3. *The maximum critical-section duration for a task τ_i, $i > \ell$, that maintains schedulability is unbounded. That is,*

$$\forall i, i > \ell, Q_i = \infty.$$

Proof. A task τ_i with $i > \ell$ does not use the resource (by the definition of ℓ). Therefore, no task with priority higher than that of τ_i could be blocked by τ_i (by Rule P1). Thus, there is no constraint on Q_i, so $Q_i = \infty$. □

We now bound the maximum duration of critical sections for tasks with priorities in between those of $\tau_\hbar$ and τ_ℓ. Recall that B_k is the maximum pi-blocking that τ_k may experience, and β_k is the maximum pi-blocking τ_k can tolerate; we must ensure $B_k \le \beta_k$.

Lemma 4.4. *The maximum critical-section duration for a task τ_i, $\hbar + 1 \le i \le \ell$, that maintains schedulability is given by*

$$Q_i = \min\{Q_{i-1}, \beta_{i-1}\}.$$

Proof. To bound the maximum critical-section duration for task τ_i, we start by focusing on possible pi-blocking experienced by a given task τ_k due to critical sections of any task τ_i for $\hbar+1 \le i \le \ell$. By Rule P1, task τ_k can only be blocked by a lower-priority task that uses the resource, so we consider $k < i \le \ell$. If task τ_k uses the resource, it could experience direct blocking. Additionally, if $\hbar \le k$ then task τ_k could experience push-through blocking, regardless of whether it uses the resource.

Because we consider only one resource, by Rule P3, task τ_k can be blocked by at most one critical section of task τ_i. Thus, the blocking task τ_k may incur is given by

$$B_k = \max_{k<i\le\ell} L_i.$$

Recall that by the definition of β_k, we must ensure that $B_k \le \beta_k$. Therefore, we require

$$\max_{k<i\le\ell} L_i \le \beta_k,$$

which can be rewritten as

$$\bigwedge_{k<i\le\ell} (L_i \le \beta_k).$$

The expression $L_i \le \beta_k$ is shown in Tbl. 2 for all relevant values of i for a given k; we utilize this table to shift our focus from a task τ_k that may experience pi-blocking to the constraints on the maximum critical-section duration L_i of one of those tasks τ_i that may block τ_k. Thus, for $\hbar + 1 \le i \le \ell$,

$$\bigwedge_{\hbar\le k<i} L_i \le \beta_k,$$

which yields

$$L_i \le \min_{\hbar\le k<i} \beta_k.$$

As we require $L_i \le Q_i$, we therefore have

$$Q_i = \min_{\hbar\le k<i} \beta_k. \tag{4}$$

Because $\min_{\hbar\le k<i} \beta_k$ is the smaller of $\min_{\hbar\le k<i-1} \beta_k$ and β_{i-1}, we therefore have

$$Q_i = \min\{\min_{\hbar\le k<i-1} \{\beta_k\}, \beta_{i-1}\}.$$

Substituting in Q_{i-1} using Eq. (4), the maximum critical-section duration for a task τ_i, $\hbar + 1 \le i \le \ell$, that maintains schedulability is given by

$$Q_i = \min\{Q_{i-1}, \beta_{i-1}\}.$$ □

Theorem 4.5. *The maximum critical-section duration for a task τ_i that maintains schedulability is*

$$Q_i = \begin{cases} \infty & i \le \hbar \\ \min\{Q_{i-1}, \beta_{i-1}\} & \hbar + 1 \le i \le \ell \\ \infty & i > \ell \end{cases}$$

Proof. This holds given Lem. 4.2, Lem. 4.3, and Lem. 4.4. □

Corollary 4.6. *Given Q_{i-1} and β_{i-1}, then Q_i can be computed in constant time.*

Now that we have shown how to compute Q_i, we present an algorithm for choosing how to group resource accesses into critical sections based on that bound on critical-section duration.

4.3 Choice of Critical-Section Granularity

Any combination of resource accesses into critical sections must ensure that $L_i \le Q_i$ for each task τ_i. As illustrated in Sec. 3.3, the overhead incurred for each critical section can affect schedulability. Minimizing the number of critical sections thereby minimizes the added overhead, resulting in shorter response times for τ_i and any lower-priority tasks. We now present an algorithm for aggregating resource accesses into critical sections and prove that our algorithm minimizes the number of critical sections.

4.3.1 Greedy Algorithm. As shown in Alg. 2, our algorithm greedily groups resource accesses into critical sections for task τ_i based on the provided set of non-access durations Γ_i, set of access durations A_i, maximum critical-section duration Q_i, and per-critical-section overhead O. Alg. 2 produces a mapping from a critical-section number to the corresponding access indices. If no mapping is possible, Alg. 2 returns `NULL`.

We illustrate the behavior of this algorithm with an example.

Example 4.7. Consider $\tau_i = (150, [3, 5, 2, 2, 1], [8, 2, 1, 6])$, as depicted in Fig. 4(a). We now trace a call to GroupAccesses(Γ_i, A_i, Q_i, O) assuming $Q_i = 20$ and $O = 1$. First, Alg. 2 checks if any

Algorithm 2 Greedy algorithm for grouping accesses into critical sections for task τ_i.

```
1:  procedure GroupAccesses(Γ_i, A_i, Q_i, O)
2:      for ν = 1 to |A_i| do ▷ verify that no individual access+overhead exceeds Q_i
3:          if O + |α_i^ν| > Q_i then
4:              return NULL
5:      criticalSections = {}        ▷ map critical section index to list of access indices
6:      σ = 1
7:      criticalSections[σ] = [1]        ▷ initialize first critical section to contain α_i^1
8:      currentDuration = O + |α_i^1|
9:      for ν = 2 to |A_i| do                                   ▷ consider access α_i^ν
10:         temp = currentDuration + |γ_i^{ν-1}| + |α_i^ν|
11:         if temp ≤ Q_i then
12:             criticalSections[σ] += [ν]
13:             currentDuration = temp              ▷ include α_i^ν in this critical section
14:         else
15:             σ += 1
16:             criticalSections[σ] = [ν]
17:             currentDuration = O + |α_i^ν|       ▷ put α_i^ν in a new critical section
18:     return criticalSections
```

single-access critical section would exceed Q_i (Lines 2–4); if a single access plus the overhead is larger than Q_i, it is not possible to assign accesses to critical sections, and the algorithm returns NULL.

Next, the algorithm initializes an empty map (Line 5) and assigns access α_i^1 to the first critical section (Lines 6–7). So far, we have $currentDuration = O + |\alpha_i^1| = 1 + 8 = 9$ (Line 8). We must account for one duration of overhead for each critical section; future accesses added to this same critical section do not incur additional overhead.

For each additional access, Alg. 2 checks if the next access (and the intermediate non-access segment) can be included in the current critical section (Lines 9–17). For $\nu = 2$, $temp = 9 + |\gamma_i^{\nu-1}| + |\alpha_i^\nu| = 9 + |\gamma_i^1| + |\alpha_i^2| = 9 + 5 + 2 = 16$ (Line 10). This does not exceed Q_i (Line 11), so access α_i^2 (and the intermediate non-access segment γ_i^1) are added to the critical section (Line 12) and its duration updated (Line 13). This process is repeated for $\nu = 3$, for which $temp = 16 + 2 + 1 = 19 \leq 20$; access α_i^3 is also added to the first critical section.

For $\nu = 4$, as $temp = 19 + 2 + 6 > 20$, access α_i^4 becomes the first access in a new critical section. Alg. 2 updates the critical section number (Line 15), adds this access to the map (Line 16), and sets the current size of this new critical section to $O + |\alpha_i^4| = 7$ (Line 17).

4.3.2 Proving the Greedy Algorithm is Optimal. We call an approach for assigning accesses to critical sections for a given task τ_i *optimal* if it results in the minimum number of critical sections such that no critical-section duration exceeds Q_i. We leverage this in Sec. 4.4 to show the optimality of Alg. 1.

In order to show that Alg. 2 is optimal, we compare a solution from our greedy algorithm to an arbitrary optimal solution to show that after the completion of each critical section, Alg. 2 has assigned at least as many resource accesses as the optimal solution. We illustrate this with an example and formalize this observation with Lem. 4.9 and its proof. Then, we show that this fact allows us to prove that the number of critical sections found with our greedy approach is at most the number found by any optimal solution, showing that the greedy algorithm is itself optimal.

Example 4.8 (cont'd). Note that for $Q_i = 20$ and $O = 1$ task τ_i must have at least two critical sections, as combining all accesses into one critical section would have $L_i = O + |\alpha_i^1| + |\gamma_i^1| + |\alpha_i^2| + |\gamma_i^2| + |\alpha_i^3| + |\gamma_i^3| + |\alpha_i^4| = 1 + 8 + 5 + 2 + 2 + 1 + 2 + 6 = 27 > Q_i$.

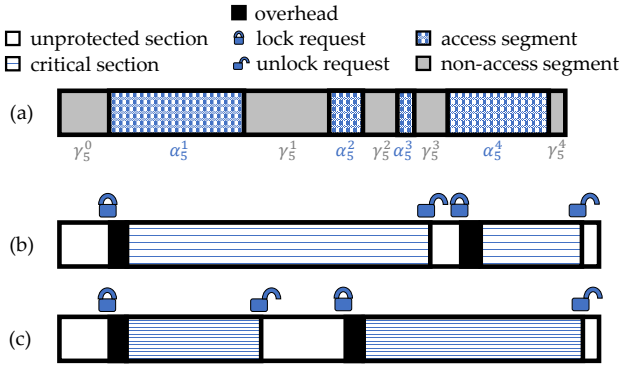

Figure 4: The task described in Ex. 4.7: (a) divided into segments, (b) with greedy grouping of accesses into critical sections, and (c) with an arbitrary optimal grouping.

As depicted in Fig. 4(b), Alg. 2 results in two critical sections; the first comprises segments α_i^1, γ_i^1, α_i^2, γ_i^2, and α_i^3. An arbitrary optimal grouping is depicted in Fig. 4(c), in which the first critical section contains only the first access: α_i^1. Thus, in the first critical section, our greedy approach has assigned at least as many resource accesses (three) to a critical section as the optimal solution (one). When comparing through the second critical section, this pattern continues: both Alg. 2 and the optimal solution have assigned all four accesses.

To formally reason about our greedy algorithm and an optimal algorithm, we define some additional notation. Let G be the result given by our greedy algorithm for task τ_i, and let H be an optimal result; that is, H gives the minimum number of critical sections and ensures that $L_i \leq Q_i$. Assume that critical sections are numbered in the order that they occur during the execution of the task. Let $|X_\sigma|$ denote the number of resource accesses included in the first σ critical sections of solution X.

Lemma 4.9. *The first σ critical sections produced by Alg. 2 include at least as many resource accesses as any arbitrary optimal algorithm. That is, $|G_\sigma| \geq |H_\sigma|$.*

Proof. For $\sigma = 1$, the first critical section of both G and H start with entry α_i^1. Suppose for the sake of contradiction $|H_1| > |G_1|$. Then the optimal solution was able to keep adding accesses to its first critical section that were not added to G. The sum of any access durations (plus the duration of intermediate non-access segments and the overhead) must be at most Q_i for this to be a valid solution. However, Alg. 2 would have also added these accesses, as it continues adding accesses until doing so would exceed Q_i (Line 11). Therefore $|G_\sigma| \geq |H_\sigma|$ for $\sigma = 1$.

Suppose that $|G_\sigma| \geq |H_\sigma|$ holds for all values of σ through $\sigma = x$. Next we show that this result holds through $\sigma = x + 1$. The last accesses included in the x^{th} and $(x+1)^{\text{th}}$ critical sections for H have indices $|H_x|$ and $|H_{x+1}|$, respectively. By the definition of H, the sum of the overhead plus the durations of accesses and intermediate non-accesses from $\alpha_i^{|H_x|+1}$ through $\alpha_i^{|H_{x+1}|}$ must be at most Q_i.

Because we assumed $|G_x| \geq |H_x|$, the sum of the overhead plus the durations of accesses and intermediate non-accesses from $\alpha_i^{|G_x|+1}$ to $\alpha_i^{|H_{x+1}|}$ must be at most Q_i as well, as this is a subset of the values summed for the optimal solution and each is non-negative. Therefore, Alg. 2 would have been able to add through access $|H_{x+1}|$ as well without exceeding Q_i. Thus, $|G_{x+1}| \geq |H_{x+1}|$, establishing that $|G_\sigma| \geq |H_\sigma|$ holds for all values of σ through $\sigma = x + 1$. □

This greedy stays-ahead proof shows that the greedy approach in Alg. 2 will always be able to include at least as many accesses through a given critical section as an optimal solution.

Theorem 4.10. *The greedy approach presented in Alg. 2 is optimal.*

Proof. Suppose for the sake of contradiction that there is a different assignment of accesses to critical sections that results in fewer total critical sections. Let ω be the index of the last critical section of H. Then, $|H_\omega| = |A_i|$; all accesses have been assigned by the final critical section. By Lem. 4.9, $|G_\omega| \geq |H_\omega|$, so by the ω^{th} critical section of G, G's critical sections have also included all resource accesses, and it is not possible that G contains an additional critical section compared to H. Thus, Alg. 2 is optimal. □

When assigning accesses to critical sections, minimizing the number of critical sections for a task minimizes its WCET.

Lemma 4.11. *Given Γ_i, A_i, Q_i, and O for task τ_i, Alg. 2 produces an assignment of accesses to critical sections that minimizes C_i.*

Proof. The computation of C_i depends only on Γ_i, A_i, and O. The choice of grouping does not change Γ_i or A_i. Because Alg. 2 creates the minimum number of critical sections (Thm. 4.10), it minimizes the addition of overhead O, thus minimizing C_i. □

We now look at the computational complexity of Alg. 2.

Theorem 4.12. *The running time of Alg. 2 for task τ_i is linear in the number of resource accesses, i.e., $O(|A_i|)$.*

Proof. The for-loop in Lines 2–4 iterates at most $|A_i|$ times, doing a constant amount of work in each iteration, resulting in $O(|A_i|)$. Lines 5–8 add a constant amount of work. The for-loop in Lines 9–17 iterates at most $|A_i| - 1$ times, with each iteration performing a constant amount of work. Thus, in total the second for-loop requires $O(|A_i| - 1) = O(|A_i|)$ operations. The return in Line 18 is $O(1)$. Thus, in total, Alg. 2 has a running time of $O(|A_i|) + O(1) + O(|A_i|) + O(1) = O(|A_i|)$. □

4.4 Optimality and Complexity

Now that we have provided details of the computations required for Alg. 1, we discuss its optimality and computational complexity. We begin by observing the order in which Alg. 1 performs computations and which values are required to compute the blocking bound.

Lemma 4.13. *The WCET values used in the computation of β_i are C_h for $h \leq i$.*

Proof. By Eq. (1) and Eq. (2), we compute β_i using only tasks τ_h such that $h \leq i$. □

Corollary 4.14. *The computations in Alg. 1 are performed in the appropriate order; that is, the values Q_i, C_i, and β_i for a given task τ_i depend only on values already computed.*

Proof. The computation of Q_i in Line 3 is either $\min\{Q_{i-1}, \beta_{i-1}\}$ or ∞ (Thm. 4.5); any required values were computed in the previous for-loop iteration. The computation of C_i in Line 8 depends only on the access grouping in Line 5, which depends only on Γ_i, A_i, Q_i, O, all of which are already known. Finally, β_i is computed in Line 9 based on C_i and WCETs computed previously (Lem. 4.13). □

Now that we have shown that Alg. 1 is correct, we show that it is optimal. Recall that an algorithm for grouping accesses is optimal if, given that a valid (i.e., schedulability-guaranteeing) grouping exists, the algorithm is guaranteed to find a valid grouping. We first consider how different WCET values, C_x and C'_x, for a single task τ_x impact the response time, the bound on critical-section length, and the bound on incurred blocking for all tasks in the task set.

Lemma 4.15. *Consider C_x and C'_x, where $C_x \leq C'_x$. Let $W'_i(t)$, Q'_i, and β'_i correspond to $W_i(t)$, Q_i, and β_i calculated with C'_x instead of C_x. Then $\forall i$, $Q_i \geq Q'_i$ and $\beta_i \geq \beta'_i$.*

Proof. For any $i < x$, C_x is not included in Eq. (1), so $W_i(t) = W'_i(t)$. Similarly, $Q_i = Q'_i$ and $\beta_i = \beta'_i$.

For $i = x$, $C_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{t}{T_h} \right\rceil \cdot C_h \leq C'_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{t}{T_h} \right\rceil \cdot C_h$, so $W_i(t) \leq W_i(t)'$. Thus, $Q_i \geq Q'_i$ and $\beta_i \geq \beta'_i$.

For $i > x$, we have $\tau_x \in hp(\tau_i)$. Because $\left\lceil \frac{t}{T_x} \right\rceil \cdot C_x \leq \left\lceil \frac{t}{T_x} \right\rceil \cdot C'_x$, $W_i(t) \leq W'_i(t)$, $Q_i \geq Q'_i$, and $\beta_i \geq \beta'_i$. □

We use this to show the optimality of Alg. 1.

Theorem 4.16. *Alg. 1 is optimal.*

Proof. Consider $i = 1$. Because τ_1 has the highest priority, it must have index $i \leq \hbar$ (by definition). Thus, in Line 3, $Q_1 = \infty$ (Thm. 4.5). If τ_1 does not access the shared resource, its WCET is unchanged. If $|A_1| > 0$ and a valid access grouping exists, Alg. 2 returns a grouping for τ_1 in Line 5. This ensures that the computation of C_1 in Line 8 results in the minimum possible WCET for τ_1 (Lem. 4.11). Because C_1 is minimized, Q_k is maximized for all $k \neq i$ (Lem. 4.15).

Suppose the values for Q_i, C_i, and β_i have been set for all values through $i = x$, with C_i minimized by the choice of access grouping. Thus, if a valid grouping exists, all tasks with $i \leq x$ have all jobs meeting their respective deadlines. Because WCETs of higher priority tasks have been minimized, Q_x is maximized (Lem. 4.15); when accesses are grouped into critical sections for τ_x in Line 5, this is done with the maximum possible Q_x. Thus, if a valid grouping exists, Alg. 2 will produce a grouping for τ_x that ensures no deadline misses for its jobs. As above, this grouping will minimize C_x (Lem. 4.11) and maximize Q and β for other tasks (Lem. 4.15).

Therefore, if a valid grouping exists, Alg. 1 will produce an access grouping for every task that ensures no deadline misses. □

Finally, we examine the computational complexity of Alg. 1.

Theorem 4.17. *Alg. 1 has pseudopolynomial running time.*

Proof. Alg. 1 begins in Line 2 by iterating over all n tasks ($O(n)$). For each task τ_i, Lines 3–9 are executed. Consider the first task, τ_1. The bound Q_1 is computed Line 3 with a constant-time operation ($Q_1 = \infty$, by Thm. 4.5). After a constant-time comparison in Line 4, any accesses by τ_1 are grouped into critical sections in Line 5, taking $O(|A_1|)$ time (Thm. 4.12). Then C_1 is computed in Line 8, taking constant time. Finally, β_1 is computed, which takes pseudopolynomial time [25, 26].

For all remaining tasks, the same computations occur, with the exception of Q_i, which may require the use of the expression $\min\{Q_{i-1}, \beta_{i-1}\}$ (Thm. 4.5). As the tasks are considered in increasing-index order, both Q_{i-1} and β_{i-1} will already have been computed in the previous iteration of the for-loop, so the computation of Q_i is also a constant-time operation (Cor. 4.6).

Thus, in total, Alg. 1 has pseudopolynomial running time. □

5 EXPERIMENTAL EVALUATION

We now discuss our experimental evaluation. We first detail our experiments that provide the motivation for this work, showing the possible decrease in access durations as a result of grouping accesses into a longer critical section. Then we present a schedulability study to evaluate the algorithms proposed in Sec. 4.

5.1 The Impact of Critical-Section Granularity on Access Durations

In prior work, we introduced a framework to enable temporal isolation for component-based workloads executing on CPU+GPU platforms [3]. That work acknowledged that multiple GPU accesses may be made during a critical section, but neglected to explore the impact of different access groupings.

When accesses are grouped into a critical section, the durations of the accesses themselves may decrease, as illustrated in Tbl. 1. These measurements were taken from a CV application executed on a CPU+GPU platform equipped with two eight-core 2.10-GHz Intel Xeon Silver 4110 processors and one NVIDIA Titan V GPU.

Note that in these experiments, both the scheduler and the synchronization protocol differ from the context we consider in this paper (G-EDF within component-based time partitions versus uniprocessor fixed priority, and a modified version of the OMLP [9] versus the PIP). However, the measurements were taken while HOG executed non-preemptively while holding the lock, and our focus in presenting them here is the impact on kernel-execution times given different groupings of kernels (accesses) into critical sections.

Our CV workload, illustrated in Fig. 5, is called Histogram of Oriented Gradients (HOG) [8]; a simplified depiction is given in Fig. 1. For each input image, this application resizes the image (kernel A) to one of several different resolutions, and then performs the remaining four computations on each resolution. In total, for 13 image resolutions there are 64 GPU kernels (one resolution doesn't require resizing) executed for each input image, along with a single copy-in of the image and 13 copy-out operations to retrieve results from the GPU.

We executed two instances of this application for 25000 images each, measuring the duration of each GPU kernel on the CPU using `clock_gettime()`, for two configurations: one in which each GPU access (78 including both copies and kernels) composed its own critical section, and another in which all kernels for a given resolution were grouped into one critical section and all copy-out operations were grouped (for a total of 1 + 13 + 1 = 15 critical sections). The CPU-based 99.9^{th}-percentile measurements of durations of individual accesses are reported in Tbl. 1.

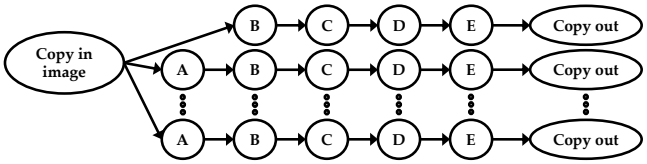

Figure 5: The GPU-based HOG algorithm comprises five kernels per image resolution (kernel A resizes the image, and is skipped for the original resolution).

As noted in Sec. 1, when accesses are grouped into one critical section, the duration of the first access (kernel A for most image resolutions, and kernel B for one) is relatively unchanged and the durations of subsequent accesses are significantly reduced by almost-identical amounts. Our extended task model allows us to represent systems with this type of behavior: we model the reduction in access durations as overhead introduced with each individual critical section. From the experiment summarized in Tbl. 1, we model the GPU-access overhead to be approximately 100 μs.

5.2 Trading Off Overhead and Blocking

We now present our experimental evaluation of the algorithms proposed in Sec. 4. We describe our experimental setup and then discuss the results.

5.2.1 Experimental Setup. We consider randomly generated task sets and a uniprocessor platform. We assume deadline monotonic (DM) scheduling, with task priorities assigned by non-increasing relative deadline, and that resource access is controlled by the PIP.

Our focus is evaluating the improvement in schedulability gained from optimally grouping resource accesses into critical sections. As such, we compare against two other approaches, named based on their behavior for each task τ_i:

- AlwaysCombine: Assigns all accesses α_i^v to one critical section; minimizes C_i without regard for schedulability impacts.
- NeverCombine: Assigns each access α_i^v to its own critical section; minimizes the blocking of higher-priority tasks.

We also compare against NOLOCK, in which resource accesses are ignored; this does not represent the behavior of the system (as accesses must be managed), but rather represents a theoretical upper-bound on possible performance. For example, at higher system utilizations, some task sets are not schedulable even if the synchronization-related overhead and blocking are ignored.

We utilized the SchedCAT [1] library to generate task systems and determine schedulability for each task system under the different access-grouping approaches. As our focus is demonstrating that the trade-off in critical-section granularity is a worthwhile one to consider, we present the results of schedulability experiments exploring a variety of *configurations*, *i.e.*, selections of the parameters shown in Tbl. 3. When using a given range, values were selected uniformly from that range on a per-task basis.

Table 3: Parameter Selections for Schedulability Study

Parameter	Selections
utilization u_i	light (0.001 – 0.1) medium (0.1 – 0.4)
period T_i	short (3 ms – 33 ms) moderate (10 ms – 100 ms)
deadline D_i	$(0.4 \cdot T_i - 0.6 \cdot T_i)$
overhead O	3 μs, 100 μs
access duration $\lvert\alpha_i^v\rvert$	short (1 μs – 15 μs) moderate (15 μs – 100 μs) gpu (10 μs – 200 μs)
WCET C_i (before adding O)	$u_i \cdot T_i$
goal number of accesses	4, 10
ratio of durations of access to interleaving non-access segments $\lvert\alpha_i\rvert/\lvert\gamma_i\rvert$	0.2, 1.0, or 2.0
fraction of tasks that require access to the resource	0.6, 0.8, or 1.0

The chosen values of $|\alpha_i|$ and the target number of accesses for a given task τ_i may result in an amount of time spent accessing the shared resource that exceeds C_i. Therefore, we capped the number of accesses for task τ_i such that AlwaysCombine would result in a single critical section with $L_i < 0.95 \cdot C_i$.

We generated accesses in a cluster centered temporally within the execution of a task, with the duration $|\gamma_i|$ of each interleaving non-access segment determined by the ratio $|\alpha_i|/|\gamma_i|$.

Given our motivation of exploring the impact of critical-section granularity for shared GPU accesses and the experiments discussed in Sec. 5.1, we chose $O = 100$ μs. To provide a point of comparison, we also considered a smaller overhead value of $O = 3$ μs, which is nearer to the overhead introduced by a variety of locking protocols [19].

5.2.2 Experimental Results. For each value of system utilization, we generated at least 100 task systems, and for each randomly chosen resource-accessing task in a given task system, we generated the accesses and then separately grouped them using one of the three grouping policies. Each data point in one of our resulting graphs indicates the fraction of task sets generated for that system utilization that were deemed schedulable using the execution-requirement computation from Eq. (3), assuming a specific grouping policy.

Before addressing our focus of GPU-inspired accesses, we first observe that for some systems, the granularity of critical sections may have no impact on schedulability.

Obs. 1. *For low-overhead configurations, the grouping of accesses into critical sections has minimal impact on schedulability.*

This is demonstrated by Fig. 6, which depicts the results for a configuration with $O = 3$ μs. When access-related overhead is significantly less than both the access durations and the period, then the difference in worst-case execution with one versus ten critical sections is insignificant. This is the case for many of the configurations we explored, including those with 3 μs overhead, non-"gpu" access durations, and even some with "gpu" access durations.

We now focus on the configurations for which schedulability is impacted by access groupings. Specifically, we look at the impact of critical-section granularity for accesses similar to those made to a GPU, using $O = 100$ μs. The results for two such configurations (differing in per-task utilization) are depicted in Figs. 7 and 8.

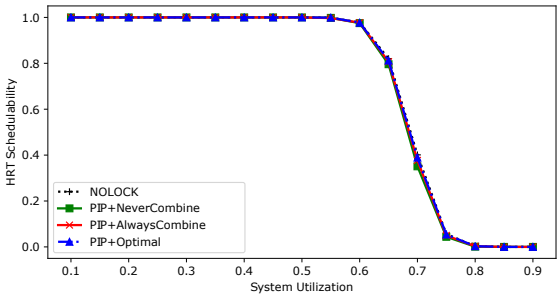

Figure 6: Schedulability given different access-grouping policies, for "short" access durations, "moderate" periods, "light" task utilizations, 3 μs overhead, with 100% of tasks accessing the shared resource with a goal of 4 accesses each, and with a ratio of 0.2 for $|\alpha_i|/|\gamma_i|$.

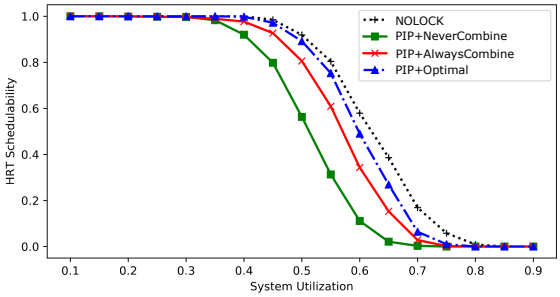

Figure 7: Schedulability given different access-grouping policies, for "gpu" access durations, "short" periods, "medium" task utilizations, 100 μs overhead, with 80% of tasks accessing the shared resource with a goal of 10 accesses each, and with a ratio of 2.0 for $|\alpha_i|/|\gamma_i|$.

Obs. 2. *For GPU-inspired configurations, coarse-grained access groupings generally result in higher schedulability than fine-grained access groupings.*

This can be seen by comparing the AlwaysCombine and NeverCombine curves in Fig. 7. For example, with a system utilization of 0.55, NeverCombine was schedulable for only 31.3% of generated task systems, whereas AlwaysCombine was deemed schedulable for 60.9% of generated task systems.

The trend in Obs. 2 was generally the case in the medium-task-utilization GPU-inspired configurations we explored. However, for systems with smaller per-task utilizations (and thus for which the generated task systems contain more tasks), we see that this trend no longer holds.

Obs. 3. *For GPU-inspired configurations with small task utilizations, neither AlwaysCombine nor NeverCombine dominates the other.*

This can be seen in comparing the AlwaysCombine and NeverCombine curves in Fig. 8. For total system utilizations greater than 0.4, the behavior is similar to that of Fig. 7. However, for lower system utilizations, NeverCombine resulted in higher schedulability than AlwaysCombine; for example, at a total system utilization of

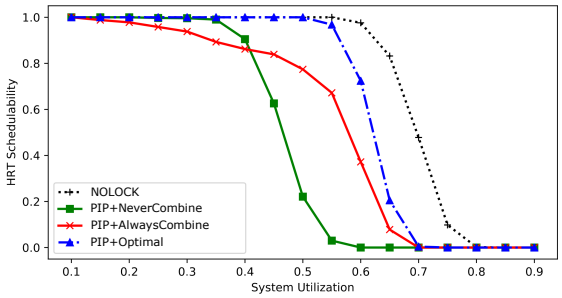

Figure 8: Schedulability given different access-grouping policies, for "gpu" access durations, "short" periods, "light" task utilizations, 100 μs overhead, with 80% of tasks accessing the shared resource with a goal of 10 accesses each, and with a ratio of 2.0 for $|\alpha_i|/|\gamma_i|$.

0.35, NeverCombine resulted in 99.0% of systems being schedulable, whereas AlwaysCombine resulted in only 89.3% being schedulable.

We conclude with a general observation.

Obs. 4. *The optimal grouping algorithm performs at least as well as both AlwaysCombine and NeverCombine.*

This can be seen in both Figs. 7 and 8. For example, in Fig. 8 for system utilization 0.55, our algorithm achieves 96.8% schedulability, whereas AlwaysCombine and NeverCombine achieve only 67.2% and 3.0% schedulability, respectively. Our algorithm performed at least as well the others for every configuration we explored.

6 CONCLUSION

In this paper, we have presented an extended resource model that considers as top-level entities the individual accesses made to shared resources, and demonstrated the trade-off between different choices of granularity when grouping the accesses into critical sections. To address this trade-off, we proposed an algorithm to determine the optimal grouping of accesses into critical sections for task systems under fixed-priority scheduling. Our algorithm requires only a single pass through tasks in decreasing-priority order, determining critical sections and updating the worst-case execution time of each task accordingly. We have presented a CV-based GPU experiment that showed decreased critical-section durations when accesses were grouped; we categorized this difference as overhead. We performed schedulability experiments to demonstrate the potential improvements in schedulability due to decreased overhead of one task at the expense of increased pi-blocking incurred by others.

In the future, we plan to extend our analysis to systems utilizing multiple shared resources and with accesses arbitrated by different synchronization protocols. We also plan to extend our work to multiprocessor platforms and to more accurately account for the overhead due to the synchronization mechanism or any loss of affinity with the shared resource. Finally, we will explore real-world task systems to more accurately model the distribution of accesses within a task, including the durations of the non-access segments.

REFERENCES

[1] 2024. SchedCAT: Schedulability test collection and toolkit. https://github.com/brandenburg/schedcat.

[2] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. 2022. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems* 58, 3 (2022), 358–398.

[3] Tanya Amert, Zelin Tong, Sergey Voronov, Joshua Bakita, F Donelson Smith, and James H Anderson. 2021. TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*. 455–468.

[4] Sanjoy Baruah. 2005. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. 137–144.

[5] Marko Bertogna and Sanjoy Baruah. 2010. Limited preemption EDF scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics* 6, 4 (2010), 579–591.

[6] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. 2011. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. 217–227.

[7] Enrico Bini and Giorgio C Buttazzo. 2004. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Comput.* 53, 11 (2004), 1462–1473.

[8] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[9] Björn B Brandenburg and James H Anderson. 2013. The OMLP family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems* 17, 2 (2013), 277–342.

[10] Björn B Brandenburg, John M Calandrino, Aaron Block, Hennadiy Leontyev, and James H Anderson. 2008. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?. In *Proocedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*. 342–353.

[11] Reinder J Bril, Johan J Lukkien, and Wim FJ Verhaegh. 2009. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems* 42 (2009), 63–119.

[12] Reinder J Bril, Martijn MHP van den Heuvel, Ugur Keskin, and Johan J Lukkien. 2012. Generalized fixed-priority scheduling with limited preemptions. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*. IEEE, 209–220.

[13] Reinder J Bril, Martijn MHP van den Heuvel, and Johan J Lukkien. 2013. Improved feasibility of fixed-priority scheduling with deferred preemption using preemption thresholds for preemption points. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. 255–264.

[14] Alan Burns. 1994. Preemptive priority-based scheduling: an appropriate engineering approach. In *Advances in Real-Time Systems*. 225–248.

[15] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. 2013. Limited preemptive scheduling for real-time systems: A survey. *IEEE Transactions on Industrial Informatics* 9, 1 (2013), 3–15.

[16] Uğur Keskin, Reinder J Bril, and Johan J Lukkien. 2010. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *Proceedings of the 15th IEEE Conference on Emerging Technologies & Factory Automation*. IEEE, 1–4.

[17] Jinkyu Lee and Kang G Shin. 2014. Preempt a job or not in EDF scheduling of uniprocessor systems. *IEEE Trans. Comput.* 63, 5 (2014), 1197–1206.

[18] John Lehoczky, Lui Sha, and Yuqin Ding. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*. 166–171.

[19] Catherine E Nemitz. 2021. *Efficient Synchronization for Real-Time Systems with Nested Resource Access*. Ph. D. Dissertation. University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

[20] Srikanth Ramamurthy. 1997. *A lock-free approach to object sharing in real-time systems*. Ph. D. Dissertation. University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

[21] John Regehr. 2002. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*. IEEE, 315–326.

[22] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185.

[23] Mohsen Shekarisaz, Mehdi Kargahi, and Lothar Thiele. 2024. Inter-Task Energy-Hotspot Elimination in Fixed-Priority Real-Time Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[24] Yun Wang. 1999. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*. IEEE, 328–335.

[25] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. 2009. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 351–360.

[26] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. 2011. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems* 47, 3 (2011), 198–223.