Leibniz Transactions on Embedded Systems

Volume 1 | Issue 2 | November 2014

ISSN 2199-2002

Published online and open access by
the European Design and Automation Association (EDAA) / EMbedded Systems Special Interest Group (EMSIG) and Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.
Online available at
http://www.dagstuhl.de/dagpub/2199-2002.

Publication date
November 2014

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier
10.4230/LITES-v001-i002

Aims and Scope
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Contact
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de
http://www.dagstuhl.de/lites

Contents

Regular Papers

Blocking Optimality in Distributed Real-Time Locking Protocols

Björn B. Brandenburg

Max Planck Institute for Software Systems (MPI-SWS)
Paul-Ehrlich-Straße G 26, 67663 Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Lower and upper bounds on the *maximum priority inversion blocking* (*pi-blocking*) that is generally unavoidable in *distributed* multiprocessor real-time locking protocols (where resources may be accessed only from specific synchronization processors) are established. Prior work on suspension-based *shared-memory* multiprocessor locking protocols (which require resources to be accessible from all processors) has established asymptotically tight bounds of $\Omega(m)$ and $\Omega(n)$ maximum pi-blocking under suspension-oblivious and suspension-aware analysis, respectively, where m denotes the total number of processors and n denotes the number of tasks. In this paper, it is shown that, in the case of distributed semaphore protocols, there exist two different task allocation scenarios that give rise to distinct lower bounds. In the case of *co-hosted* task allocation, where application tasks may also be assigned to synchronization processors (*i. e.*, processors hosting critical sections), $\Omega(\Phi \cdot n)$ maximum pi-blocking is unavoidable for some tasks under any locking protocol under both suspension-aware and suspension-oblivious schedulability analysis, where Φ denotes the ratio of the maximum response time to the shortest period. In contrast, in the case of *disjoint* task allocation (*i. e.*, if application tasks may not be assigned to synchronization processors), only $\Omega(m)$ and $\Omega(n)$ maximum pi-blocking is fundamentally unavoidable under suspension-oblivious and suspension-aware analysis, respectively, as in the shared-memory case. These bounds are shown to be asymptotically tight with the construction of two new distributed real-time locking protocols that ensure $O(m)$ and $O(n)$ maximum pi-blocking under suspension-oblivious and suspension-aware analysis, respectively.

2012 ACM Subject Classification Real-time systems, Synchronization
Keywords and phrases distributed multiprocessor real-time systems, real-time locking, priority inversion, blocking optimality
Digital Object Identifier 10.4230/LITES-v001-i002-a001
Received 2013-08-29 **Accepted** 2014-06-11 **Published** 2014-09-12

1 Introduction

The principal purpose of a real-time locking protocol is to provide tasks with mutually exclusive access to shared resources such that the maximum blocking incurred by any task can be bounded *a priori*. Such blocking is problematic in real-time systems and must be bounded because it increases worst-case response times, and hence may cause deadline violations if left unchecked. Real-time locking protocols should thus avoid blocking as much as possible. Unfortunately, if tasks require exclusive access, some blocking is inherently possible and can generally not be avoided. This naturally raises the question of optimality: if some blocking is inevitable when using locks, then what is the *minimal* bound on worst-case blocking that *any* locking protocol can guarantee? In other words, when can a real-time locking protocol be deemed (asymptotically) optimal?

This question has long been answered for uniprocessor systems [2, 44, 48], where it has been shown that the real-time mutual exclusion problem can be solved with $O(1)$ maximum blocking: the *priority ceiling protocol* (PCP) [44, 48] and the *stack resource policy* (SRP) [2] both ensure

that the maximum blocking incurred by any task is bounded by the length of a single (outermost) critical section, which is obviously optimal.

In the multiprocessor case, the picture is not as straightforward, and not as complete. First of all, there are two classes of multiprocessor locking protocols to consider: *spin-based* (or *spin lock*) protocols, in which waiting tasks remain scheduled and execute a delay loop, and *suspension-based* (or *semaphore*) protocols, in which waiting tasks suspend to make the processor available to other tasks. Of the two classes, spin locks are much simpler to analyze: with non-preemptive FIFO spin locks, a lock acquisition is delayed by at most one critical section on each other processor [23, 29], and it is easy to see that this cannot be improved upon in the general case.

In the case of multiprocessor real-time semaphore protocols, however, the question of blocking optimality is considerably more challenging, and has only recently been answered in part [11, 16, 18, 49]. In particular, it has been answered only for the case of *shared-memory* multiprocessor semaphore protocols, which fundamentally require shared resources to be accessible from all processors because they assume that tasks execute critical sections locally on the processor(s) on which they are scheduled. In this paper, we extend the theory of blocking optimality to *distributed* multiprocessor semaphore protocols, which are required if (some) shared resource(s) can be accessed only from specific (subsets of) processors.

1.1 Motivation

Besides the fact that the restriction to shared-memory systems in prior work is an obvious limitation, our work is motivated by the observation that there are many systems that either inherently require, or at least can benefit from, distributed real-time locking protocols.

For instance, in the absence of a shared memory or on heterogeneous hardware platforms (*e. g.*, if only some processor cores support special-purpose instructions), the execution of critical sections can be inherently restricted to specific processors. Similarly, when tasks share physical resources such as network links, I/O co-processors, graphics processing units (GPUs), or digital signal processors (DSPs), certain devices may be accessible only from specific processors.

Second, even if all processors technically could access all shared resources, it sometimes is preferable to centralize resource access nonetheless. For example, many shared-memory multicore processors intended for embedded systems are not cache-consistent (*e. g.*, Infineon's Aurix platform for automotive applications does not support hardware-based cache coherency). On such a platform, the coherency of shared data structures either must be managed in software (thus introducing an additional implementation burden), or alternatively the execution of critical sections can simply be centralized on a dedicated processor with the help of a distributed real-time locking protocol. In fact, even on a cache-consistent shared-memory platform, it can be beneficial to centralize the execution of critical sections due to cache affinity issues [39]. Furthermore, the use of distributed real-time locking protocols in shared-memory systems can also yield improved schedulability [14].

As the final example, consider *multi-kernel* operating systems [6, 51], where each core is managed as a uniprocessor and system-wide resource management is carried out using message passing. Multi-kernels tend to aggressively optimize locality—intuitively, they form a "distributed system on a chip"—with the effect that some resources may be accessed only on specific cores.

In each of these examples, the critical sections of some tasks must be executed on a specific remote processor, since executing them locally is either infeasible or disallowed. This renders shared-memory semaphore protocols as studied in [11, 16, 18, 49] inapplicable, and a distributed real-time semaphore protocol must be employed instead.

Naturally, as in the uniprocessor and shared-memory cases, distributed real-time locking protocols should minimize blocking to the extent possible. However, to the best of our knowledge, blocking optimality in distributed real-time locking protocols has not been studied to date, and it

is thus not even clear what the minimal "extent possible" is, nor is it known how protocols should be structured to obtain (asymptotically) optimal blocking bounds. In this paper, we seek to close this gap in the understanding of multiprocessor real-time synchronization.

1.2 Related Work

The first discussion of the effects of uncontrolled blocking in real-time systems and possible solutions dates back to the Mesa project [36]. Sha *et al.* [48] were the first to study the problem from an analytical point of view and proposed uniprocessor protocols that provably bound the worst-case blocking duration. As already mentioned, the Sha *et al.*'s PCP [48] and Baker's SRP [2] were the first uniprocessor semaphore protocols to ensure optimal blocking bounds.

In the first work on synchronization in multiprocessor real-time systems, Rajkumar *et al.* [45] proposed the *distributed priority ceiling protocol* (DPCP) [44, 45] for partitioned[1] multiprocessors, which applies the PCP on each processor and uses "agents" to carry-out critical sections on behalf of tasks assigned to remote processors. As the first and prototypical distributed real-time semaphore protocol, the DPCP is central to this paper and reviewed in greater detail in Section 2.2.2. Rajkumar also developed the first suspension-based shared-memory real-time locking protocol, namely the *multiprocessor priority ceiling protocol* (MPCP) [43],[2] an extension of the PCP for partitioned shared-memory multiprocessors based on priority queues. Like the DPCP, the MPCP uses the regular PCP for *local* resources (*i. e.*, resources used on only one processor), but when accessing *global* resources (*i. e.*, resources used by tasks on multiple processors), tasks execute critical section on their assigned processor in the MPCP (rather than delegating resource access to "agents" as in the DPCP). In contrast to the PCP and the SRP, which are obviously optimal on a uniprocessor, the MPCP and the DPCP were not studied from a blocking optimality perspective.

Favoring spin locks over semaphores, Gai *et al.* [28, 29] developed the MSRP, a multiprocessor extension of the SRP for partitioned shared-memory multiprocessors, wich employs non-preemptive FIFO spin locks for global resources and the SRP for local resources; Devi *et al.* [23] similarly analyzed non-preemptive FIFO spin locks in the context of globally scheduled multiprocessors.[3] As already pointed out, it is not possible to construct spin lock protocols that ensure, in the worst case, asymptotically less blocking to all tasks than the protocols by Gai *et al.* [28, 29] and Devi *et al.* [23], although it is possible to use priority-ordered spin locks [32, 33, 41] to ensure that some tasks are less susceptible to blocking than others [52].

In subsequent work on shared-memory real-time locking protocols (both spin-based and suspension-based), numerous new protocols, analysis improvements, and evaluations have been presented [10, 11, 14, 17, 19, 20, 22, 24, 26, 27, 35, 40, 42, 47]; however, in contrast to this paper, they are not primarily concerned with questions of blocking optimality.

Perhaps more closely related are two studies targeting different notions of optimality. Soon after the MPCP was proposed, Lortz and Shin [38] observed that ordering conflicting critical sections by scheduling priority, as in the MPCP, does not always yield the best results in terms of schedulability, and proposed using FIFO queues or semaphore-specific locking priorities instead. They further showed that assigning per-semaphore locking priorities that maximize schedulability

[1] Under partitioned scheduling, each task is statically assigned to a processor, and each processor is scheduled individually using a uniprocessor policy.

[2] The name "multiprocessor priority ceiling protocol" originally referred to the DPCP in [45], but was later repurposed to refer to the MPCP in [43]. We follow the terminology from [35, 43, 44], wherein the MPCP denotes the shared-memory variant.

[3] Under global scheduling, all processors serve a shared ready queue and tasks migrate among all processors.

is an NP-complete problem [38]. More recently, Hsiu *et al.* [31] studied three problems related to finding task and resource assignments that minimize system-wide resource usage (*i. e.*, the number of processors hosting real-time tasks, the number of processors hosting shared resources, and the total number of processors) assuming a distributed, priority-queue-based semaphore protocol similar to the DPCP. Unsurprisingly, they found the exact optimization problems to be intractable (NP-hard). In contrast to Hsiu *et al.*'s work [31], the notion of optimality studied herein focuses on the locking protocol itself (and not system-wide allocation properties), which makes it possible to find simple, asymptotically optimal solutions, as we show in Section 5.

Most closely related to this paper is [16], which was the first work to consider blocking optimality in (shared-memory) multiprocessor real-time systems, and from which we adopt the analytical framework and key definitions (as reviewed in detail in Section 2.3). In short, it was shown that even in the shared-memory case alone, there exist not only one, but two lower bounds on maximum blocking [16]. This is because there exist two sets of analysis assumptions, termed *suspension-aware* and *suspension-oblivious* schedulability analysis, that yield different lower bounds due to differences in how semaphore-related suspensions are accounted for during schedulability analysis. More precisely, in a system with m processors and n tasks, a lower bound of $\Omega(n)$ was established in the suspension-aware case, whereas the suspension-oblivious case yields a lower bound of $\Omega(m)$. In other words, it was shown that there exist pathological scenarios in which some tasks incur blocking that is (at least) linear in the number of processors (under suspension-oblivious) or linear in the number of tasks (under suspension-aware analysis), regardless of the employed locking protocol. These bounds have further been shown to be asymptotically tight with the construction of practical shared-memory semaphore protocols that ensure bounds on maximum blocking that are within a small constant factor of the established lower bounds [11, 13, 16, 18, 25, 49, 50].

To the best of our knowledge, no equivalent results are known for the case of distributed multiprocessor real-time semaphore protocols.[4]

1.3 Contributions

We study the question of optimal blocking in distributed multiprocessor real-time semaphore protocols and show that there exist two distinct task allocation strategies, which we call *co-hosted* and *disjoint* task allocation, that lead to *different* lower bounds on blocking. In the disjoint case, synchronization processors are dedicated exclusively to executing critical sections and may not host real-time tasks, whereas in the co-hosted case tasks also execute on synchronization processors. Notably, in a co-hosted scenario, we observe two surprising results:

1. in terms of the lower bound, there is no difference between suspension-aware and suspension-oblivious analysis, in contrast to the shared-memory case; and
2. blocking can be asymptotically worse than in an equivalent shared-memory system by a factor of Φ, where Φ denotes the ratio of the maximum response time and the minimum period (formalized in Section 2)—we establish $\Omega(\Phi \cdot n)$ as a lower bound on maximum blocking under both suspension-oblivious and suspension-aware analysis (Theorem 8).

We further show that any "reasonable" distributed locking protocol that does not artificially delay requests (formalized as "weakly work-conserving" in Section 2) causes at most $O(\Phi \cdot n)$ blocking (Theorem 10); any "reasonable" protocol is hence asymptotically optimal in the co-hosted case.

[4] The material presented herein was previously made available online in preliminary form as an unpublished manuscript [12]. Based on [12], an in-kernel implementation and a fine-grained linear-programming-based blocking analysis of the protocol presented in Section 5.1 was previously discussed and evaluated in [14]. Whereas [14] focuses on accurate (non-asymptotic) analysis and practical concerns, the material presented in Sections 3–5 pertains exclusively to questions of optimality and has previously not been published.

In contrast to the co-hosted case, we show that, under disjoint task allocation, distributed locking protocols exist that ensure blocking bounds analogous to the shared-memory case: we establish lower bounds of $\Omega(n)$ and $\Omega(m)$ under suspension-aware and suspension-oblivious analysis, respectively, and show these bounds to be asymptotically tight by constructing two new distributed real-time semaphore protocols that ensure $O(n)$ and $O(m)$ maximum blocking under suspension-aware and suspension-oblivious analysis, respectively (Theorems 12 and 14).

The remainder of the paper is organized as follows. Section 2 provides essential definitions and a detailed review of the needed background. Section 3 establishes a lower bound on blocking with the construction of a task set that exhibits pathological blocking under co-hosted task allocation, and argues that prior constructions apply in the case of disjoint task allocation. Section 4 considers the co-hosted case and shows that any "reasonable" distributed locking protocol without artificial delays ensures maximum blocking within at most a constant factor of the established lower bound. Section 5 pertains to the case of disjoint task allocation and introduces two new protocols that establish the asymptotic tightness of the lower bounds under suspension-oblivious and suspension-aware analysis. Finally, Section 6 concludes with a discussion of the impact of communication delays.

2 Background and Definitions

In this section, we establish required definitions and review key prior results. In short, the results presented in this paper apply to sets of sporadic real-time tasks with arbitrary deadlines that are provisioned on a multiprocessor platform comprised of non-uniform processor clusters. Shared resources are accessible only from select synchronization processors and may be accessed from other processors using *remote procedure calls* (RPCs). These assumptions are formalized as follows; a summary of our notation is subsequently given at the end of the section in Table 1.

2.1 System Model

We consider the problem of scheduling a set of n sporadic real-time tasks $\tau = \{T_1, \ldots, T_n\}$ on a set of m processors. A sporadic task T_i is characterized by its *minimum inter-arrival separation* (or *period*) p_i, its per-job *worst-case execution time* e_i, and its *relative deadline* d_i, where $e_i \leq \min(d_i, p_i)$. Each task releases a potentially infinite sequence of jobs, where two consecutive jobs of a task T_i are released at least p_i time units apart.

We let J_i denote an arbitrary job of task T_i. A job is *pending* from its release until it completes, and while it is pending, it is either *ready* and may be scheduled on a processor, or *suspended* and not available for scheduling. A job J_i released at time t_a has its *absolute deadline* at time $t_a + d_i$. Both tasks and jobs are sequential: each job can be scheduled on at most one processor at a time, and a newly released job cannot be processed until the task's previous job has been completed.

A task's *maximum response time* r_i describes the maximum time that any J_i remains pending. A task T_i is *schedulable* if it can be shown that $r_i \leq d_i$; the set of all tasks τ is schedulable if each $T_i \in \tau$ is schedulable. We define Φ to be the ratio of the maximum response time and the minimum period; formally $\Phi \triangleq \frac{\max_i\{r_i\}}{\min_i\{p_i\}}$.

The set of m processors consists of K pairwise disjoint *clusters* (or sets) of processors, where $2 \leq K \leq m$. We let C_j denote the jth cluster, and let m_j denote the number of processors in C_j, where $\sum_{j=1}^{K} m_j = m$. A common special case is a *partitioned* system, where $K = m$ and $m_j = 1$ for each C_j. However, in general, clusters do *not* necessarily have a uniform size. We preclude the special case of $K = 1$ and $m_1 = m$ because distributed locking protocols are relevant only if there are at least two clusters (*i. e.*, the case of $K = 1$ and $m_1 = m$ corresponds to a globally scheduled shared-memory platform, which is already covered by prior work [11, 15, 16, 18]).

For notational convenience, we assume that clusters are indexed in order of non-decreasing cluster size: $m_j \leq m_{j+1}$ for $1 \leq j < K$. In particular, m_1 denotes the (or one of the) smallest cluster(s) in the system (with ties broken arbitrarily). Since $K \geq 2$, we have $m_1 \leq \frac{m}{2}$. This fact is exploited by the lower-bound argument in Section 3.

Each task T_i is statically assigned to one of the K clusters; we let $C(T_i)$ denote T_i's assigned cluster. Each cluster is scheduled independently according to a work-conserving *job-level fixed-priority* (JLFP) scheduling policy [21]. Two common JLFP policies are *fixed-priority* (FP) scheduling, where each task is assigned a fixed priority and jobs are prioritized in order of decreasing task priority, and *earliest-deadline first* (EDF) scheduling, where jobs are prioritized in order of decreasing absolute deadlines (with ties broken arbitrarily).

In general, a JLFP scheduler assigns each pending job a fixed priority and, at any point in time, schedules the m_j highest-priority ready jobs (or agents, see below) in each cluster C_j. Jobs may freely migrate among processors belonging to the same cluster (*i. e.*, global JLFP scheduling is used within each cluster), but jobs may not migrate across cluster boundaries. Note that this model includes the partitioned scheduling of shared-memory systems (each processor forms a singleton cluster). Each cluster may use a different JLFP policy. Our results apply to any JLFP policy.

Next, we discuss how resources may be shared in the assumed system architecture.

2.2 Distributed Real-Time Semaphore Protocols

In many real-time systems, tasks may have to share serially reusable resources (*e. g.*, co-processors, I/O ports, shared data structures, *etc.*). This paper is concerned with systems in which mutually exclusive access to such resources is governed by a distributed (binary) semaphore protocol. In a distributed semaphore protocol, each resource can be accessed only from a (set of) designated processor(s); critical sections must hence be executed remotely if tasks use resources that are not local to their assigned processor.

We next formalize the assumed resource model and review a distributed semaphore protocol.

2.2.1 Resource Model

The tasks in τ are assumed to share n_r resources (besides the processors). Each shared resource ℓ_q (where $1 \leq q \leq n_r$) is *local* to exactly one of the K clusters (but can be accessed from any cluster using RPC invocations). We let $C(\ell_q)$ denote the cluster to which ℓ_q is local. Cluster $C(\ell_q)$ is also called the *synchronization cluster* for ℓ_q.

To allow tasks to use non-local resources, access to each shared resource is mediated by one or more *resource agents*. To use a shared resource ℓ_q, a job J_i invokes an agent on cluster $C(\ell_q)$ to carry out the request on J_i's behalf using a synchronous RPC. After issuing an RPC, J_i suspends until notified by the invoked agent that the request has been carried out. A *locking protocol* such as the DPCP (reviewed in Section 2.2.2) determines how concurrent requests are serialized.

We let $N_{i,q}$ denote the maximum number of times that any J_i uses ℓ_q, and let $L_{i,q}$ denote the corresponding per-request *maximum critical section length*, that is, the maximum time that the agent handling J_i's RPC requires exclusive access to ℓ_q as part of carrying out any single operation invoked by J_i. For notational convenience, we require $L_{i,q} = 0$ if $N_{i,q} = 0$ and define $L^{max} \triangleq \max\{L_{i,q} \mid 1 \leq q \leq n_r \ \wedge \ T_i \in \tau\}$.

Jobs invoke at most one agent at any time, and agents do not invoke other agents as part of handling a resource request (*i. e.*, resource requests are not nested). An agent is *active* while it is processing requests, and *inactive* otherwise. While active, an agent is either *ready* (and can be

scheduled) or *suspended* (and is not available for execution). Active agents are typically ready, but may suspend temporarily when serving a request that involves synchronous I/O operations.

Following Rajkumar *et al.* [44, 45], we assume that jobs can invoke agents without significant delay. That is, we assume that the overhead of cluster-to-cluster communication is negligible, in the sense that any practical system overheads can be incorporated into task parameters using standard overhead accounting techniques (*e. g.*, see [11, Ch. 7]). If a distributed locking protocol is implemented on top of a platform with dedicated point-to-point links, or if the maximum communication delay across a shared network can be bounded by a constant (*e. g.*, when communicating over a time-triggered network [34]), this assumption is appropriate, as any constant invocation cost can be accounted for using standard overhead accounting techniques. Further, such communication delays do not affect the blocking analysis *per se* (*i. e.*, they do not affect the contention for shared resources) and thus can be ignored when deriving asymptotic bounds. We revisit the issue of non-negligible communication delays in Section 6.

Finally, in a real system, there likely exist resources in each cluster that are shared only among local tasks. Such *local resources* can be readily handled using shared-memory protocols (or uniprocessor protocols) and are not the subject of this paper. We hence assume that each resource ℓ_q is accessed by tasks from at least two different clusters.

Given our resource model, a locking protocol is required to determine how agents are prioritized, how conflicting requests are ordered, and when jobs may invoke agents. We next review the classic protocol for this purpose, namely the DPCP.

2.2.2 The Distributed Priority Ceiling Protocol

As the first (distributed) real-time semaphore protocol for multiprocessors, the DPCP [44, 45] can be considered to be the prototypical distributed semaphore protocol for *partitioned fixed-priority* (P-FP) scheduling, a special case of the clustered JLFP scheduling assumed in this paper. We briefly review the DPCP as a concrete example of the considered class of protocols.

The DPCP fundamentally requires $m_j = 1$ for each cluster (or, rather, partition) C_j. Each resource ℓ_q is statically assigned to a specific processor and may not be directly used on other processors. Rather, tasks residing on other processors must indirectly access the resource by issuing RPCs to resource agents. To this end, the DPCP provides one resource agent $A_{q,i}$ for each resource ℓ_q and each task T_i. To ensure a timely completion of critical sections, resource agents are subject to *priority boosting*, which means that they have priorities higher than any regular task (and thus cannot be preempted by regular jobs). Nonetheless, under the DPCP, resource agents acting on behalf of higher-priority tasks may still preempt agents acting on behalf of lower-priority tasks. That is, an agent $A_{q,h}$ may preempt another agent $A_{r,l}$ if T_h has a higher priority than T_l. After a job has invoked an agent, it suspends until its request has been carried out.

On each processor, conflicting accesses are mediated using the PCP [44, 48]. The PCP assigns each resource a *priority ceiling*, which is the priority of the highest-priority task (or agent) accessing the resource, and, at runtime, maintains a *system ceiling*, which is the maximum priority ceiling of any currently locked resource. A job (or agent) is permitted to lock a resource only if its priority exceeds the current system ceiling. Waiting jobs/agents are ordered by effective scheduling priority, and priority inheritance [44, 48] is applied to prevent unbounded "priority inversion" (Section 2.3).

From an optimality point of view, not all of the details of the DPCP are relevant. Therefore, we abstract from the specifics of the DPCP in our analysis to consider a larger class of "DPCP-like" protocols, as defined next.

2.2.3 Simplified Protocol Assumptions

Specifically, in this paper, we focus on the class of distributed real-time locking protocols that ensure progress by means of two properties adopted from the DPCP [44, 48].

A1 Agents are priority-boosted: agents always have a higher priority than regular jobs.

A2 The distributed locking protocol is *weakly work-conserving*: a resource request $\mathcal{R}$ for a resource ℓ_q is unsatisfied at time t (*i. e.*, $\mathcal{R}$ has been issued but is not yet being processed) only if *some* resource (but not necessarily ℓ_q) is currently unavailable (*i. e.*, some agent is currently processing a request for any resource).

Assumption A1 is necessary to expedite request completion since excessive delays cannot generally be avoided if jobs can preempt agents. Assumption A2 rules out pathological protocols that "artificially" delay requests. We consider this form of work conservation to be "weak" because it does not require the *requested* resource to be unavailable; a request for an available resource may also be delayed if some *other* resource is currently in use. Notably, the DPCP is only weakly work-conserving (and not work-conserving w.r.t. each resource) since requests for available resources may remain temporarily unsatisfied due to ceiling blocking [44, 48].

Assumptions A1 and A2 together ensure that any delay in the processing of resource requests can be attributed exclusively to other resource requests.

Another simplification pertains to the use of agents. Under the DPCP, jobs do not require agents to access resources local to their assigned processor since jobs can directly participate in the PCP. In a sense, this can be seen as jobs taking on the role of their agent on their local processor. To simplify the discussion in this paper, we assume herein that resources are accessed *only* via agents (*i. e.*, jobs invoke agents even for resources that happen to be local to their assigned processors). This does not change the algorithmic properties of the DPCP.

Finally, we assume that there is only a single local agent for each resource. As seen in the DPCP [44, 45], it can make sense to use more than one agent per resource; however, in the following, we abstract from such protocol specifics and let a single agent A_q represent all agent activity corresponding to a resource ℓ_q.

A key assumption in our system model is that both tasks and resources are statically assigned to clusters, which gives rise to two allocation scenarios, as we discuss next.

2.2.4 Co-Hosted and Disjoint Task Allocation

Processor clusters that host resource agents are called *synchronization clusters*. Conversely, processor clusters that host sporadic real-time tasks are called *application clusters*.

In this paper, we establish asymptotically tight lower and upper blocking bounds on maximum blocking in two separate scenarios, which we refer to as "co-hosted" and "disjoint" task allocation, respectively. Under *co-hosted task allocation*, the set of application clusters overlaps with the set of synchronization clusters, that is, there exists a cluster that hosts both tasks and agents. In contrast, under *disjoint task allocation*, clusters may host either agents or tasks, but not both. The significance of these two allocation strategies is that they give rise to two distinct lower bounds on worst-case blocking, as will become apparent in Section 3.

Next, we give a precise definition of what actually constitutes "blocking."

2.3 Priority Inversion Blocking

The sharing of resources subject to mutual exclusion constraints inevitably causes some delays because conflicting concurrent requests must be serialized. Such delays are problematic in a real-time system if they lead to an increase in worst-case response times (*i. e.*, if they affect

some r_i). Conversely, delays that do not affect r_i are not considered to constitute "blocking" in real-time systems. This is captured by the concept of *priority inversion* [44, 48], which, intuitively, exists if a job that should be scheduled according to its base priority is not scheduled, either because it is suspended (while waiting to gain access to a shared resource) or because a job or agent with elevated effective priority prevents it from being scheduled. To avoid confusion with other interpretations of the term "blocking" (*e. g.*, in an OS context, "blocking" often is used synonymously with suspending), the term *priority inversion blocking* (*pi-blocking*) denotes any resource-sharing-related delay that affects worst-case response times [16]. We let b_i denote a bound on the *maximum pi-blocking* incurred by any job of task T_i.

2.3.1 Suspension-Oblivious vs. Suspension-Aware Analysis

Prior work has shown that there exist in fact two kinds of priority inversion [16], depending on how suspensions are accounted for by the employed schedulability analysis. The difference arises because many published schedulability tests simply assume the absence of self-suspensions, which are notoriously difficult to analyze (*e. g.*, see [46]), and thus ignore a major source of pi-blocking. Such *suspension-oblivious* (*s-oblivious*) schedulability tests can still be employed to analyze task systems that exhibit self-suspensions, but require pi-blocking to be accounted for pessimistically by inflating each execution requirement e_i by b_i prior to applying the schedulability test. This results in sound, but likely pessimistic results: over-approximating all pi-blocking as additional processor demand is safe because converting execution time to suspensions does not increase the response time of any task (under preemptive JLFP scheduling), but is also likely pessimistic as the processor load is lower in practice than assumed during analysis.

As an example of an s-oblivious schedulability test, consider Liu and Layland's classic uniprocessor EDF utilization bound for implicit-deadline tasks: a set of *independent* sporadic tasks τ is schedulable under EDF on a uniprocessor if and only if $\sum_{T_i \in \tau} \frac{e_i}{p_i} \leq 1$ [37]. This test is s-oblivious because tasks are assumed to be independent (*i. e.*, there are no shared resources) and because jobs are assumed to always be ready (*i. e.*, there are no self-suspensions). However, even if these assumptions are violated (*i. e.*, if $b_i > 0$ for some T_i), Liu and Layland's utilization bound can still be used after inflating all execution costs e_i by the maximum pi-blocking bounds b_i [11, 16, 18]. That is, in the presence of locking-related self-suspensions, a set of resource-sharing, implicit-deadline sporadic tasks τ is schedulable under EDF on a uniprocessor if $\sum_{T_i \in \tau} \frac{e_i + b_i}{p_i} \leq 1$.

While s-oblivious schedulability analysis may at first sight appear too pessimistic to be useful, it is still relevant because some of the pessimism can actually be "reused" to obtain less pessimistic pi-blocking bounds [11, 16, 18], and because many published multiprocessor schedulability tests (*e. g.*, [3–5, 7–9, 30]) do not account for self-suspensions explicitly.

In contrast, *suspension-aware* (*s-aware*) schedulability analysis explicitly accounts for all effects of pi-blocking. For instance, *response-time analysis* (RTA) for (uniprocessor) FP scheduling [1, 35] is a good example of effective s-aware schedulability analysis, and can be applied to partitioned scheduling as follows. Let b_i^r denote a bound on maximum *remote* pi-blocking (*i. e.*, pi-blocking caused by tasks or agents assigned to remote clusters), and let b_i^l denote a bound on maximum *local* pi-blocking (*i. e.*, pi-blocking caused by tasks or agents assigned to cluster $C(T_i)$), where $b_i = b_i^r + b_i^l$. Then, assuming constrained deadlines (*i. e.*, $d_i \leq p_i$), a task T_i's maximum response time r_i is bounded by the smallest positive solution to the recursion [1, 35]

$$r_i = e_i + b_i^r + b_i^l + \sum_{T_h \in hp(T_i)} \left\lceil \frac{r_i + b_h^r}{p_h} \right\rceil \cdot e_h, \tag{1}$$

where $hp(T_i)$ denotes the set of tasks assigned to processor $C(T_i)$ with higher priorities than T_i. Equation (1) is an s-aware schedulability test because $b_i = b_i^r + b_i^l$ is explicitly accounted for.

This difference—explicit vs. implicit suspension accounting—has a profound impact on the exact nature of pi-blocking, as we review next.

2.3.2 S-Oblivious and S-Aware PI-Blocking

From the point of view of schedulability analysis, a priority inversion exists if a job is delayed (*i. e.*, not scheduled) and this delay *cannot* be attributed to the execution of a higher-priority job.[5] Prior work [11, 16, 18] has shown that, since s-oblivious schedulability analysis over-approximates a task's processor demand, the definition of "priority inversion" depends on the type of analysis.

▶ **Definition 1.** Under **s-oblivious** schedulability analysis, a job J_i of a task T_i assigned to cluster $C_j = C(T_i)$ incurs *s-oblivious pi-blocking* at time t if J_i is pending but not scheduled and fewer than m_j higher-priority jobs of tasks assigned to C_j are **pending** [16].

▶ **Definition 2.** Under **s-aware** schedulability analysis, a job J_i of a task T_i assigned to cluster $C_j = C(T_i)$ incurs *s-aware pi-blocking* at time t if J_i is pending but not scheduled and fewer than m_j higher-priority ready jobs of tasks assigned to C_j are **scheduled** [16].

Note that there cannot be fewer *pending* higher-priority jobs than there are *scheduled* higher-priority jobs (*i. e.*, a scheduled job is necessarily also pending). Hence, if a job J_i incurs s-oblivious pi-blocking at a time t, then it incurs also s-aware pi-blocking at time t. However, the converse does not hold: if J_i incurs s-aware pi-blocking time t, then it may be the case that it does *not* incur s-oblivious pi-blocking at time t. More precisely, J_i incurs s-aware pi-blocking, but not s-oblivious pi-blocking, at time t if there are at least m_j higher-priority jobs pending, but fewer than m_j of them are scheduled at time t.

In other words, if Definition 1 is satisfied, then Definition 2 is satisfied as well. Therefore, an upper bound on s-aware pi-blocking (Definition 2) implies an upper bound on s-oblivious pi-blocking (Definition 1), as previously pointed out in [16]. Conversely, a lower bound on s-oblivious pi-blocking (Definition 1) also implies a lower bound on s-aware pi-blocking (Definition 2). We use this relationship in Section 3.

From a practical point of view, the difference between s-oblivious and s-aware pi-blocking suggests that it is useful to design locking protocols specifically for a particular type of analysis. From an optimality point of view, which we review next, the difference between s-oblivious and s-aware pi-blocking is fundamental because—in shared-memory systems—the two types of analysis have been shown to yield two *different* lower bounds on the amount of pi-blocking that is unavoidable under any locking protocol [11, 16].

2.3.3 PI-Blocking Complexity

As discussed in Section 1, blocking optimality is concerned with finding the smallest possible bound on worst-case blocking. To enable systematic study of this question, *maximum pi-blocking*, formally $\max\{b_i \mid T_i \in \tau\}$, has been proposed as a metric of blocking complexity in prior work [11, 16, 18].

Concrete bounds on pi-blocking must necessarily depend on each $L_{i,q}$—long requests will cause long priority inversions under any protocol. Similarly, bounds for any reasonable protocol grow linearly with the maximum number of requests per job. Thus, when deriving asymptotic bounds, we consider, for each T_i, $\sum_{1 \leq q \leq n_r} N_{i,q}$ and each $L_{i,q}$ to be constants and assume $n \geq m$. All other parameters are considered variable (or dependent on m and n).

[5] Regular interference due to the scheduling of higher-priority jobs is accounted for by any sound schedulability test. A priority inversion exists if *additional* delay is incurred.

Table 1 Summary of notation.

Symbol	Definition	Symbol	Definition
m	total number of processors	n_r	number of shared resources
K	number of clusters, $2 \leq K \leq m$	ℓ_q	the qth shared resource, $1 \leq q \leq n_r$
C_j	the jth cluster, $1 \leq j \leq K$	A_q	the agent handling requests for ℓ_q
m_j	number of processors in C_j	$C(\ell_q)$	cluster to which ℓ_q is local
n	total number of tasks	$N_{i,q}$	max. number of requests of any J_i for ℓ_q
T_i	the ith sporadic task, $1 \leq i \leq n$	$L_{i,q}$	max. critical section length of T_i w.r.t. ℓ_q
J_i	a job of T_i	L^{max}	max. $L_{i,q}$ for any T_i and any ℓ_q
e_i	T_i's WCET	b_i	max. pi-blocking incurred by any J_i
p_i	T_i's period		
d_i	T_i's relative deadline	Φ	ratio of the longest max. response time of
r_i	T_i's max. response time		any T_i and the shortest period of any T_i
$C(T_i)$	T_i's assigned cluster		

Under these assumptions, it was shown [11, 16, 18] that, in the case of shared-memory locking protocols, the lower bound on unavoidable pi-blocking depends on whether s-oblivious or s-aware schedulability analysis is employed. More specifically, it was shown that there exist pathological task sets such that maximum pi-blocking is linear in the number of processors m (and independent of the number of tasks n) under s-oblivious analysis, but linear in n (and independent of m) under s-aware analysis [11, 16, 18]. Further, it was shown that these bounds are asymptotically tight with the construction of shared-memory semaphore protocols that ensure for *any* task set maximum pi-blocking that is within a constant factor of the established lower bounds. In other words, in the case of *shared-memory* semaphore protocols, the real-time mutual exclusion problem can be solved such that $\max\{b_i \mid T_i \in \tau\} = \Theta(m)$ under s-oblivious schedulability analysis, and such that $\max\{b_i \mid T_i \in \tau\} = \Theta(n)$ under s-aware schedulability analysis [11, 16, 18].

We can now precisely state the contribution of this paper: in the following sections, we establish upper and lower bounds on $\max\{b_i \mid T_i \in \tau\}$ under s-oblivious and s-aware schedulability analysis for distributed (*i. e.*, DPCP-like) real-time locking protocols, thereby complementing the earlier results on shared-memory (*i. e.*, MPCP-like) real-time locking protocols [11, 16, 18]. For ease of reference, the notation used in this paper is summarized in Table 1.

3 Lower Bounds on Maximum PI-Blocking

We start by establishing a lower bound on maximum s-oblivious and s-aware pi-blocking in the case of co-hosted task allocation. To establish a general lower bound, it is sufficient to construct an example task set that demonstrates that the claimed amount of pi-blocking (either s-aware or s-oblivious) is always possible under *any* locking protocol compliant with Assumptions A1 and A2. To this end, we establish the existence of pathological task sets in which some task always incurs $\Omega(\Phi \cdot n)$ pi-blocking due to priority boosting (Assumption A1), regardless of whether s-oblivious or s-aware schedulability analysis is used. This family of task sets is defined as follows.

▶ **Definition 3.** For a given smallest cluster size m_1, a given number of tasks n (where $n \geq m \geq 2 \cdot m_1$), and an arbitrary positive integer parameter R (where $R \geq 1$), let $\tau^{seq}(n, m_1, R) \triangleq \{T_1, \ldots, T_n\}$ denote a set of n periodic tasks, with parameters as given in Table 2, that share one resource ℓ_1 local to cluster C_1 (*i. e.*, $C(\ell_1) = C_1$).

Table 2 Parameters of the tasks in $\tau^{seq}(n, m_1, R)$, where $a = 1$ and $b = 2$ if $m_1 > 1$, and $a = \frac{1}{2}$ and $b = 1$ if $m_1 = 1$. Tasks $T_1, \ldots, T_{m_1}$ are assigned to the first cluster C_1; all other tasks are assigned in a round-robin fashion to clusters other than C_1. Recall that $n \geq m \geq 2 \cdot m_1$.

e_i	p_i	d_i	$N_{i,1}$	$L_{i,q}$	$C(T_i)$	
$\frac{R \cdot n}{2}$	$R \cdot n$	$R \cdot n$	0	0	C_1	for $i \in \{1, \ldots, m_1\}$
a	n	$n+1$	1	b	$C_{(2+i \bmod (K-1))}$	for $i \in \{m_1+1, \ldots, 2 \cdot m_1\}$
a	n	$n+1$	1	a	$C_{(2+i \bmod (K-1))}$	for $i > 2 \cdot m_1$ (if any)

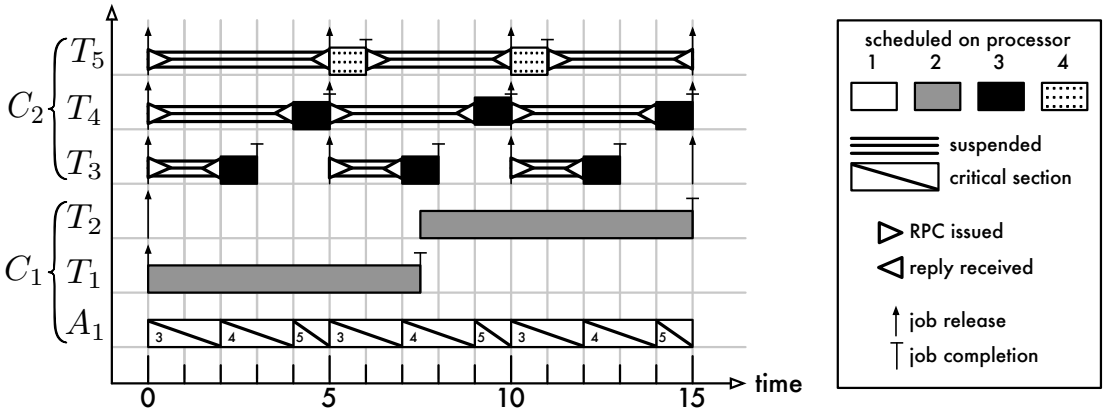


Figure 1 Example schedule of the task set $\tau^{seq}(n, m_1, R)$ as specified in Table 2 for $K = 2$, $m_1 = 2$, $m_2 = 2$, $n = 5$, and $R = 3$. There are five tasks $T_1, \ldots, T_5$ assigned to $K = 2$ clusters sharing one resource ℓ_1, which is local to cluster C_1. Agent A_1 is hence assigned to cluster C_1. The small digit in each critical section signifies the task on behalf of which the agent is executing the request. Deadlines have been omitted from the schedule for the sake of clarity. By construction, the scheduling policy employed to schedule jobs is irrelevant (for simplicity, assume FP scheduling, where lower-indexed tasks have higher priority than higher-indexed tasks). The response-time of T_2 is $r_2 = n \cdot R = 5 \cdot 3 = 15$ since it has the lowest priority in its assigned cluster C_1, and because agent A_1 is continuously occupying a processor.

The task set $\tau^{seq}(n, m_1, R)$ depends on the smallest cluster size m_1 because, by construction, the maximum pi-blocking will be incurred by tasks in cluster C_1. Note in Table 2 that the maximum critical section lengths (w.r.t. ℓ_1) depend on m_1, which is required to accommodate the special case of $m_1 = 1$. We first consider the case of $m_1 > 1$.

In the following, we assume a synchronous periodic arrival sequence, that is, each task T_i releases a job at time zero and periodically every p_i time units thereafter. We consider periodic tasks (and not sporadic tasks) in this section because it simplifies the example, and since periodic tasks are a special case of sporadic tasks and thus sufficient to establish a lower bound.

For simplicity and without loss of generality, we further assume that each job of tasks $T_{m_1+1}, \ldots, T_n$ immediately accesses resource ℓ_1 as soon as it is allocated a processor (*i. e.*, at the very beginning of the job). This results in a pathological schedule in which tasks $T_{m_1+1}, \ldots, T_n$ are serialized. Figure 1 depicts an example schedule for $K = 2$, $m_1 = 2$, $m_2 = 2$, $n = 5$, and $R = 3$.

We begin by observing that the agent servicing requests for ℓ_1, denoted A_1 in the following, continuously occupies one of the processors in cluster C_1.

▶ **Lemma 4.** *If $m_1 > 1$, then only $m_1 - 1$ processors of cluster C_1 service jobs of tasks $T_1, \ldots, T_{m_1}$.*

Proof. By construction, the agent A_1 servicing requests for resource ℓ_1 is located in cluster C_1. By Assumption A1, when servicing requests, agent A_1 preempts any job of $T_1, \ldots, T_{m_1}$. By Assumption A2, and since there exists only a single shared resource, agent A_1 becomes active as soon as a request for ℓ_1 is issued. Thus, a processor in C_1 is unavailable for servicing jobs of tasks $T_1, \ldots, T_{m_1}$ whenever A_1 is servicing requests issued by jobs of tasks $T_{m_1+1}, \ldots, T_n$.

Consider an interval $[t_a, t_a + n)$, where $t_a = x \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous, periodic arrival sequence, tasks $T_{m_1+1}, \ldots, T_n$ each release a job at time t_a. Upon being scheduled, each such job immediately accesses resource ℓ_1 and suspends until its request is serviced. As a result, regardless of the JLFP policy used to schedule jobs, A_1 is active during $[t_a, t_a + n)$ for the cumulative duration of all requests issued by jobs of tasks $T_{m_1+1}, \ldots, T_n$ released at time t_a. Assuming each request requires the maximum time to service, agent A_1 is thus active for a duration of

$$\sum_{i=m_1+1}^{n} N_{i,1} \cdot L_{i,1} = \sum_{i=m_1+1}^{2m_1} N_{i,1} \cdot L_{i,1} + \sum_{i=2m_1+1}^{n} N_{i,1} \cdot L_{i,1} = \sum_{i=m_1+1}^{2m_1} 2 + \sum_{i=2m_1+1}^{n} 1 = n$$

time units during the interval $[t_a, t_a + n)$, regardless of how the employed locking protocol serializes requests for ℓ_1. Hence, only $m_1 - 1$ processors are available to service jobs of $T_1, \ldots, T_{m_1}$ during the interval $[t_a, t_a + n)$. Since such intervals are contiguous (as $t_a = x \cdot n$ and $x \in \mathbb{N}$), one processor in C_1 is continuously unavailable to jobs of $T_1, \ldots, T_{m_1}$ under any JLFP scheduling policy and any distributed locking protocol satisfying assumptions A1 and A2. ◀

This in turn implies that the execution of one of the jobs of tasks $T_1, \ldots, T_{m_1}$ is delayed.

▶ **Lemma 5.** *If $m_1 > 1$, then* $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$.

Proof. Consider an interval $[t_a, t_a + R \cdot n)$, where $t_a = x \cdot R \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous, periodic arrival sequence, tasks $T_1, \ldots, T_{m_1}$ each release a job at time t_a. Regardless of the (work-conserving) JLFP policy employed to assign priorities to jobs, one of these m_1 jobs will have lower priority than the other $m_1 - 1$ ready pending jobs in cluster C_1. Recall that we assume that priorities are unique (*i. e.*, any ties in priorities are subject to arbitrary but consistent tie-breaking). Let J_l denote this lowest-priority job. By Lemma 4, there are only $m_1 - 1$ processors available to service jobs. Thus J_l will only be scheduled after one of the other jobs has finished execution. Since each task assigned to cluster C_1 has a worst-case execution time of $e_i = \frac{R \cdot n}{2}$, in the worst case, job J_l is not scheduled until time $t_a + \frac{R \cdot n}{2}$, and then requires another $e_l = \frac{R \cdot n}{2}$ time units of processor service to complete. Hence, $\max\{r_i \mid 1 \leq i \leq m_1\} = 2e_i = R \cdot n$. ◀

So far we have considered only the case of $m_1 > 1$. By construction, the same maximum response-time bound arises also in the case of $m_1 = 1$.

▶ **Lemma 6.** *If $m_1 = 1$, then* $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$.

Proof. If $m_1 = 1$, then there is only one task assigned to cluster C_1. The single processor in C_1 is available to jobs of T_1 only when A_1 is inactive. Recall from Table 2 that the maximum critical section lengths of tasks $T_{m_1+1}, \ldots, T_n$ are halved if $m_1 = 1$. Analogously to Lemma 4, it can thus be shown that, in the worst case, the single processor in C_1 is available to T_1 for only $\frac{n}{2}$ time units out of each interval $[x \cdot n, x \cdot n + n)$, where $x \in \mathbb{N}$.

Consider an interval $[t_a, t_a + R \cdot n)$, where $t_a = x \cdot R \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous arrival sequence, task T_1 releases a J_1 at time t_a. In the worst case, J_1 requires $e_1 = \frac{R \cdot n}{2}$ time units to complete. Assuming maximum interference by A_1 (*i. e.*, if the processor is unavailable to J_1 for $\frac{n}{2}$ time units every n time units), J_1 will accumulate e_1 time units of processor service only by time $t_a + 2e_i = t_a + R \cdot n$. ◀

Since there are m_1 processors and m_1 pending jobs in cluster C_1, all pending jobs should be immediately scheduled under any work-conserving scheduling policy. However, since the priority-boosted agent occupies one of the processors, this is not the case, which implies that one job incurs s-oblivious pi-blocking (under any work-conserving JLFP policy).

▶ **Lemma 7.** *Under s-oblivious schedulability analysis,* $\max\{b_i \mid T_i \in \tau^{seq}(n, m_1, R)\} \geq \frac{R \cdot n}{2}$.

Proof. By construction, there are at most m_1 pending jobs in cluster C_1 at any time. Hence any delay of a pending job constitutes s-oblivious pi-blocking (recall Definition 1): $b_i = r_i - e_i$ for each $T_i \in \{T_1, \ldots, T_{m_1}\}$, regardless of the employed JLFP scheduling policy. Since $e_i = \frac{R \cdot n}{2}$ for each $T_i \in \{T_1, \ldots, T_{m_1}\}$, we have $\max\{b_i \mid 1 \leq i \leq n\} \geq \max\{r_i \mid 1 \leq i \leq m_1\} - \frac{R \cdot n}{2}$. By Lemmas 5 and 6, $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$, and thus $\max\{b_i \mid 1 \leq i \leq n\} \geq R \cdot n - \frac{R \cdot n}{2} = \frac{R \cdot n}{2}$. ◀

Since the agent A_1 and tasks $T_1, \ldots, T_{m_1}$ share a cluster in $\tau^{seq}(n, m_1, R)$, and because the s-oblivious pi-blocking implies s-aware pi-blocking, we obtain the following lower bound on maximum pi-blocking under co-hosted task allocation.

▶ **Theorem 8.** *Under JLFP scheduling, using either s-aware or s-oblivious schedulability analysis, there exists a task set such that, under co-hosted task allocation,* $\max\{b_i\} = \Omega(\Phi \cdot n)$ *under any weakly work-conserving distributed multiprocessor real-time semaphore protocol that employs priority-boosted agents (i. e., under protocols matching Assumptions A1 and A2).*

Proof. By Lemma 7, there exists a task set $\tau^{seq}(n, m_1, R)$ such that, under s-oblivious schedulability analysis, any JLFP policy, and any distributed multiprocessor semaphore protocol satisfying Assumptions A1 and A2, $\max\{b_i \mid T_i \in \tau^{seq}(n, m_1, R)\} \geq \frac{R \cdot n}{2}$ for any $R \in \mathbb{N}$. Recall from Section 2.1 that $\Phi = \frac{\max\{r_i\}}{\min\{p_i\}}$, and hence $\Phi = \frac{R \cdot n}{n} = R$ in the case of $\tau^{seq}(n, m_1, R)$. Since R can be freely chosen, we have $\max\{b_i\} = \Omega(R \cdot n) = \Omega(\Phi \cdot n)$ under s-oblivious schedulability analysis.

Recall from Section 2.3 that s-oblivious pi-blocking implies s-aware pi-blocking (*i. e.*, Definition 2 holds if Definition 1 is satisfied). The established lower bound on s-oblivious pi-blocking therefore also applies to s-aware pi-blocking [16], and thus $\max\{b_i\} = \Omega(\Phi \cdot n)$ under either s-aware or s-oblivious schedulability analysis. ◀

Compared to a shared-memory system, where the shared-memory mutual exclusion problem can be solved with $\Theta(n)$ maximum s-aware pi-blocking in the general case [11, 16], Theorem 8 shows that maximum pi-blocking under distributed locking protocols is *asymptotically worse* by a factor of Φ. Maximum s-oblivious pi-blocking is also asymptotically worse—the equivalent shared-memory mutual exclusion problem can be solved with $\Theta(m)$ maximum s-oblivious pi-blocking [11, 16, 18] (recall that we assume $n \geq m$). Note that, Φ, the ratio of the maximum response time and the minimum period, can in general be arbitrarily large and is independent of either m or n. This suggests that, from a schedulability point of view, the mutual exclusion problem is fundamentally more difficult in a distributed environment.

The observed discrepancy, however, is entirely due to the effects of preemptions caused by priority-boosted agents. While it is not possible to avoid priority boosting entirely (otherwise excessive pi-blocking could result when agents are preempted by jobs with large execution costs), such troublesome preemptions can be easily ruled out by disallowing the co-hosting of agents and tasks in the same cluster. And in fact, when using such a disjoint task allocation approach, the asymptotic lower bounds on maximum pi-blocking under distributed locking protocols are identical to those previously established for shared-memory semaphore protocols. The matching lower bounds can be trivially established with the setup previously used in [16]; we omit the details here and summarize the correspondence with the following theorem.

▶ **Theorem 9.** *There exist task sets such that, under JLFP scheduling, disjoint task allocation, and any distributed real-time semaphore protocol satisfying Assumptions A1 and A2,* $\max\{b_i\} = \Omega(n)$ *under s-aware schedulability analysis and* $\max\{b_i\} = \Omega(m)$ *under s-oblivious schedulability analysis.*

Having established lower bounds on s-oblivious and s-aware pi-blocking under both co-hosted and disjoint task allocation, we next explore the question of asymptotic optimality—how to construct protocols that ensure upper bounds on maximum pi-blocking that are within a constant factor of the established lower bounds? We begin with the co-hosted case in Section 4, and consider the disjoint case in Section 5 thereafter.

As a final remark, we note that the task set $\tau^{seq}(n, m_1, R)$ as given in Table 2 contains tasks with relative deadlines larger than periods (*i. e.*, $d_i > p_i$ for $i > m_1$). This is purely a matter of convenience; asymptotically equivalent bounds can be derived with implicit-deadline tasks.

4 Asymptotic Optimality under Co-Hosted Task Allocation

Theorem 8 shows that there exist pathological scenarios in which the choice of real-time locking protocol is seemingly irrelevant: regardless of the specifics of the employed locking protocol, worst-case pi-blocking is asymptotically worse than in a comparable shared-memory system simply because resources are inaccessible from some processors. Curiously, from an asymptotic point of view, protocol-specific rules are indeed immaterial: *any* distributed real-time locking protocol that does not starve requests is *asymptotically optimal* in the case of co-hosted task allocation.

▶ **Theorem 10.** *Under any JLFP scheduler, any weakly-work-conserving, distributed real-time semaphore protocol that employs priority boosting (i. e., any protocol matching Assumptions A1 and A2) ensures* $O(\Phi \cdot n)$ *maximum pi-blocking, regardless of whether s-aware or s-oblivious schedulability analysis is employed.*

Proof. Recall from Definition 2 that a pending job J_b incurs *s-aware* pi-blocking if J_b is not scheduled and not all processors in its assigned cluster are occupied by higher-priority jobs. This happens either when **(i)** J_b is suspended while waiting for a resource request to be completed, or when **(ii)** J_b is preempted by a priority-boosted agent that executes on behalf of another job.

Concerning (i), the completion of J_b's own requests can only be delayed by other requests (and not by the execution of other jobs) since agents are priority-boosted, and since the employed distributed locking protocol is weakly work-conserving (*i. e.*, whenever one of J_b's requests is delayed, at least one other request is being processed by some agent).

Concerning (ii), agents only become active when invoked by other jobs.

Hence the total duration of all requests (issued by jobs of any task) that are executed while J_b is pending provides a trivial upper bound on the maximum cumulative agent activity, and hence also on the maximum total duration of pi-blocking incurred by J_b.

To this end, consider for any task T_x the maximum number of jobs of T_x that execute while J_b is pending, which is bounded by $\left\lceil \frac{r_x + r_b}{p_x} \right\rceil$.[6] Since there are n tasks in total, this implies that at most $\sum_{x=1}^{n} \left\lceil \frac{r_x + r_b}{p_x} \right\rceil = \sum_{x=1}^{n} \left\lceil \frac{r_x}{p_x} + \frac{r_b}{p_x} \right\rceil \leq \sum_{x=1}^{n} \left\lceil \frac{\max_i\{r_i\}}{\min_i\{p_i\}} + \frac{\max_i\{r_i\}}{\min_i\{p_i\}} \right\rceil = n \cdot \lceil 2\Phi \rceil = O(\Phi \cdot n)$ jobs (in total across all tasks) are executed while J_b is pending. Since $\sum_{\ell_q} N_{i,q} \cdot L_{i,q} = O(1)$ for each T_i (*i. e.*, since each job issues at most a constant number of requests), it follows that $\max_i\{b_i\} = O(n \cdot \Phi)$, regardless of any protocol-specific rules.

Recall from Section 2.3 that s-oblivious pi-blocking implies s-aware pi-blocking (*i. e.*, if Definition 1 is satisfied, then Definition 2 holds, too). Hence, an upper bound on s-aware pi-blocking

[6] See *e. g.* [11, Ch. 4] for a formal proof of this well-known bound.

implicitly also upper-bounds s-oblivious pi-blocking, and thus $\max_i\{b_i\} = O(n \cdot \Phi)$ under either s-oblivious or s-aware schedulability analysis. ◀

As a corollary, Theorem 10 implies that the DPCP, which orders requests according to task priority, is asymptotically optimal in the co-hosted setting. However, it also shows that requests may be processed in arbitrary order (*e. g.*, in FIFO order, or even in random order) without losing asymptotic optimality (as long as at least one request at a time is satisfied and agents are priority-boosted), which is surprising as the queue order is crucial in the shared-memory case [16].

As already noted in the previous section, by prohibiting the co-hosting of resources and tasks—that is, somewhat counter-intuitively, by making the system *less* similar to a shared-memory system (in which tasks and critical sections are necessarily co-hosted, *i. e.*, executed on the same set of processors)—it is indeed possible to ensure maximum s-aware pi-blocking that is asymptotically no worse than under a shared-memory locking protocol. We establish this fact next by introducing two new protocols that realize $O(n)$ and $O(m)$ maximum pi-blocking under s-aware and s-oblivious schedulability analysis, respectively, in the case of disjoint task allocation. As one might expect, the choice of queue order is significant in this case.

5 Asymptotic Optimality under Disjoint Task Allocation

Prior work [11, 15, 16, 18] has established shared-memory protocols that yield upper bounds on maximum s-aware and s-oblivious pi-blocking of $O(n)$ and $O(m)$, respectively. These protocols, namely the *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) for s-aware analysis [11, 15] and the family of $O(m)$ *Locking Protocols* (the OMLP family) for s-oblivious analysis [11, 16, 18], rely on specific queue structures with strong progress guarantees to obtain the desired bounds. In the following, we show how the key ideas underlying the FMLP$^+$ and the OMLP family can be adopted to the problem of designing asymptotically optimal locking protocols for the distributed case studied in this paper. We begin with the slightly simpler s-aware case.

5.1 Asymptotically Optimal Maximum S-Aware PI-Blocking

Inspired by the FMLP$^+$ [11], the *Distributed FIFO Locking Protocol* (DFLP) relies on simple FIFO queues to avoid starvation. Notably, the DFLP ensures $O(n)$ maximum s-aware pi-blocking under disjoint task allocation and transparently supports arbitrary, non-uniform cluster sizes (*i. e.*, unlike the DPCP, the DFLP supports distributed systems consisting of multiprocessor nodes with $m_j > 1$ for some C_j and allows $m_j \neq m_h$ for any $j \neq h$). We first describe the structure and rules of the DFLP, and then establish its asymptotic optimality.

5.1.1 Rules

Under the DFLP, conflicting requests for each serially-reusable resource ℓ_q are ordered with a per-resource FIFO queue FQ_q. Requests for ℓ_q are served by an agent A_q assigned to ℓ_q's cluster $C(\ell_q)$. Resource requests are processed according to the following rules.

1. When J_i issues a request $\mathcal{R}$ for resource ℓ_q, J_i suspends and $\mathcal{R}$ is appended to FQ_q. J_i's request is processed by agent A_q when $\mathcal{R}$ becomes the head of FQ_q.
2. When $\mathcal{R}$ is complete, it is removed from FQ_q and J_i is resumed.
3. Active agents are scheduled preemptively in the order in which their current requests were issued (*i. e.*, an agent processing an earlier-issued request has higher priority than one serving a later-issued request). Any ties can be broken arbitrarily (*e. g.*, in favor of agents serving requests of lower-indexed tasks).
4. Agents have statically higher priority than jobs (*i. e.*, agents are subject to priority-boosting).

We next show that these simple rules yield asymptotic optimality.

5.1.2 Blocking Complexity

The co-hosted case is trivial since the DFLP uses priority boosting (Rule 4) and because it is weakly work-conserving (requests are satisfied as soon as the requested resource is available—see Rule 1); Theorem 10 hence applies.

To show asymptotic optimality in the disjoint case, we first establish a per-request bound on the number of interfering requests that derives from FIFO-ordering both requests and agents.

▶ **Lemma 11.** *Let $\mathcal{R}$ denote a request issued by a job J_i for a resource ℓ_q and let T_x denote a task other than T_i (i. e., $i \neq x$). Under the DFLP, jobs of T_x delay the completion of $\mathcal{R}$ with at most one request.*

Proof. J_i's request $\mathcal{R}$ cannot be delayed by later-issued requests since FQ_q is FIFO-ordered and because agents are scheduled in FIFO order according to the issue time of the currently-served request. Since $\mathcal{R}$ is not delayed by later-issued requests (and clearly not by earlier-completed requests), all requests that delay the completion of $\mathcal{R}$ are incomplete at the time that $\mathcal{R}$ is issued. Since tasks and jobs are sequential, and since jobs request at most one resource at a time, there exists at most one incomplete request per task at any time. ◀

An $O(n)$ bound on maximum s-aware pi-blocking follows immediately since each of the other $n-1$ tasks delays J_i at most once each time J_i requests a resource, and since agents cannot preempt jobs in the disjoint setting.

▶ **Theorem 12.** *Under the DFLP with disjoint task allocation,* $\max\{b_i\} = O(n)$.

Proof. Let J_i denote an arbitrary job. Since, by assumption, no agents execute on J_i's cluster, J_i incurs pi-blocking only when suspended while waiting for a request to complete. By Lemma 11, each other task delays each of J_i's $\sum_q N_{i,q}$ requests for at most the duration of one request, that is, per request, J_i incurs no more than $n \cdot L^{max}$ s-aware pi-blocking. Since J_i issues at most $\sum_q N_{i,q}$ requests, and since by assumption $\sum_q N_{i,q} = O(1)$ and $L^{max} = O(1)$, we have $b_i \leq n \cdot L^{max} \cdot \sum_q N_{i,q} = O(n)$. ◀

The DFLP is thus asymptotically optimal with regard to maximum s-aware pi-blocking, under both co-hosted (Theorem 10) and disjoint task allocation (Theorem 12). In contrast, the DPCP does not generally guarantee $O(n)$ s-aware pi-blocking in the disjoint case since it orders conflicting requests by task priority and is thus prone to starvation issues (this can be shown similarly to the lower bound on priority queues established in [11, 16]).

This concludes the case of s-aware analysis. Next, we consider the s-oblivious case.

5.2 Asymptotically Optimal Maximum S-Oblivious PI-Blocking

In this section, we define and analyze the *Distributed $O(m)$ Locking Protocol* (D-OMLP), which augments the OMLP family with support for distributed systems.

In order to prove optimality under s-oblivious analysis, a protocol must ensure an upper bound of $O(m)$ s-oblivious pi-blocking. Since there are $n \geq m$ tasks in total, if each task is allowed to submit a request concurrently, excessive contention could arise at each agent: if an agent is faced with n concurrent requests, it is not possible to ensure $O(m)$ maximum s-oblivious pi-blocking regardless of the order in which requests are processed. Thus, it is necessary to limit contention early within each application cluster (where job priorities can be taken into account) to only allow a subset of high-priority jobs to invoke agents at the same time. In the interest of practicality, such "contention limiting" should not require coordination across clusters, but rather must be

based on only local information. As we show next, this can be accomplished by reusing (aspects of) two protocols of the OMLP family.

The first technique is to introduce *contention tokens*, which are virtual, cluster-local resources that a job must acquire prior to invoking an agent. This technique was previously used in the shared-memory OMLP variant for partitioned JLFP scheduling [16]. By limiting the number of contention tokens to m in total (*i. e.*, by assigning exactly m_j such tokens to each cluster C_j), each agent is faced with at most m concurrent requests.

This in turn creates the problem of managing access to contention tokens. However, since contention tokens are a cluster-local resource, this reduces to a shared-memory problem and prior results on optimal shared-memory real-time synchronization can be reused. In fact, as there may be multiple contention tokens in each cluster (if $m_j > 1$), of which a job may use any one, this reduces to a k-exclusion problem (where k denotes the number of tokens per cluster in this case). Several asymptotically optimal solutions for the k-exclusion problem under s-oblivious analysis have been developed [18, 25, 50], including a variant of the OMLP [18]; the contention tokens can thus be readily managed within each cluster using any of the available k-exclusion protocols [18, 25, 50]. These considerations lead to the following protocol definition.

5.2.1 Rules

Under the D-OMLP, there are m_j contention tokens in each cluster C_j, for a total of $m = \sum_{j=1}^{K} m_j$ such tokens. As in the DFLP, there is one agent A_q and a FIFO queue FQ_q for each resource ℓ_q.

Jobs may access shared resources according to the following rules. In the following, let J_i denote a job that must access resource ℓ_q.

1. Before J_i may invoke agent A_q, it must first acquire a contention token local to cluster $C(T_i)$ according to the rules of an asymptotically optimal k-exclusion protocol.
2. Once J_i holds a contention token, it immediately issues its request $\mathcal{R}$ by invoking A_q and suspends. $\mathcal{R}$ is appended to FQ_q and processed by A_q when it becomes the head of FQ_q.
3. When $\mathcal{R}$ is complete, it is removed from FQ_q. J_i is resumed and immediately relinquishes its contention token.
4. Active, ready agents are scheduled preemptively in order of non-decreasing request enqueueing times (*i. e.*, while processing $\mathcal{R}$, agent A_q's priority is the point in time at which $\mathcal{R}$ was enqueued in FQ_q). Any ties in the times that requests were enqueued can be broken arbitrarily.
5. Agents have a statically higher priority than jobs (*i. e.*, agents are subject to priority-boosting).

As shown next, the contention tokens in combination with FIFO-ordering requests and agents yield an asymptotically optimal maximum s-oblivious pi-blocking bound.

5.2.2 Blocking Complexity

As with the DFLP, the co-hosted case is trivial since Theorem 10 applies to the D-OMLP.

In the disjoint case, we first establish a bound on the *maximum token-hold time*, since jobs can incur s-oblivious pi-blocking both due to Rule 1 (*i. e.*, when no contention tokens are available) and due to Rules 2 and 4 (*i. e.*, when $\mathcal{R}$ is preceded by other requests in FQ_q or if A_q is preempted while processing $\mathcal{R}$).

▶ **Lemma 13.** *A job J_i holds a contention token for at most $m \cdot L^{max}$ time units per request.*

Proof. By Rules 1 and 3, a job J_i holds a contention token while it waits for its request $\mathcal{R}$ to be completed. Analogously to Lemma 11, since FQ_q is FIFO-ordered and since agents are scheduled in FIFO order w.r.t. the time that requests are enqueued (Rule 4), the completion of $\mathcal{R}$ can only

be delayed due to the execution of requests that were incomplete at the time that $\mathcal{R}$ was enqueued in FQ_q. By Rule 1, only jobs holding a contention token may issue requests to agents. Since there are only $m = \sum_{j=1}^{K} m_j$ contention tokens in total, there exist at most $m - 1$ incomplete requests at the time that $\mathcal{R}$ is enqueued in FQ_q. Hence, $\mathcal{R}$ is completed and J_i relinquishes its contention token after at most $m \cdot L^{max}$ time units. ◀

By leveraging a k-exclusion protocols that is asymptotically optimal under s-oblivious analysis (Rule 1), Lemma 13 immediately yields an $O(m)$ bound on maximum s-oblivious pi-blocking.

▶ **Theorem 14.** *Under the D-OMLP with disjoint task allocation,* $\max\{b_i\} = O(m)$.

Proof. Let H denote the maximum token-hold time. By Lemma 13, the maximum token-hold time is $H = m \cdot L^{max} = O(m)$. Further, H represents the "maximum critical section length" w.r.t. the contention token k-exclusion problem. By Rule 1, an asymptotically optimal k-exclusion protocol is employed to manage access to contention tokens within each cluster. Applied to a cluster with m_j processors, the k-exclusion problem can be solved such that jobs incur s-oblivious pi-blocking for the duration of at most $O\left(\frac{m_j}{k}\right)$ critical section lengths per request [18, 25, 50]. Under the D-OMLP, there are $k = m_j$ contention tokens in each cluster C_j. Hence, in the disjoint setting, a task assigned to cluster C_j incurs $O\left(\frac{m_j}{m_j} \cdot H\right) = O(H) = O(m)$ s-oblivious pi-blocking. ◀

The D-OMLP is thus asymptotically optimal under s-oblivious schedulability analysis, and hence a natural extension of the OMLP family to the distributed real-time locking problem.

6 Conclusion

In this paper, we studied blocking optimality in distributed real-time locking protocols. We identified two different task and resource allocation strategies, namely co-hosted and disjoint task allocation, that give rise to different answers to this question. In the co-hosted case, under both s-aware and s-oblivious analysis, $\Omega(\Phi \cdot n)$ maximum pi-blocking is unavoidable in the general case, whereas in the disjoint case, $\Omega(n)$ maximum s-aware and $\Omega(m)$ maximum s-oblivious pi-blocking are the fundamental lower bounds. The significance of these bounds is that the lower bound on maximum pi-blocking in the case of co-hosted task allocation is asymptotically worse than in an equivalent shared-memory scenario. In contrast, disjoint task allocation yields the same lower bounds already known from the analysis of shared-memory synchronization.

We further showed that the established lower bounds are asymptotically tight. In the co-hosted case, any distributed locking protocol satisfying Assumptions A1 and A2 is asymptotically optimal (Theorem 10). To prove asymptotic tightness in the disjoint case, we introduced two new distributed real-time semaphore protocols. Specifically, the DFLP is asymptotically optimal under s-aware analysis, and the D-OMLP is asymptotically optimal under s-oblivious analysis, both w.r.t. the maximum pi-blocking metric.

Pi-blocking is generally undesirable, and hence protocols that guarantee lower asymptotic pi-blocking bounds are intuitively preferable. Our results are the first formal characterization of the fundamental limits on pi-blocking in a distributed setting and serve to structure the design space of distributed real-time locking protocols. However, one should also note that a lower asymptotic pi-blocking bound does not necessarily imply better overall schedulability.

For one, while disjoint task allocation permits lower bounds on pi-blocking, it also requires dedicating some cluster(s) to agents, which, depending on constant factors such as the level of contention and critical section lengths, may decrease the overall utilization of the system. Whether disjoint task allocation is beneficial is thus a workload-specific question that must be answered individually for each task set.

Further, asymptotic optimality does *not* imply that an asymptotically optimal protocol is always preferable to a non-optimal one. Rather, blocking bounds of asymptotically similar locking protocols can still differ significantly in absolute terms. Whether a particular locking protocol is suitable for a particular task set depends on both the task set's specific requirements and a protocol's constant factors, which asymptotic analysis does not reflect. In particular, this is the case under co-hosted task allocation, where *all* distributed locking protocols (in the considered class of protocols) differ *only* in terms of constant factors. Fine-grained (*i. e.*, non-asymptotic) bounds on maximum pi-blocking suitable for schedulability analysis are thus required for practical use and to enable a detailed comparison. Such bounds should not only reflect a detailed analysis of protocol rules, but also exploit task-set-specific properties such as per-task bounds on request lengths and request frequencies. For the DFLP and the DPCP, we have recently developed such bounds [14]; the same techniques could also be applied to analyze the D-OMLP.

As noted in Section 2.2, we have made the assumption that jobs can invoke agents with "negligible" overheads (*i. e.*, with overheads that can be accounted for using known overhead accounting techniques [11]). This is a reasonable assumption in platforms with point-to-point links, in systems with networks employing TDMA or time-triggered [34] arbitration policies, or if distributed semaphore protocols are implemented on top of a (large) shared-memory platform (*e. g.*, see [14] for such a case). However, the assumption may be more problematic in systems that require explicit message routing across a shared, dynamically arbitrated network. Assuming there exists an upper bound $\Delta_{i,q}$ on the message delay between a task T_i and each agent A_q, such delays can be incorporated by simply increasing T_i's self-suspension time by $2\Delta_{i,q}$ for each agent invocation (under the D-OMLP, the maximum token-hold time is increased by $2\Delta_{i,q}$ as well). If $\Delta_{i,q}$ can be considered constant (*i. e.*, if $\Delta_{i,q} = O(1)$ from an asymptotic analysis point of view), then the asymptotic upper and lower bounds established in this paper remain unaffected. If, however, $\Delta_{i,q}$ depends on m or n, or on other non-constant factors, then additional analysis is required, which may be an interesting direction for future work.

In another opportunity for future work, it will also be interesting to explore how to accommodate *nested requests*, that is, how to allow complex requests that require agents to invoke other agents. Ward and Anderson have recently shown that arbitrarily deep nesting can be supported in shared-memory locking protocols without loss of asymptotic optimality [49]; however, it remains to be seen how their techniques can be extended to distributed real-time semaphore protocols.

References

1. Neil C. Audsley, Alan Burns, Mike F. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
2. Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991. doi:10.1007/BF00365393.
3. Theodore P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *24th IEEE Real-Time Systems Symposium, RTSS'03*, pages 120–129. IEEE Computer Society, December 2003. doi:10.1109/REAL.2003.1253260.
4. Sanjoy K. Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE Real-Time Systems Symposium, RTSS'07*, pages 119–128. IEEE Computer Society, December 2007. doi:10.1109/RTSS.2007.48.
5. Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010. doi:10.1007/s11241-010-9096-3.
6. Andrew Baumann, Paul Barham, Pierre Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *22nd ACM Symposium on Operating Systems Principles 2009, SOSP'09*, pages 29–44. ACM, October 2009. doi:10.1145/1629575.1629579.
7. Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE Real-Time Systems Symposium, RTSS'07*, pages 149–160. IEEE Computer Society, December 2007. doi:10.1109/RTSS.2007.41.

8 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems, ECRTS'05*, pages 209–218. IEEE Computer Society, July 2005. doi:10.1109/ECRTS.2005.18.

9 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, 2009. doi:10.1109/TPDS.2008.129.

10 Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'07*, pages 47–56. IEEE Computer Society, August 2007. doi:10.1109/RTCSA.2007.8.

11 Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. URL: http://www.cs.unc.edu/~bbb/diss/.

12 Björn B. Brandenburg. A note on blocking optimality in distributed real-time locking protocols. unpublished manuscript, 2012. URL: https://www.mpi-sws.org/~bbb/papers/index.html.

13 Björn B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *25th Euromicro Conference on Real-Time Systems, ECRTS'13*, pages 292–302. IEEE, July 2013. doi:10.1109/ECRTS.2013.38.

14 Björn B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS'13*, pages 141–152. IEEE Computer Society, April 2013. doi:10.1109/RTAS.2013.6531087.

15 Björn B. Brandenburg. The FMLP$^+$: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *26th Euromicro Conference on Real-Time Systems, ECRTS'14*, pages 61–71. IEEE, July 2014.

16 Björn B. Brandenburg and James H. Anderson. Optimality results for multiprocessor real-time locking. In *31st IEEE Real-Time Systems Symposium, RTSS'10*, pages 49–60. IEEE Computer Society, November 2010. doi:10.1109/RTSS.2010.17.

17 Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010. doi:10.1007/s11241-010-9097-2.

18 Björn B. Brandenburg and James H. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, online first:1–66, July 2012. doi:10.1007/s10617-012-9090-1.

19 Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS'08*, pages 342–353. IEEE Computer Society, April 2008. doi:10.1109/RTAS.2008.27.

20 Alan Burns and Andy J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – MrsP. In *25th Euromicro Conference on Real-Time Systems, ECRTS'13*, pages 282–291. IEEE, July 2013. doi:10.1109/ECRTS.2013.37.

21 John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H. Anderson, and Sanjoy K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC, 2004.

22 Yang Chang, Robert I. Davis, and Andy J. Wellings. Reducing Queue Lock Pessimism in Multiprocessor Schedulability Analysis. In *18th International Conference on Real-Time and Network Systems*, pages 99–108, Toulouse, France, November 2010. URL: http://hal.archives-ouvertes.fr/hal-00546915.

23 UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *18th Euromicro Conference on Real-Time Systems, ECRTS'06*, pages 75–84. IEEE Computer Society, July 2006. doi:10.1109/ECRTS.2006.10.

24 Arvind Easwaran and Björn Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In Theodore P. Baker, editor, *30th IEEE Real-Time Systems Symposium, RTSS'09*, pages 377–386. IEEE Computer Society, December 2009. doi:10.1109/RTSS.2009.37.

25 Glenn A. Elliott and James H. Anderson. An optimal k-exclusion real-time locking protcol motivated by multi-gpu systems. In Sébastien Faucou, Alan Burns, and Laurent George, editors, *19th International Conference on Real-Time and Network Systems, RTNS'11*, pages 15–24, September 2011. URL: http://rtns2011.irccyn.ec-nantes.fr/files/rtns2011.pdf.

26 Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The multiprocessor bandwidth inheritance protocol. In *22nd Euromicro Conference on Real-Time Systems, ECRTS'10*, pages 90–99. IEEE Computer Society, July 2010. doi:10.1109/ECRTS.2010.19.

27 Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6):789–825, 2012. doi:10.1007/s11241-012-9162-0.

28 Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *22nd IEEE Real-Time Systems Symposium, RTSS'01*, pages 73–83. IEEE Computer Society, December 2001. doi:10.1109/REAL.2001.990598.

29 Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03 2003)*, page 189. IEEE Com-

puter Society, May 2003. doi:10.1109/RTTAS.2003.1203051.

30 Joël Goossens, Shelby Funk, and Sanjoy K. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003. doi:10.1023/A:1025120124771.

31 Pi-Cheng Hsiu, Der-Nien Lee, and Tei-Wei Kuo. Task synchronization and allocation for many-core real-time systems. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *11th International Conference on Embedded Software, EMSOFT'11*, pages 79–88. ACM, October 2011. doi:10.1145/2038642.2038656.

32 Craig T. Jin. Queuing spin lock algorithms to support timing predictability. In *Real-Time Systems Symposium*, pages 148–157, December 1993. doi:10.1109/REAL.1993.393505.

33 Theodore Johnson and Krishna Harathi. A prioritized multiprocessor spin lock. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):926–933, 1997. doi:10.1109/71.615438.

34 Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC'05*, pages 22–33. IEEE Computer Society, May 2005. doi:10.1109/ISORC.2005.56.

35 Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In Theodore P. Baker, editor, *30th IEEE Real-Time Systems Symposium, RTSS'09*, pages 469–478. IEEE Computer Society, December 2009. doi:10.1109/RTSS.2009.51.

36 Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980. doi:10.1145/358818.358824.

37 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

38 Victor B. Lortz and Kang G. Shin. Semaphore queue priority assignment for real-time multiprocessor synchronization. *IEEE Transactions on Software Engineering*, 21(10):834–844, 1995. doi:10.1109/32.469457.

39 Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference*, pages 65–76. USENIX Association, June 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

40 Georgiana Macariu and Vladimir Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In Karl-Erik Årzén, editor, *23rd Euromicro Conference on Real-Time Systems, ECRTS'11*, pages 262–271. IEEE Computer Society, July 2011. doi:10.1109/ECRTS.2011.32.

41 Evangelos P. Markatos and Thomas J. LeBlanc. Multiprocessor synchronization primitives with priorities. In *8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, 1991.

42 Farhang Nemati, Moris Behnam, and Thomas Nolte. Independently-developed real-time systems on multi-cores with shared resources. In Karl-Erik Årzén, editor, *23rd Euromicro Conference on Real-Time Systems, ECRTS'11*, pages 251–261. IEEE Computer Society, July 2011. doi:10.1109/ECRTS.2011.31.

43 Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems, ICDCS'90*, pages 116–123. IEEE Computer Society, May 1990. doi:10.1109/ICDCS.1990.89257.

44 Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

45 Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *9th IEEE Real-Time Systems Symposium, RTSS'88*, pages 259–269. IEEE Computer Society, December 1988. doi:10.1109/REAL.1988.51121.

46 Frédéric Ridouard, Pascal Richard, and Francis Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *25th IEEE Real-Time Systems Symposiumm, RTSS'04*, pages 47–56. IEEE Computer Society, December 2004. doi:10.1109/REAL.2004.35.

47 Simon Schliecker, Mircea Negrean, and Rolf Ernst. Response time analysis in multicore ecus with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, 2009. doi:10.1109/TII.2009.2032068.

48 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

49 Bryan C. Ward and James H. Anderson. Supporting nested locking in multiprocessor real-time systems. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS'12*, pages 223–232. IEEE Computer Society, July 2012. doi:10.1109/ECRTS.2012.17.

50 Bryan C. Ward, Glenn A. Elliott, and James H. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'12*, pages 280–289. IEEE Computer Society, August 2012. doi:10.1109/RTCSA.2012.26.

51 Richard West, Ye Li, and Eric S. Missimer. Time management in the Quest-V RTOS. In *8th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2012. URL: http://www.cs.bu.edu/fac/richwest/papers/questv-multicore.pdf.

52 Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *IEEE 34th Real-Time Systems Symposium, RTSS'13*, pages 45–56. IEEE, December 2013. doi:10.1109/RTSS.2013.13.

Computation Offloading for Frame-Based Real-Time Tasks under Given Server Response Time Guarantees

Anas Toma[1] and Jian-Jia Chen[2]

1 **Department of Informatics, Karlsruhe Institute of Technology, Germany**
anas.toma@student.kit.edu

2 **Department of Informatics, TU Dortmund University, Germany**
jian-jia.chen@cs.uni-dortmund.de

Abstract

Computation offloading has been adopted to improve the performance of embedded systems by offloading the computation of some tasks, especially computation-intensive tasks, to servers or clouds. This paper explores computation offloading for real-time tasks in embedded systems, provided given response time guarantees from the servers, to decide which tasks should be offloaded to get the results in time. We consider frame-based real-time tasks with the same period and relative deadline. When the execution order of the tasks is given, the problem can be solved in linear time. However, when the execution order is not specified, we prove that the problem is $\mathcal{NP}$-complete. We develop a pseudo-polynomial-time algorithm for deriving feasible schedules, if they exist. An approximation scheme is also developed to trade the error made from the algorithm and the complexity. Our algorithms are extended to minimize the period/relative deadline of the tasks for performance maximization. The algorithms are evaluated with a case study for a surveillance system and synthesized benchmarks.

2012 ACM Subject Classification Software and its engineering, Scheduling, Computer systems organization, Real-time systems

Keywords and phrases Computation offloading, task scheduling, real-time systems

Digital Object Identifier 10.4230/LITES-v001-i002-a002

Received 2013-02-28 **Accepted** 2014-08-22 **Published** 2014-11-14

1 Introduction

In the recent years, a significant increase in the development of mobile devices has been achieved. They have become devices that provide various computation-intensive services and applications, including video, audio, images, etc. Also, mobile robots have become more and more popular and important in the recent years. For instance, the sales of service robots for personal and household purposes have been increased significantly in the past years, i. e., 35 % increase in 2010 [7]. Furthermore, the number of service robots sold per year is also expected to increase in the next few years [7].

However, due to the resource constraints on both mobile devices and robots, their computation capabilities are still quite limited. For some applications on these devices, if the peak performance requirement happens rarely or is not always required, designing the embedded system for the extreme case to achieve the peak performance is usually too pessimistic, as most resources will be wasted. Moreover, when increasing the performance of an embedded system, we will also usually increase the power consumption, the weight, and also the cost of the devices.

Improving the embedded systems just for extreme cases, for executing some computation-intensive applications, may waste the device resources in normal cases if the extreme case is needed rarely. Therefore, *computation offloading* can be used to move a task from a resource-constrained device (here, we call it a *client*) to one or more devices (here, we call them *servers*). Figure 1

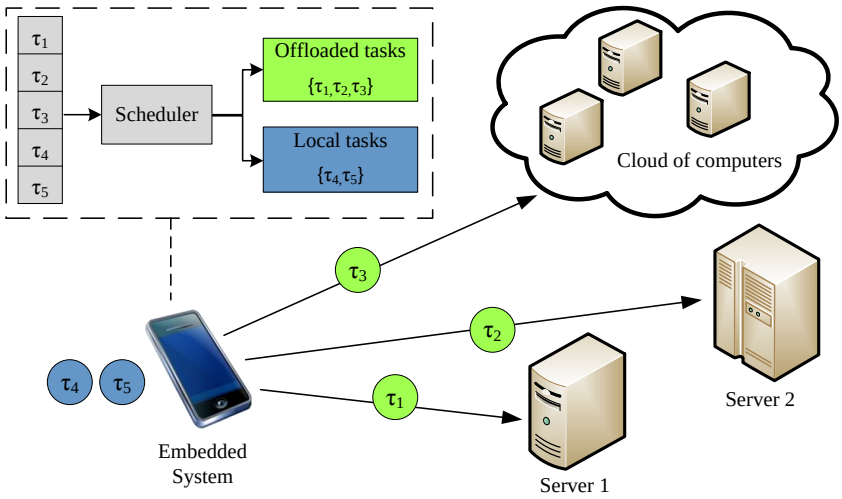


Figure 1 Offloading Mechanism.

illustrates the computation offloading mechanism. The task can be a part of an active program (e. g., function, class, etc.) or a complete one. The servers can either provide faster execution in general (e. g., powerful desktop, an array of high-performance blade servers, cloud of computers, etc.) or accelerate the execution for some specific tasks (e. g., digital signal processing (DSP) units for signal decoding/encoding, General-purpose computing on graphics processing units (GPGPU) for accelerating, etc.). Furthermore, the server may be slower than the client. For such a case, the offloading may also be beneficial. Because the computation is done remotely, the energy consumption of the client can be reduced, or another task can be executed on the client while awaiting the results from the servers.

For example, some computation-intensive real-time tasks may be required to run on the *Electronic Control Units* (**ECU**s), that are distributed in the the automobiles, for specific time. However, this resource-constrained ECUs may not be able to finish the tasks execution in time. Improving the ECUs just for the extreme cases, if they happen rarely, to execute computation-intensive tasks may waste the resources in the normal cases and increases their cost. Therefore, offloading the computation-intensive tasks to a server (i. e., an additional processing unit inside the automobile with timing predictable communication), that serves all the ECUs in the extreme cases, is a cheaper and more flexible solution.

The idea of computation offloading has been studied previously [16, 9, 17, 10, 6, 3, 12, 8]. The existing approaches decide whether to execute a task locally or offload it without changing the execution order for the independent tasks. So, the client remains idle during the remote execution of an offloaded task until the result of this task returns from the server. Also, they consider, implicitly, a dedicated server for each client to run the offloaded task immediately. Furthermore, most of the existing computation offloading approaches either do not consider the timing satisfaction requirement for real-time properties, e. g., in [9, 16, 10, 6, 17], or use pessimistic offloading mechanism for deciding whether a task can be offloaded [12]. Timing requirements are important for real-time embedded systems, in which the results may become useless or even harmful to the client if the deadlines are missed.

Our Contributions. In this paper, computation offloading is exploited for real-time systems to meet the timing constraints. We consider frame-based real-time tasks with the same period and relative deadline under given response time guarantees from the servers. Our model is more

applicable for real-time embedded systems than the existing related work [16, 9, 17, 10, 6, 3, 12], in which (1) the client can execute another task locally while some offloaded tasks are executed on the servers, and (2) the server is not dedicated to a client to provide the service immediately, but provide a certain response timing assurance for the offloaded tasks. Our contributions are as follows:

- We prove that the offloading problem is $\mathcal{NP}$-complete even for frame-based real-time tasks with the same period and relative deadline without a specified execution order.
- We develop algorithms for deciding which tasks to be offloaded and how the tasks are executed to meet the timing constraints, for frame-based real-time tasks. We consider two cases, depending on whether the execution order of the tasks on the client is given or not. In case the order is given, the problem can be solved efficiently. Otherwise, we develop a pseudo-polynomial-time algorithm to derive a feasible schedule, if and only if it exists.
- We also provide an approximation scheme to trade the error made from the algorithm and the time/space complexity.
- Our algorithms can also be extended to maximize the sampling rate of the frame-based tasks by minimizing the period/relative deadline of the tasks.
- We evaluate for our proposed algorithms using a case study of a real-world application and randomly synthesized benchmarks. In our case study, a surveillance system is used to capture images periodically and execute four tasks within a deadline (i. e. sampling period).

The remainder of this paper is organized as follows: Section 2 summarizes the related work on computation offloading. Section 3 provides system model. Section 4 presents an efficient algorithm when the execution order is given. The hardness of the studied problem is shown in Section 5. Section 6 presents our approaches when the execution order is not given. Experimental results are presented in Section 7, and Section 8 concludes the paper.

2 Related Work

Computation offloading has been adopted in the literature to satisfy real-time requirements [12], improve performance [16], save energy [9, 17, 10, 6], and improve the quality of service [3].

For reducing the execution time and also the response time, Nimmagadda et al. [12] propose an offloading framework for mobile robots to satisfy the real-time constraints. Also, Wolski et al. [16] formulate computation offloading as a statistical decision problem by considering both the client and the servers are in computational grids. Offloading decisions in both of the above approaches are based on the comparison between two values: (1) the local execution time, and (2) the summation of the expected remote execution time in the server(s) and data transfer time. If the second value is less than the first one for a specific task, then this task is offloaded to the server(s) [12, 16].

Hong et al. [6] present an offloading strategy with three offloading options to reduce the energy consumption. Their strategy is dedicated for content-based image retrieval applications in mobile systems. For handheld devices, Li et al. [9] develop a scheme to run a program (task) by characterizing its corresponding client subtasks and server subtasks for executing on the client and servers, respectively. They build a cost graph for each program and use a branch-and-bound algorithm to minimize the energy consumption of the client. Moreover, Li et al. [10] also develop a computation offloading scheme by applying the standard maximum-flow/minimum-cut algorithm for deciding the server and client subtasks. For reducing the energy consumption, Xian et al. [17] apply timeout mechanism so that a task will be offloaded to a server if it cannot be finished before the timeout (timestamp) set for it. A middleware for mobile Android platforms is developed by Kovachev et al. [8] to offload the computation-intensive tasks from the mobile device to a remote

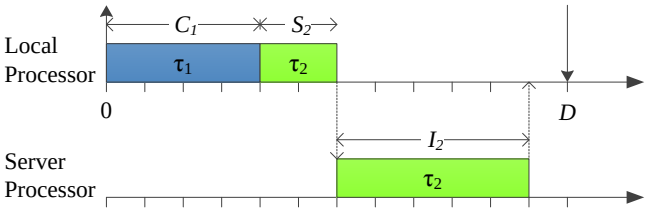


Figure 2 Timing parameters for two tasks.

cloud. The Offloading decision is represented as an optimization problem and solved using Integer Linear Programming (ILP).

3 System Model

We consider a system of one client and one or more servers for computation enhancement. Servers may provide higher computation capability than the client. On the client side, a set of frame-based real-time tasks arrive periodically and require execution within a common relative deadline. The tasks can be offloaded to the servers, but the results should be returned in time, i. e., no later than the deadline. The tasks are independent in execution without precedence constraints. The client has to schedule task executions to satisfy the real-time constraints.

3.1 Client Side

Suppose that we are given a set $\mathcal{T}$ of n independent frame-based real-time tasks. Each task τ_i in $\mathcal{T}$ (for $i = 1, 2, \ldots, n$) represents an execution unit, and it can be considered as an infinite sequence of instances, which called jobs. All the tasks have the same arrival time 0, period D and relative deadline D, i. e., with implicit deadlines. Each task $\tau_i \in \mathcal{T}$ is associated with the following timing parameters:

- **Worst-case *local* execution time C_i**: If task τ_i is decided to be executed locally on the client, the worst-case execution time required to finish task τ_i is up to C_i.
- **Setup time S_i:** is the execution time required on the client so that the required information can be sent to a corresponding server for offloading. It includes transmission time to the server and any local pre-processing operations such as data compression and transformation. As a result, when a task τ_i is offloaded, it has to be executed on the client for up to S_i amount of time, we say that τ_i is *settled for offloading*. After the setup time finishes on the client for offloading, the corresponding server can start task processing on its side.[1]
- **Round-trip offloading time I_i:** the interval length starting from the end of setting up S_i for task τ_i until getting the result from the server. If a server, or a processor, with a speed-up factor of α is dedicated for each offloaded task, then I_i is equal to the execution time on its side which can be computed as $\frac{C_i}{\alpha}$. Otherwise, when the server may handle more than one task, the server has its own scheduling policy and it provides a response time guarantee I_i for each task. The client contacts the server/s before scheduling to get the values of I_i. Subsection 3.3 describes how this value can be calculated.

[1] If the transmission time can be estimated with the worst case when the communication fabric is timing predictable, the worst-case transmission time can be used for guaranteeing the setup time. Otherwise, a pessimistic estimation can be used for providing soft timing analysis. For example, the transmission time can be computed as $\frac{Z}{\beta}$, where Z is the estimated maximum size of the offloaded data and β is the estimated lowest network bandwidth between the client and the server.

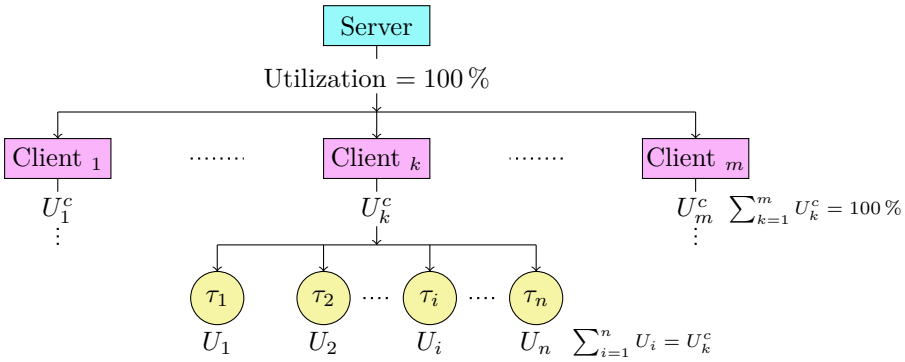


Figure 3 The distribution of the server's utilization.

Figure 2 shows these timing parameters for two tasks, where task τ_1 is locally executed and task τ_2 is offloaded. We assume that the results returned from the servers need very short post processing time, which is negligible. For instance, the returned results in our case study are the coordinates of the moving object or the distance between it and the cameras. Therefore, an offloaded task is said to meet the deadline/timing constraint if the result can return before the deadline. Our model is a special case of a general model (where each task has its own arrival time, period and relative deadline) that has never been considered before for offloading. Also, we prove that the offloading problem for this model is $\mathcal{NP}$-complete.

We say that a task is *locally executed* if it is processed on the client, while a task is called *offloaded* if it is processed on a server. The *finishing time* of a locally-executed task is the time that the task finishes its local execution. The finishing time of an offloaded task is its round-trip offloading time plus the time that this task is settled for offloading.

3.2 Server Side

The server can provide offloading services for more than one client, and the offloading decisions from a client will not control how the servers schedule the tasks. The servers can have their own scheduling policies to handle the tasks that are offloaded from the clients. They can decide how to provide the response time guarantee by themselves. For example, servers can use Earliest Deadline First (EDF) scheduling algorithm or resource reservation servers to ensure I_i. The response time guarantee can be either (1) hard if the scheduling in the servers and the communication fabric between the client and the servers are both timing predictable, or (2) soft if only the scheduling in the servers is timing predictable. However, for each case, when the client has the information about the round-trip offloading time, the open problem is how to meet the timing constraint by exploiting the services provided from the servers.

3.3 Calculating the Value of I_i

To calculate the value of I_i, the server has to provide a response time guarantee for the offloaded tasks. *Resource reservation* technique [2] (the resource here is the CPU of the server) can be used to provide such guarantee, and then satisfy the real-time constraints. In *Resource Reservation Server*[2] (RRS) model, the client can be given a bandwidth or a budget guarantee. In this paper, we consider the *Total Bandwidth Server* (TBS) model [14, 15] as a RRS on the server side. In this

[2] This is a logical server, inherited from the literature.

model, the server reserves a specific bandwidth (or utilization) U_k^c for each requesting client, if it is possible. U_k^c represents the fraction of the processor bandwidth of the server that is assigned to the client $_k$, where $1 \leq k \leq m$ and m is the total number of clients. The total reserved (or given) utilization for all clients should not exceed 100 %, i. e., $\sum_{k=1}^{m} U_k^c = 100\,\%$. Using this technique, the server is able to provide offloading services for more than one client without violating the real-time constraints.

For a client k with a given bandwidth of U_k^c and n tasks, the server allocates a TBS for each task τ_i with a utilization of U_i, such that $\sum_{i=1}^{n} U_i = U_k^c$ to preserve the system feasibility. Figure 3 shows how the utilization of the server can be distributed. A client $_k$ with a given utilization of U_k^c can divided it equally over all of its tasks, i. e., $U_i = \frac{U_k^c}{n}$, or with different ratios based on a specific algorithm. A task τ_i with a given utilization of U_i seems to be executed alone on a processor (TBS) which is $\frac{1}{U_i}$ times slower than the processor of the server. The TBS assigns an absolute deadline $d_i(t)$ for each offloaded task τ_i as follows:

$$d_i(t) = \max\{t, d_i(t^-)\} + \frac{R_i}{U_i}\,,$$

where t is the arrival time of the task at the server side, $d_i(t^-)$ is the absolute deadline of the previous instance (or frame), R_i is the remote execution time of the task (the execution time on the server side), and $d_i(0^-)$ is defined as 0. The offloaded tasks are scheduled on the server side using the Earliest Deadline First (EDF) algorithm based on the assigned TBS deadlines.

The candidate tasks for offloading are the tasks with $S_i + I_i \leq D$, i. e., feasible for offloading. Therefore, all the offloaded tasks finish within the deadline D (before the next frame), and then $t > d_i(t^-)$. Also, the task τ_i arrives at the server side immediately after the setting up time S_i (the transmission time is included in S_i). So, the round-trip offloading time can be calculated as $I_i = \frac{R_i}{U_i}$. In this way, each task can be executed independently of the behavior or the order of the other tasks.

3.4 Problem Definition

The problem explored in this paper is defined as follows:
Given a set $\mathcal{T}$ of n frame-based real-time tasks, the SElective Real-Time Offloading (SERTO) problem is to schedule the tasks and to decide when and what to offload without violating timing constraints for a client.

We consider two types of input instances of the SERTO problem, depending on whether the task execution ordering on the client is given or not. When the execution order is given and has to be preserved, we suppose that τ_i is executed on the client (either with S_i amount of time for offloading or C_i amount of time for local execution) before τ_j if $i < j$.

A schedule of a set $\mathcal{T}$ of tasks for the SERTO problem is an assignment of the executions of the tasks either on the client locally or on a remote server with computation offloading. A schedule is *feasible* if the finishing times of all locally-executed and offloaded tasks are within the deadline D. A scheduling algorithm is said to be *optimal offloading scheduling* algorithm if it is able to find a feasible schedule, if and only if one exists. Moreover, as we are dealing with frame-based real-time tasks, we always consider how to scheduling within a frame, starting from time 0. Therefore, the response time of a task is the same as the finishing time of a task.

Suppose that x_i is equal to 1 if task τ_i is decided to be offloaded; otherwise, x_i is 0. We use a vector $\vec{x}_n = (x_1, x_2, \ldots, x_n)$ to denote an *offloading decision* for the given n tasks.

Algorithm 1 GMF

```
1: t_1 ← 0;
2: for i = 1 to n do
3:    if S_i < C_i and t_i + S_i + I_i ≤ D then
4:       τ_i is assigned for offloading;
5:       t_{i+1} ← t_i + S_i;
6:    else if t_i + C_i ≤ D then
7:       τ_i is assigned for local computation;
8:       t_{i+1} ← t_i + C_i;
9:    else
10:      return "There is no feasible schedule";
11:   end if
12: end for
```

4 Greedy Minimum Finishing Algorithm

In this section we consider a set $\mathcal{T}$ of tasks with a given execution order. Let the tasks be indexed based on the given execution order from 1 to n, where n is the number of tasks. The problem is to decide whether a task should be executed locally or to be offloaded without violating timing constraints.

Under the given ordering, the SERTO problem can be solved by a greedy algorithm, called *Greedy Minimum Finishing (**GMF**)*. Suppose that t_i is the time when task τ_i can start to execute in the client, either offloaded or locally executed. The greedy algorithm simply makes the decision to offload a task τ_i if it is beneficial and feasible: that is, if $S_i < C_i$ (beneficial for offloading) and $t_i + S_i + I_i \leq D$ (feasible for offloading). If it is either not beneficial ($S_i \geq C_i$) or not feasible ($t_i + S_i + I_i > D$) for offloading, the algorithm checks if it can be executed locally, i. e., $t_i + C_i \leq D$. Otherwise, there is no feasible solution. Algorithm 1 presents the pseudo-code of the *GMF* algorithm. The time complexity of the algorithm is $O(|\mathcal{T}|)$.

▶ **Theorem 1.** *The GMF algorithm is an optimal offloading scheduling algorithm for the SERTO problem when the execution ordering is given.*

Proof. This theorem can be proved by an induction on the value t_i. We claim that t_{i+1} in the GMF algorithm is the *earliest* time on the client that τ_i finishes its local execution or is settled for offloading and τ_{i+1} can start to run by following the given ordering. For the **base case**, when $i = 1$, the statement is correct by definition.

Inductive step: Assume that t_{k+1} is the *earliest* time on the client that τ_k finishes its local execution or is settled for offloading and τ_{k+1} can start to run, for $k \geq 2$. There are two cases to run task τ_{k+1}:

- τ_{k+1} is offloaded: For such a case, we know that $t_{k+1} + S_{k+1} + I_{k+1} \leq D$ and $S_{k+1} \leq C_{k+1}$.
- τ_{k+1} is locally executed: For such a case, we know that either $t_{k+1} + S_{k+1} + I_{k+1} > D$ or $S_{k+1} > C_{k+1}$.

For both cases, we know that t_{k+2} is also the earliest time on the client that τ_{k+1} finishes its local execution or is settled for offloading and τ_{k+2} can start to run.

Clearly, if task τ_{k+1} cannot finish before the deadline D, the schedule is infeasible and there is no feasible schedule for the first $k + 1$ tasks. Therefore, based on the induction hypothesis, this theorem is proved. ◀

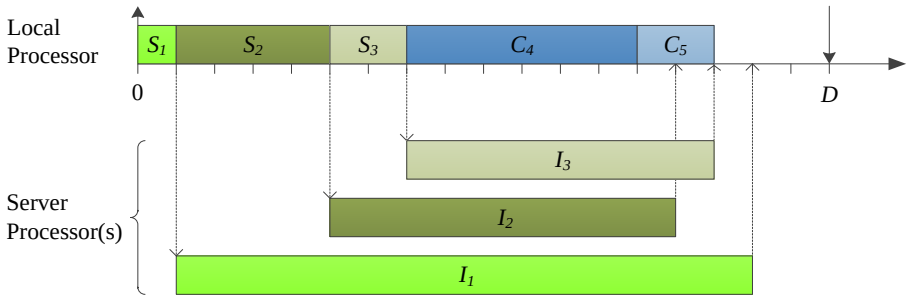


Figure 4 Example of optimal ordering for a set of tasks.

5 Hardness of the SERTO Problem

This section presents the $\mathcal{NP}$-completeness of the SERTO problem. Throughout this section, we implicitly consider the case that the execution ordering is not specified. Before presenting the hardness, we need the following lemma for deciding the optimal execution order on the client, provided that the computation offloading decisions have been made.

▶ **Lemma 2.** *If the execution order is not specified, all the offloaded tasks should be executed before any locally-executed task.*

Proof. Suppose that τ_i is decided to be locally executed, while task τ_j is to be offloaded. If a feasible schedule executes τ_i on the client before the next task τ_j in the schedule starts on the client, we can also swap the execution ordering of τ_i and τ_j on the client to be still feasible. Let τ_i starts to run on the client at time t at the original schedule. So, the total finishing time of the two tasks τ_i and τ_j is equal to $t + C_i + S_j + I_j \leq D$ (because the schedule is feasible). After swapping, the finishing time of τ_j is now at most $t+S_j+I_j$, the finishing time of τ_i now is at most $t+S_j+C_i$, and the total finishing time of both tasks is at most $\max\{t+S_j+I_j, t+S_j+C_i\}$. Therefore, the the total finishing time of the two tasks after swapping is less than before swapping without violating the feasibility of the schedule, because $\max\{t + S_j + I_j, t + S_j + C_i\} < t + C_i + S_j + I_j \leq D$.

After swapping, the worst-case finishing time of the other tasks does not change. By repeating the above procedure, we know that the statement in the lemma holds. ◀

When the offloading decision $\vec{x}_n$ for the tasks is known, we define $d_i = x_i(D-I_i)+(1-x_i)D$ as the *virtual offloaded deadline*. If there is a feasible schedule based on an offloading decision $\vec{x}_n$, then executing the tasks by following the order of $d_i = x_i(D - I_i) + (1 - x_i)D$ non-decreasingly is also a feasible schedule. This ordering is called Earliest Virtual Offloaded Deadline First (**EVODF**). Please refer to Figure 4, as an illustration example for an optimal ordering for a given set of five tasks. We have the following lemma for **EVODF**.

▶ **Lemma 3.** *If the execution order is not specified and there is a feasible schedule based on the offloading decisions, the schedule by using **EVODF** is also a feasible schedule.*

Proof. Suppose that a given schedule is feasible. By Lemma 2, we can reorder the execution ordering, such that any locally-executed task should be executed after the offloaded tasks, to maintain the feasibility. Now, for two consecutively offloaded tasks τ_i and τ_j in that feasible schedule, if $d_i > d_j$ and τ_i starts its execution at time t on the client before τ_j, we can still swap these two jobs to maintain the feasibility. Suppose that the server returns the result of task τ_i at time $f_i = t + S_i + I_i$ and τ_j at time $f_j = t + S_i + S_j + I_j$, respectively. By the definition of $d_i > d_j$, we know that $I_i < I_j$.

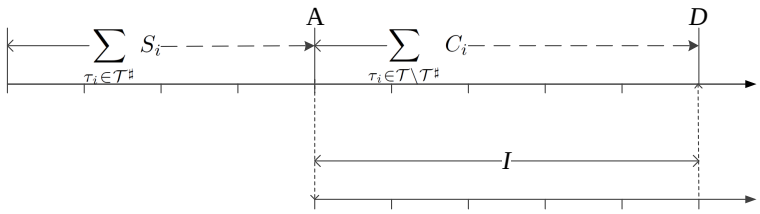


Figure 5 Illustration for the $\mathcal{NP}$-completeness proof of the SERTO problem.

Clearly, due to the feasibility before swapping, we know that $f_i = t + S_i + I_i \leq D$ and $f_j = t+S_i+S_j+I_j \leq D$. Therefore, the finishing time f'_j of τ_j after swapping is $f'_j = t+S_j+I_j \leq D$, and the finishing time of τ_i after swapping is $f'_i = t + S_j + S_i + I_i < t + S_j + S_i + I_j \leq D$. Clearly, after swapping, the worst-case finishing time of the other tasks does not change.

By repeating the above procedure, we know that the schedule by using (**EVODF**) is also a feasible schedule. ◀

Based on Lemma 3, we have the following lemma for testing whether an offloading decision results in a feasible schedule or not.

▶ **Lemma 4.** *Suppose that tasks $\tau_i \in \mathcal{T}$ for $i = 1, 2, ..., n$ are ordered non-decreasingly according to $D - I_i$. An offloading decision $\vec{x}_n$, $x_i = \{0, 1\}$, results in a feasible schedule (by using **EVODF**) if and only if (a) $\sum_{j=1}^{n} x_j S_j + (1 - x_j)C_j \leq D$ and (b) $x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D, \forall k = 1, 2, \ldots, n$.*

Proof. This comes directly from Lemma 3. ◀

Now, we will prove the $\mathcal{NP}$-completeness of the SERTO problem when the execution ordering is unknown.

▶ **Theorem 5.** *The SERTO problem is $\mathcal{NP}$-complete if the execution order is not given.*

Proof. Due to Lemma 3, verifying whether an offloading decision with **EVODF** scheduling is feasible or not can be done in polynomial time. Therefore, the SERTO problem is in $\mathcal{NP}$. The $\mathcal{NP}$-completeness can be proved by a reduction from the SUBSET SUM problem [4]: Given a set of integers $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ and an integer A, the problem is to find a subset $\mathcal{V}^\sharp$ of $\mathcal{V}$ such that $\sum_{v_i \in \mathcal{V}^\sharp} v_i = A$. For each v_i in $\mathcal{V}$, the reduction creates τ_i with

- $C_i = 2v_i$, $S_i = v_i$,
- $I_i = I = 2((\sum_{v_j \in \mathcal{V}} v_j) - A)$, and
- $D = 2\sum_{v_j \in \mathcal{V}} v_j - A$.

Since all the tasks have the same round-trip offloading time and by Lemma 4, for an offloaded task set $\mathcal{T}^\sharp$ (with the corresponding set $\mathcal{V}^\sharp$), the resulting **EVODF** schedule is feasible if and only if $\sum_{\tau_i \in \mathcal{T}^\sharp} S_i \leq D - I$ and $\sum_{\tau_i \in \mathcal{T}^\sharp} S_i + \sum_{\tau_i \in \mathcal{T} \setminus \mathcal{T}^\sharp} C_i \leq D$, see Figure 5.

By the construction of task set $\mathcal{T}$, we know that

$$\sum_{v_i \in \mathcal{V}^\sharp} v_i = \sum_{\tau_i \in \mathcal{T}^\sharp} S_i \leq D - I = A \tag{1}$$

and

$$\begin{aligned} & \sum_{\tau_i \in \mathcal{T}^\sharp} S_i + \sum_{\tau_i \in \mathcal{T} \setminus \mathcal{T}^\sharp} C_i \leq D \\ \Rightarrow \quad & 2\sum_{v_i \in \mathcal{V}} v_i - \sum_{v_i \in \mathcal{V}^\sharp} v_i \leq 2\sum_{v_j \in \mathcal{V}} v_j - A \quad \Rightarrow \quad \sum_{v_i \in \mathcal{V}^\sharp} v_i \geq A. \end{aligned} \tag{2}$$

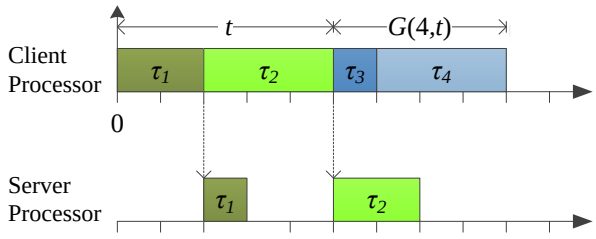


Figure 6 An example for illustrating the dynamic programming parameters.

Therefore, by (1) and (2), we know that there exists such a $\mathcal{V}^\sharp$ with $\sum_{v_i \in \mathcal{V}^\sharp} v_i = A$, if and only if the reduced input instance for the SERTO problem has a feasible schedule by offloading the corresponding task set $\mathcal{T}^\sharp$ created from $\mathcal{V}^\sharp$.

Since the reduction is in linear time complexity, we know that the SERTO problem is $\mathcal{NP}$-complete. ◀

6 Algorithms for Tasks without Specified Ordering

In this section, we consider real-time tasks without any specified ordering, and present our proposed scheduling algorithms for the SERTO problem. We will present a pseudo-polynomial-time algorithm and an approximation algorithm with polynomial-time complexity. At the end of the section, we will extend our algorithms to find the minimum D for executing the frame-based real-time tasks to maximize the performance.

6.1 Dynamic Real-time Scheduling Algorithm

Based on dynamic programming, we introduce *Dynamic Real-time Scheduling (*DRS) algorithm to find a feasible solution for the SERTO problem. At the beginning, tasks $\tau_i \in \mathcal{T}$ for $i = 1, 2, ..., n$ are ordered non-decreasingly according to $D - I_i$.

An offloading decision $\vec{x}_i$ for the first i tasks, i. e., $\{\tau_1, \tau_2, \ldots, \tau_i\}$, is said *partially feasible for offloading* (or a *partially feasible offloading decision*) if the offloaded tasks can finish the execution in the servers before the given deadline D. Similar to Lemma 4, we know that a vector $\vec{x}$ is partially feasible for offloading for $\{\tau_1, \tau_2, \ldots, \tau_i\}$ if and only if $x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D, \forall k = 1, 2, \ldots, i$.

Our strategy is to build a dynamic programming table by maintaining and storing some scheduling results for the partially feasible offloading decisions for the first i tasks. Specifically, among all the partially feasible offloading decisions for $\{\tau_1, \tau_2, \ldots, \tau_i\}$, let $G(i, t)$ be the minimum total local execution time for the locally-executed tasks under the constraint that the total setup time for the offloaded tasks in $\{\tau_1, \tau_2, \ldots, \tau_i\}$ is less than or equal to t. Figure 6 presents an example of four tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ with the dynamic programming parameters, where $\{\tau_1, \tau_2\}$ are offloaded and $\{\tau_3, \tau_4\}$ are executed locally. That is, for a given i and t, the value $G(i, t)$ is the objective function of the following integer linear programming (ILP):

$$\text{minimize} \sum_{j=1}^{i} (1 - x_j) C_j \tag{3a}$$

$$\text{s.t.} \sum_{j=1}^{i} x_j S_j \leq t \tag{3b}$$

$$x_k I_k + \sum_{j=1}^{k} x_j S_j \leq D \qquad \forall k = 1, 2, \ldots, i \tag{3c}$$

$$x_j \in \{0, 1\} \qquad \forall j = 1, 2, \ldots, i. \tag{3d}$$

For notational brevity, when the above ILP has no feasible solution, $G(i,t)$ is defined as ∞. Moreover, $G(i,t) = \infty$ when $t < 0$.

The optimal solution for (3) is also called *optimal partially offloaded decision* when the total local setup time for the offloaded tasks in these i tasks is no more than t. Clearly, when i is 1, we know that

$$G(1,t) = \begin{cases} 0 & \text{if } S_1 \leq t \leq D - I_1 \\ C_1 & \text{otherwise} \end{cases} . \tag{4}$$

Instead of solving the above ILP for building $G(i,t)$, the construction of $G(i,t)$, for $i \geq 2$, can be achieved by using the following recurrence:

$$G(i,t) = \min \begin{cases} \begin{cases} G(i-1, t-S_i) & \text{if } t \leq D - I_i \\ \infty & \text{otherwise} \end{cases} \\ \\ G(i-1,t) + C_i \end{cases} . \tag{5}$$

Suppose that $\vec{x}^{\sharp}_{i-1}$ is the corresponding partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ with respect to the result in $G(i-1, t-S_i)$. Similarly, let $\vec{x}^{\dagger}_{i-1}$ be the corresponding partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ with respect to the result in $G(i-1,t)$. The recursive function in (5) represents the selection of the minimum solution by comparing two cases:

- Case 1: task τ_i is offloaded when its local setup execution finishes at time t. For such a case, if $t + I_i > D$, offloading τ_i is an infeasible offloading decision; otherwise, we consider the offloading decision $\vec{x}^{\sharp}_{i-1}$ for the first $i-1$ tasks, in which the total local execution time of this solution is as the same as that in $\vec{x}^{\sharp}_{i-1}$, i. e., $G(i-1, t-S_i)$.
- Case 2: task τ_i is locally executed. Therefore, we consider the offloading decision $\vec{x}^{\dagger}_{i-1}$ for the first $i-1$ tasks. As a result, the total local execution time of this solution is the sum of C_i and the total local execution time in solution $\vec{x}^{\dagger}_{i-1}$. That is, $C_i + G(i-1,t)$.

We assume in this subsection that S_i is a non-negative integer for a task τ_i in $\mathcal{T}$. The standard dynamic-programming procedure can be applied by constructing a table with n rows for $i = 1, 2, \ldots, n$ and $\lfloor D \rfloor + 1$ columns for $t = 0, 1, 2, \ldots, \lfloor D \rfloor$.

▶ **Lemma 6.** *For given integers i and t, the recursive function defined in (4) and (5) computes the optimal solution for $G(i,t)$.*

Proof. The optimality is proved by induction on i. For the **base case**, $G(1,t) = 0$ if there is enough time for feasible offloading of task τ_1. Otherwise τ_1 is locally executed, and then $G(1,t) = C_1$, which is optimal.

Inductive step: Assume that $G(i-1,t)$ is optimal for the subproblem for the first $i-1$ tasks with $i \geq 2$ for any given $t \geq 0$ (i. e., the ILP described in (3)). Recall that the two offloading decisions $\vec{x}^{\sharp}_{i-1}$ and $\vec{x}^{\dagger}_{i-1}$ which represent the optimal partially offloading decisions for $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ when the total local setup time for the offloaded tasks in these $i-1$ tasks is no more than $t - S_i$ and t, respectively.

Suppose for contradiction that $\vec{x}^*_i$ is a partially feasible offloading decision for $\{\tau_1, \tau_2, \ldots, \tau_i\}$ in which $\sum_{j=1}^{i} x^*_j S_j \leq t$ and $\sum_{j=1}^{i} (1 - x^*_j) C_j < G(i,t)$. There are two cases for task τ_i in $\vec{x}^*_i$.

- **Case 1:** x^*_i is 0 (τ_i is locally executed) in $\vec{x}^*_i$. Clearly, under the assumption $\sum_{j=1}^{i} (1 - x^*_j) C_j < G(i,t)$, we know that

$$\sum_{j=1}^{i} C_j (1 - x^*_j) = C_i + \sum_{j=1}^{i-1} C_j (1 - x^*_j) < G(i,t) \leq_1 C_i + \sum_{j=1}^{i-1} C_j (1 - x^{\dagger}_j),$$

where $\leq_1$ comes from the construction of $G(i,t)$ in (5). Hence, the offloading decision $\vec{x}^*_{i-1}$ by excluding x^*_i from $\vec{x}^*_i$ is a partially feasible offloading decision for the first $i-1$ tasks with $\sum_{j=1}^{i-1} S_j x^*_j \leq t$ and $\sum_{j=1}^{i-1} C_j(1-x^*_j) < \sum_{j=1}^{i-1} C_j(1-x^\dagger_j)$, which contradicts the optimality of $G(i-1,t)$.

- **Case 2:** x^*_i is 1 (τ_i is offloaded) in $\vec{x}^*_i$. Clearly, we know that $S_i \leq t \leq D - I_i$ for such a case; otherwise, $\vec{x}^*_i$ is not a partially feasible offloading decision. With this case, we know that

$$\sum_{j=1}^{i} C_j(1-x^*_j) = \sum_{j=1}^{i-1} C_j(1-x^*_j) < G(i,t) \leq \sum_{j=1}^{i-1} C_j(1-x^\sharp_j).$$

Therefore, the offloading decision $\vec{x}^*_{i-1}$ by excluding x^*_i from $\vec{x}^*_i$ is a partially feasible offloading decision for the first $i-1$ tasks with $\sum_{j=1}^{i-1} S_j x^*_j \leq t - S_i$ and $\sum_{j=1}^{i-1} C_j(1-x^*_j) < \sum_{j=1}^{i-1} C_j(1-x^\sharp_j)$, which contradicts the optimality of $G(i-1, t-S_i)$.

Hence, based on the induction hypothesis, the lemma is proved. ◀

Now, based on Lemma 6, for an input task set $\mathcal{T}$, to verify whether a feasible schedule exists for the SERTO problem or not, we just have to check whether there exists $0 \leq t \leq D$ with $G(n,t) + t \leq D$.

▶ **Theorem 7.** *There exists t with $G(n,t) + t \leq D$ if and only if there exists a feasible schedule for the SERTO problem.*

Proof. If: Suppose $\vec{x}_n$ is the corresponding offloading decision for a feasible schedule of the SERTO problem. Let ℓ be the maximum index with $x_\ell = 1$. That is, x_j is 0 for $j > \ell$. As the schedule is feasible, we know that $\vec{x}_n$ is also a partially feasible offloaded decision when t is set to $\sum_{j=1}^{\ell} S_j x_j$. Therefore, based on Lemma 6, we have $\sum_{j=1}^{n} C_j(1-x_j) \geq G(n, \sum_{j=1}^{\ell} S_j x_j)$. The necessary condition for being a feasible schedule for the SERTO problem, is $\sum_{j=1}^{\ell} S_j x_j + \sum_{j=1}^{n} C_j(1-x_j) \leq D$. This implies that $\sum_{j=1}^{\ell} S_j x_j + G(n, \sum_{j=1}^{\ell} S_j x_j) \leq D$. Therefore, when t is $\sum_{j=1}^{\ell} S_j x_j$, we know that $G(n,t) + t \leq D$.

Only-If: Suppose t^* is with $G(n,t^*) + t^* \leq D$. We can backtrack the dynamic programming table to obtain an offloading decision $\vec{x}^*_n$ for the given n tasks such that it satisfies the constraints described in (3) when t is set to t^* and i is set to n. Since $G(n,t^*) + t^* \leq D$, based on Lemma 4, we know that the resulting schedule by using **EVODF** scheduling policy is a feasible schedule. ◀

▶ **Theorem 8.** *The DRS algorithm is an optimal offloading scheduling algorithm with time complexity $O(n \log n + nD)$ for the SERTO problem when there is no specified execution ordering.*

Proof. The optimality comes from Theorem 7. The time complexity of constructing $G(i,t)$ for $i = 1, 2, \ldots, n$ and $t = 0, 1, 2, \ldots, \lfloor D \rfloor$ is $O(n \log n + nD)$, since it takes $O(n \log n)$ to sort task set $\mathcal{T}$ and $O(nD)$ to build the dynamic-programming table. For back-tracking a solution from an entry t with $G(n,t^*) + t^* \leq D$, the time complexity is $O(n)$. Therefore, the overall time complexity is $O(n \log n + nD)$, which is pseudo-polynomial time. The space complexity is $O(nD)$, but it can be improved to $O(D)$ by discarding the entries $G(i-2,t)$ when building $G(i,t)$. ◀

6.2 Approximation for DRS Algorithm

As the SERTO problem is $\mathcal{NP}$-complete, solving the problem in polynomial time is not possible unless $\mathcal{NP} = \mathcal{P}$. To allow a polynomial-time algorithm, some approximation is needed. This subsection presents a methodology to reduce the time complexity so that the user can trade the complexity with the approximation of the derived solution.

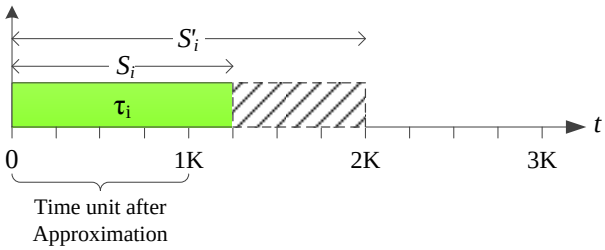


Figure 7 Approximation example.

Let $K = \frac{\epsilon D}{n}$ be the time unit after approximation, where $\epsilon > 0$ is a user-specified parameter that determines the approximation granularity. This means that a time unit after approximation is equal to K amount of time before approximation. The exact algorithm requires the assumption that all the timing parameters are integers and has pseudo-polynomial complexity. However, if the timing parameters are real numbers, the algorithm will not work. In this case, the real numbers can be rounded up to the nearest integers. But, this will affect the accuracy of the algorithm. Also, in the case of a large value of D, the time and space complexities of the algorithm will be high. Therefore, the approximation is used to trade the accuracy with time and space complexities for both cases, depending on the user parameter ϵ. Both complexity and accuracy are inversely proportional to the value of ϵ, which determines the value of K. If the value of K is less than 1, the timing parameters are scaled up to increase the accuracy. But, it will also increase the complexity of the algorithm. On the other hand, if the value of K is greater than 1, the timing parameters are scaled down which is used to reduce the complexity of the algorithm for a large value of D. As a consequence, we will get a less accurate result. For $K = 1$, the approximation does not have any effect.

For each task τ_i, we construct a corresponding task τ'_i as follows:

- $S'_i = K \left\lceil \frac{S_i}{K} \right\rceil$ (rounded up to the nearest time unit, i. e. integer multiples of K),
- $I'_i = I_i - (S'_i - S_i)$, and
- $C'_i = C_i$.

Figure 7 shows an approximation example, where the time unit after approximation (K) is equal to 4 and the setup time S_i is rounded-up to the next time unit ($2K$).

Let $\mathcal{T}'$ be the resulting task set after transformation. Moreover, we also set D' either to D or to $(1+\epsilon)D$, to be explained later. As all the setup times are integer multiples of K, we can construct the dynamic programming table by considering only the integer multiples of K. Therefore, we define $G'(i,t)$ as the minimum total local execution time for the locally executed tasks under the constraint that the total *rounded-up* setup time for the offloaded tasks in $\{\tau'_1, \tau'_2, \ldots, \tau'_i\}$ is less than or equal to $t \cdot K$

$$G'(1,t) = \begin{cases} 0 & \frac{S'_1}{K} \leq t \leq \frac{D'-I'_i}{K} \\ C'_1 & otherwise \end{cases} . \tag{6}$$

For $i \geq 2$,

$$G'(i,t) = min \begin{cases} \begin{cases} G'(i-1, t - \frac{S'_i}{K}) & \forall t \leq \frac{D'-I'_i}{K} \\ \infty & otherwise \end{cases} \\ \\ G'(i-1,t) + C'_i \end{cases} . \tag{7}$$

The time t in the equations above represents the time after approximation, which is represented in K unit of time. Therefore, the timing parameters S'_i and $D' - I'_i$ should be divided by K to be consistent with the new time scale.

▶ **Lemma 9.** *For given integers i and t, the recursive function defined in (6) and (7) computes the optimal solution for a partially feasible offloading decision for the first i tasks in $\mathcal{T}'$.*

Proof. This is similar to the proof for Lemma 6 ◀

The following theorem shows the feasibility by adopting the dynamic programming for the resulting solutions.

▶ **Theorem 10.** *When D' is set to D, and there exists t with $0 \le t \le \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t)+t\cdot K \le D'$, the offloading decision by backtracking the dynamic programming table built for $\mathcal{T}'$ is a feasible schedule of the original task set $\mathcal{T}$ by using* **EVODF** *scheduling policy.*

Proof. This basically comes directly from the definition of $\mathcal{T}'$. Suppose that $\vec{x}_n$ is an offloading decision for such a t after by backtracking the dynamic programming table built for $\mathcal{T}'$. Therefore, with the fact $\sum_{j=1}^{n} x_j S'_j \le t \cdot K$, we know that

$$t \cdot K \ge \sum_{j=1}^{n} x_j S'_j \ge \sum_{j=1}^{n} x_j S_j,$$

and, for all $k = 1, 2, \ldots, n$, as $I'_k + S'_k$ is equal to $I_k + S_k$ and $x_k I'_k + \sum_{j=1}^{k} x_j S'_j \le D$, we have

$$D \ge x_k(I_k + S_k) + \sum_{j=1}^{k-1} x_j S'_j \ge x_k I_k + \sum_{j=1}^{k} x_j S_j.$$

Therefore, we know that $\vec{x}_n$ is a partially feasible offloading decision with $\sum_{j=1}^{n} x_j S_j \le t \cdot K$. Since $G'(n,t) + t \cdot K \le D'$, the statement holds due to Lemma 4. ◀

▶ **Theorem 11.** *If there exists a feasible schedule for $\mathcal{T}$, then there exists t with $0 \le t \le \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t) + t \cdot K \le D'$ when D' is set to $(1+\epsilon)D$.*

Proof. Suppose that $\vec{x}^*_n$ is the offloading decision of a feasible schedule for $\mathcal{T}$. By Lemma 4, $\sum_{j=1}^{n} x^*_j S_j + (1 - x^*_j) C_j \le D$ and $x^*_k I_k + \sum_{j=1}^{k} x^*_j S_j \le D$ for $k = 1, 2, \ldots, n$.

By the definition of $\mathcal{T}'$ and $K = \frac{\epsilon D}{n}$, we know that

$$\sum_{j=1}^{n} x^*_j S'_j \le \sum_{j=1}^{n} x^*_j (S_j + K) \le K \cdot n + \sum_{j=1}^{n} x^*_j S_j \le \epsilon D + \sum_{j=1}^{n} x^*_j S_j. \tag{8}$$

Similarly, for $k = 1, 2, \ldots, n$, we have

$$x^*_k I_k + \sum_{j=1}^{k} x^*_j S_j \le x^*_k (I_k + S_k) + \sum_{j=1}^{k-1} x^*_j S'_j \le x^*_k (I_k + S_k) + \epsilon D + \sum_{j=1}^{k-1} x^*_j S_j \le (1+\epsilon) D.$$

Let t' be $\frac{\sum_{j=1}^{n} x^*_j S'_j}{K}$, in which t' is an integer and $t' \le \left\lfloor \frac{D'}{K} \right\rfloor$. Then, by Lemma 9 for the optimality in $G'(i,t')$, we know that $G'(n,t') \le \sum_{j=1}^{n} (1 - x^*_j) C_j$. Therefore, together with (8),

we know that

$$G'(n,t') + t' \cdot K = G'\left(n, \frac{\sum_{j=1}^{n} x_j^* S_j'}{K}\right) + \sum_{j=1}^{n} x_j^* S_j'$$

$$\leq \sum_{j=1}^{n}(1-x_j^*)C_j + \sum_{j=1}^{n} x_j^* S_j + \epsilon D \leq (1+\epsilon)D = D',$$

which proves the theorem. ◀

We now analyze the time complexity.

▶ **Theorem 12.** *For a given $\epsilon > 0$ and D', evaluating whether there exists exists t with $0 \leq t \leq \left\lfloor \frac{D'}{K} \right\rfloor$ and $G'(n,t) + t \cdot K \leq D'$ is with time complexity $O(\frac{n^2}{\epsilon})$.*

Proof. The construction of task set $\mathcal{T}'$ takes only $O(n)$. The construction of $G'(i,t)$ requires $O(n\frac{D}{K}) = O(\frac{n^2}{\epsilon})$, since K is set to $\frac{\epsilon D}{n}$. ◀

6.3 Maximizing the Sampling Rate

The SERTO problem so far is for determining a feasible schedule if there exists. Another extension is to minimize the deadline/period D for the frame-based real-time tasks so that the sampling rate of the frame-based tasks can be maximized. The DRS algorithm can be adopted to find the optimal value of D with a binary search. Suppose that D^{lower} and D^{upper} are the lower and upper bounds of the feasible deadlines in the current iteration in the binary search, respectively. Initially, D^{upper} is $\sum_{i=1}^{n} C_i$ and D^{lower} is $\sum_{i=1}^{n} \min\{S_i, C_i\}$.

Moreover, suppose that $\vec{x}_n$ is the offloading decision for a feasible schedule by setting D to $\frac{D^{lower}+D^{upper}}{2}$. We also know that setting D to

$$D^\sharp = \max\left\{ \begin{array}{l} \max_{1\leq k\leq n}\{x_k I_k + \sum_{j=1}^{k} x_j S_j\}, \\ \sum_{j=1}^{n}(1-x_j)C_j + x_j S_j \end{array} \right\}$$

is also feasible. Therefore, if such an offloading decision $\vec{x}_n$ is found, the efficiency, with respect to the time complexity, of the binary search can be further improved by setting the next D to $\frac{D^{lower}+D^\sharp}{2}$, as any $D > D^\sharp$ has feasible schedules. Clearly, the whole procedure is still with pseudo-polynomial time.

When the approximation in Section 6.2 is adopted, the above binary search still works with polynomial-time complexity. Due to Theorems 10 and 11, the derived solution is at most $(1+\epsilon)$ times the minimum feasible deadline of the input instance, by ignoring the error due to the termination condition of the binary search.

7 Experimental Results

In our experiments, our *DRS* algorithm, with and without approximation, is evaluated by adopting a surveillance system as a case study and synthesis workload simulation.

7.1 Case Study of a Surveillance System

We use a surveillance system that performs four real-time tasks to evaluate our *DRS* algorithm, and compare it with Nimmagadda et al. [12] algorithm and by offloading all the tasks. The system captures two images at the same time, left and right, periodically. Left and right images are used for a stereo vision task. For the other tasks, one image is used for processing. The tasks are frame-based real-time tasks and independent, described as follows:

Table 1 Timing parameters of case study tasks (ms).

τ_i	Description	C_i	With encoding			Without encoding		
			S_i	I_i - Multi.	I_i - Single	S_i	I_i - Multi.	I_i - Single
τ_1	Motion Detection	30	31	33	117	7	21	141
τ_2	Object Recognition	220	3	102	102	2	102	102
τ_3	Stereo Vision	88	34	47	115	16	41	127
τ_4	Motion Recording	18	31	29	115	7	14	148

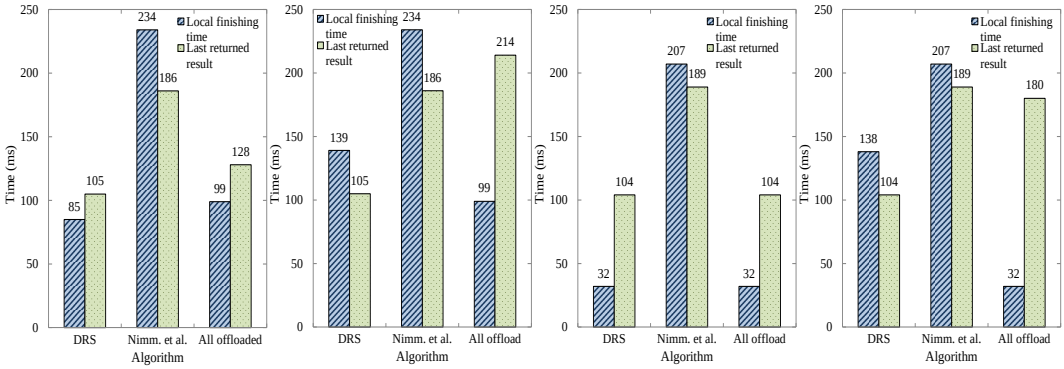
- **Motion Detection:** The motion is detected using the background subtraction technique [13]. The system computes the running average of the captured frames, and each new frame is subtracted from the moving average. Then, the output is processed to get the contours of the moving objects [5].
- **Object Recognition:** It is used to recognize a given object from the input image. Scale Invariant Feature Transform (SIFT) method [11] is used to extract features, which are not affected by object size, position or rotation.
- **Stereo Vision:** Stereo vision is used to generate a depth map for left and right images to calculate the distance between the surveillance system and the object of interest. Stereo imaging [1] is adopted in the implementation.
- **Motion Recording:** It records video for detected motion for further human check.

The system remains idle until a motion is detected. Then, it starts executing all the tasks above. Before sending an image to the server(s), scaling, encoding, or both of them may be performed on the image. Although scaling and encoding can reduce the size of the transfered image for reducing the communication overhead, they consume more time on the local device for scaling and encoding. The time used for scaling, encoding, and sending on the client for a task is considered as the setup time in our case study. For the server side, we consider two cases. First, a dedicated server (or processor) for each task, if offloaded. Second, we assume that we have only one server where a scheduling algorithm is used to schedule all the offloaded tasks, in which I_i may be larger than C_i for some task τ_i.

We consider four scenarios: (*Scenario 1*) images are encoded before sending and a dedicated server is used for each offloaded task (multiple servers), (*Scenario 2*) images are encoded before sending and only one server is used for all the offloaded tasks (single server), (*Scenario 3*) images are sent without encoding and a dedicated server is used for each offloaded task, and (*Scenario 4*) images are sent without encoding and only one server is used for all the offloaded tasks.

Timing parameters for the tasks in the four scenarios are given in Table 1, where the time values are in milliseconds based on measurements. If the system performs all the tasks locally, they will finish by 356 ms, which will be considered as the deadline (sampling period) in our case study here. We explore the three offloading approaches to reduce the local finishing time, i. e., increase the sampling rate.

Figure 8 shows the total local finishing time on the client side, and the time at which the last result returns back from the server side. In Scenario 1, tasks τ_2 and τ_3 are offloaded in both *DRS* and Nimmagadda et al. [12] algorithms. Although the offloading decisions in the previous scenario are the same for both algorithms, the total finishing time in *DRS* is shorter. This is because *DRS* algorithm continues local execution after offloading, while Nimmagadda et al. [12] remains idle waiting for the results from the server side. The decision of the Nimmagadda et al. [12] algorithm in Scenario 2 is the same as in Scenario 1. It does not change by having multiple servers because Nimmagadda et al. [12] algorithm remains idle during offloading. *DRS* algorithm just offloads task τ_2 in Scenario 2 and 4.



(a) Scenario 1 **(b)** Scenario 2 **(c)** Scenario 3 **(d)** Scenario 4

Figure 8 Case study results.

In Scenarios 3, *DRS* algorithm offloads all the tasks because their setup time S_i is less than their local execution time C_i, and they are feasible for offloading. Nimmagadda et al. [12] algorithm offloads all the tasks except task τ_4 in Scenario 3 and 4, because its local time is less than the summation of the expected remote execution time and the data transfer time. We observe that all the three evaluated algorithms reduce the local finishing time, but our algorithm has the minimum finishing time in all scenarios.

7.2 Simulation Setup and Results

We also perform simulations by using synthetic workload for task τ_i generated as follows:

- C_i: Randomly generated integer values from 1 to 50 ms with uniform distribution.
- S_i: Randomly generated integer values from 1 to C_i with uniform distribution.
- I_i: $I_i = \frac{C_i}{\alpha}$, where α is the speed-up factor of the server, i. e., the response time from the server is α times faster than the execution time of the local client. α is randomly generated such that $0 < \alpha \leq m$, where m is the maximum value of α.

We perform 100 rounds in the experiment. In each round, a set of 25 frame-based real-time tasks is randomly generated according to the above conditions. Each task set is evaluated by ten different settings according to m values, where $m = \{0.005, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2, 4, 8\}$. By using small m values, we simulate servers that require longer response time than the local execution time for tasks. While by using large m values, we simulate servers that are faster than the client and can process almost immediately for any offloaded tasks.

The *normalized finishing time reduction* of an algorithm for a task set is the finishing time for the task set execution after using the derived schedule divided by the finishing time for the same task set if all tasks are executed locally. Also, the *normalized sampling period* is the finishing time for the input task set using the approximation *DRS* algorithm described in 6.2 divided by the finishing time for the same task set using the *DRS* algorithm.

For the rest of this section, we will discuss the simulation results for the three offloading approaches using the task sets described above. Also, we evaluate the approximation *DRS* algorithm described in Section 6.2. Figure 9 shows the number of offloaded tasks for different m values. Nimmagadda et al. [12] algorithm offloads tasks only when the server is faster than the client. But, *DRS* algorithm offloads tasks even to server(s) with longer response time, while they are feasible. Also, we observe that the number of the offloaded tasks increases proportionally to m value.

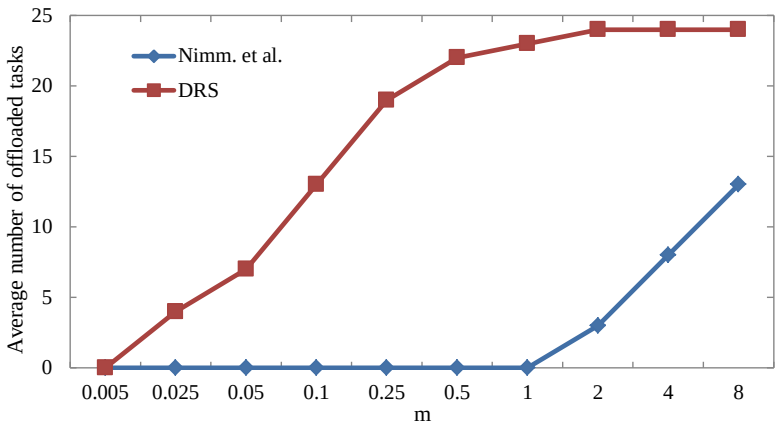


Figure 9 Number of offloaded tasks for synthesized tasks.

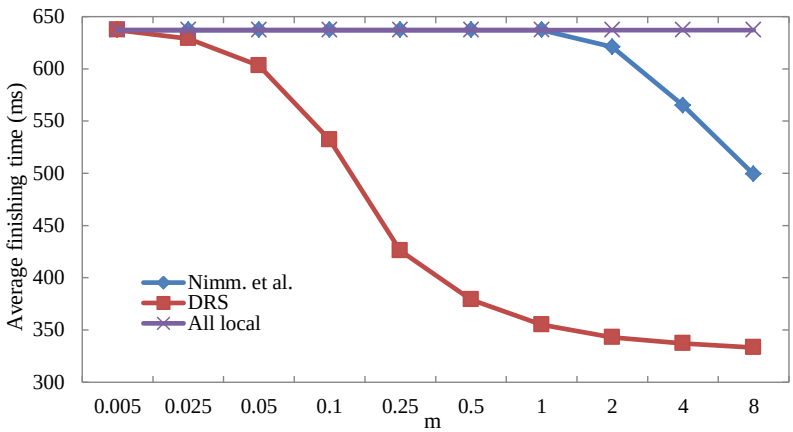


Figure 10 Finishing time for synthesized tasks.

Figure 10 illustrates the average finishing time for the generated task sets. Nimmagadda et al. [12] algorithm reduces the local execution time only when the server is faster than the client, because it doesn't offload tasks with $m \leq 1$ as shown in Figure 9. The improvement of *DRS* algorithm, compared to Nimmagadda et al. [12] algorithm, is up to 44.7 %.

Figure 11 shows the average normalized finishing time reduction. Again, Nimmagadda et al. [12] algorithm does not help in finishing time reduction for the same reason in Figures 9 and 10. Furthermore, the finishing time reduction in *DRS* algorithm is more than in Nimmagadda et al. [12] algorithm because Nimmagadda et al. [12] algorithm remains idle during offloading. In Figure 11, offloading all the tasks is not useful for $m \leq 2$ and the finishing time exceeds the summation of local execution for all the tasks, because the round-trip offloading time for most of the tasks is relatively large. *DRS* algorithm reduces the finishing time in all cases because it offloads only the beneficial and optimal tasks for offloading. The average finishing time using *DRS* algorithm is reduced up to 52 % of the local execution.

Figure 12 shows the average execution time of DRS algorithm, where n is the number of input tasks. The algorithm is evaluated with different number of input tasks (5, 10, 15, 20 and 25) and different deadlines (300, 400, 500, 600 and 700 ms). As the deadline value increases, the average execution time also increases, but more rapidly for larger number of tasks. Nevertheless, the execution time of the algorithm is very short and negligible relative to the deadline.

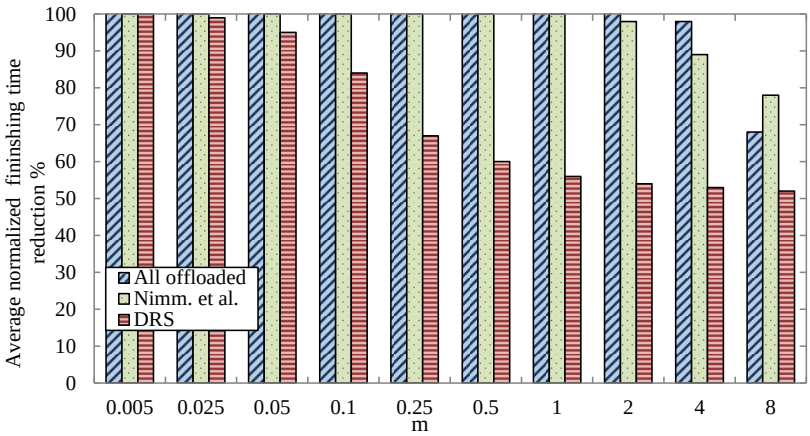


Figure 11 Finishing time reduction for synthesized tasks.

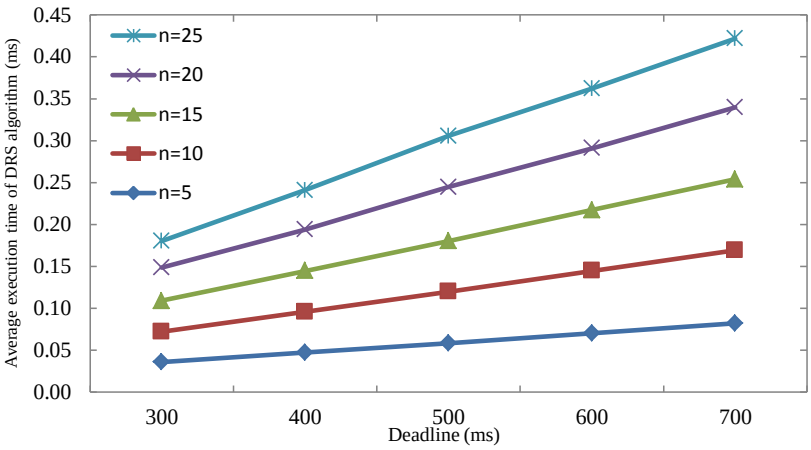


Figure 12 Execution time of DRS algorithm.

The above results are based on the *DRS* algorithm. Now, we will present the results based on the approximation in Section 6.2. Figure 13 shows the effect of the approximation parameter ϵ on the finishing time of approximation *DRS* algorithm for different m values. As the m value increases, which also implies an increase in the number of offloaded tasks, the average normalized sampling period also increases, because the offloading decision is affected by the rounded-up setup time for the offloaded tasks. For $m \geq 0.5$, the average normalized sampling period is nearly the same because almost all of the tasks are offloaded in this case. Clearly, the ϵ value affects the accuracy of the approximation for *DRS* algorithm. When the value ϵ increases for worse approximation, the finishing time of the tasks also usually, but not always, increases.

8 Conclusion

In this paper, we present two offloading algorithms, *GMF* and *DRS*, for real-time embedded systems. Our algorithms can be used to schedule tasks with and without specified execution order to meet the deadline. Also, they can be used to maximize the sampling rate for tasks execution. Our experimental results show that, even by offloading to server(s) with shorter response time, using *DRS* algorithm can result in significant finishing time reduction. The experiments also

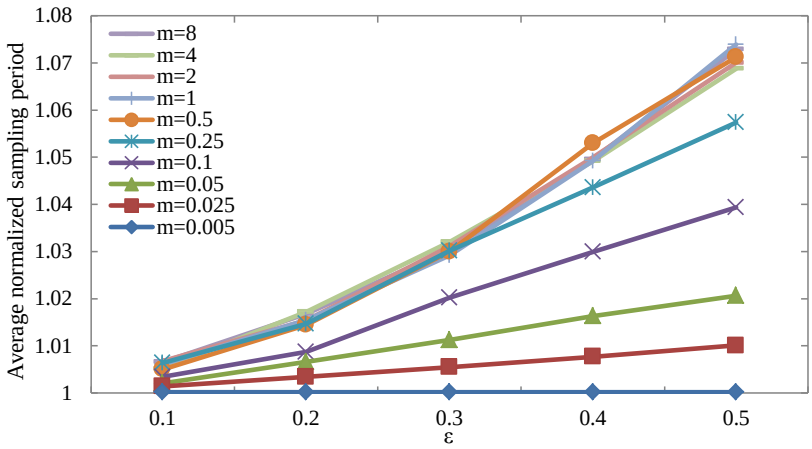


Figure 13 Normalized sampling period for synthesized tasks.

reveal that *DRS* algorithm reduces the finishing time up to 52 % of the total local execution time, and improves the finishing time of other existing offloading algorithms up to 44.7 %.

References

1. Gary R. Bradski and Adrian Kaehler. *Learning OpenCV – computer vision with the OpenCV library: software that sees*. O'Reilly, 2008. URL: `http://www.oreilly.de/catalog/9780596516130/index.html`.
2. Giorgio C. Buttazzo. *Hard Real-time Computing Systems*. Springer US, 2011. URL: `http://www.springer.com/978-1-4614-0675-4`.
3. Luis Lino Ferreira, Guilherme D. Silva, and Luís Miguel Pinho. Service offloading in adaptive real-time systems. In Zoubir Mammeri, editor, *IEEE 16th Conf. on Emerging Technologies & Factory Automation (ETFA'11), Toulouse, France, September 5–9, 2011*, pages 1–6. IEEE, 2011. `doi:10.1109/ETFA.2011.6059236`.
4. M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
5. Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., NJ, USA, 2008. URL: `http://www.imageprocessingplace.com/`.
6. Yu-Ju Hong, Karthik Kumar, and Yung-Hsiang Lu. Energy efficient content-based image retrieval for mobile systems. In *Int'l Symp. on Circuits and Systems (ISCAS'09), 24–17 May 2009, Taipei, Taiwan*, pages 1673–1676. IEEE, 2009. `doi:10.1109/ISCAS.2009.5118095`.
7. IFR International Federation of Robotics. Service Robot Statistics, September 2011. URL: `http://www.ifr.org/service-robots/statistics/`.
8. Dejan Kovachev, Tian Yu, and Ralf Klamma. Adaptive computation offloading from mobile devices into the cloud. In *10th IEEE Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA'12), Leganes, Madrid, Spain, July 10–13, 2012*, pages 784–791. IEEE, 2012. `doi:10.1109/ISPA.2012.115`.
9. Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *2001 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, pages 238–246, 2001. URL: `http://portal.acm.org/citation.cfm?id=502217.502257`.
10. Zhiyuan Li, Cheng Wang, and Rong Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. In *16th Int'l Parallel and Distributed Processing Symp. (IPDPS'02), 15–19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2002. `doi:10.1109/IPDPS.2002.1015589`.
11. David G. Lowe. Object recognition from local scale-invariant features. In *Int'l Conf. on Computer Vision (ICCV'99), Vol. 2*, pages 1150–1157, 1999. URL: `http://dl.acm.org/citation.cfm?id=850924.851523`.
12. Yamini Nimmagadda, Karthik Kumar, Yung-Hsiang Lu, and C. S. George Lee. Real-time moving object recognition and tracking using computation offloading. In *2010 IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS'10), October 18–22, 2010, Taipei, Taiwan*, pages 2449–2455. IEEE, 2010. `doi:10.1109/IROS.2010.5650303`.
13. Massimo Piccardi. Background subtraction techniques: a review. In *2004 IEEE Int'l Conf. on Systems, Man & Cybernetics (ICSMC'04), The Hague, Netherlands, 10–13 October 2004*, pages 3099–3104. IEEE, 2004. `doi:10.1109/ICSMC.2004.1400815`.
14. Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *15th IEEE Real-Time Systems Symp. (RTSS'94), San Juan, Puerto Rico, December 7–*

9, 1994, pages 2–11. IEEE Computer Society, 1994. doi:10.1109/REAL.1994.342735.

15 Marco Spuri and Giorgio C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996. doi:10.1007/BF00360340.

16 Richard Wolski, Selim Gurun, Chandra Krintz, and Daniel Nurmi. Using bandwidth data to make computation offloading decisions. In *22nd IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS'08), Miami, Florida USA, April 14–18, 2008*, pages 1–8. IEEE, 2008. doi:10.1109/IPDPS.2008.4536215.

17 Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Adaptive computation offloading for energy conservation on battery-powered systems. In *13th Int'l Conf. on Parallel and Distributed Systems (ICPADS'07), December 5–7, 2007, Hsinchu, Taiwan*, pages 1–8. IEEE Computer Society, 2007. doi:10.1109/ICPADS.2007.4447724.

Implementing Mixed-criticality Systems Upon a Preemptive Varying-speed Processor

Zhishan Guo and Sanjoy K. Baruah

University of North Carolina, Chapel Hill, NC, USA, {zsguo,baruah}@cs.unc.edu

Abstract

A *mixed criticality (MC)* workload consists of components of varying degrees of importance (or "criticalities"); the more critical components typically need to have their correctness validated to greater levels of assurance than the less critical ones. The problem of executing such a MC workload upon a preemptive processor whose effective speed may vary during run-time, in a manner that is not completely known prior to run-time, is considered.

Such a processor is modeled as being characterized by several execution speeds: a normal speed and several levels of degraded speed. Under normal circumstances it will execute at or above its normal speed; conditions during run-time may cause it to execute slower. It is desired that all components of the MC workload execute correctly under normal circumstances. If the processor speed degrades, it should nevertheless remain the case that the more critical components execute correctly (although the less critical ones need not do so).

In this work, we derive an optimal algorithm for scheduling MC workloads upon such platforms; achieving optimality does not require that the processor be able to monitor its own run-time speed. For the sub-case of the general problem where there are only two criticality levels defined, we additionally provide an implementation that is asymptotically optimal in terms of run-time efficiency.

2012 ACM Subject Classification Real-Time Schedulability
Keywords and phrases Mixed criticalities, varying-speed processor, preemptive uniprocessor scheduling
Digital Object Identifier 10.4230/LITES-v001-i002-a003
Received 2014-04-23 **Accepted** 2014-08-26 **Published** 2014-11-17

Editor Neil Audsley

1 Introduction

As stated in the title, this paper is concerned with the implementation of *mixed-criticality systems* upon *varying-speed processors*. We start out by explaining these terms.

Varying-speed CPUs. Due to cost and related considerations, there is an increasing trend in embedded computing towards implementing safety-critical systems upon commercially available general-purpose processors (commonly known as *commercial off-the-shelf* or *COTS* processors). The special-purpose processors previously used in implementing safety-critical systems were designed to be highly predictable in the sense that tight bounds on the run-time behavior of a system could be *a priori* determined during system design time itself. However, such design-time predictability is difficult to achieve with COTS processors that are typically engineered to provide good average-case performance rather than worst-case guarantees. Such design-time predictability is nevertheless essential for safety-critical functionalities whose correctness must be validated to very high levels of assurance prior to system deployment. In this paper, we focus upon one aspect of guaranteeing real-time performance upon COTS processors despite their inherent unpredictability: *worst-case execution time (WCET)*.

The WCET abstraction plays a central role in the analysis of real-time systems. For a specific piece of code and a particular platform upon which this code is to execute, the WCET of the code denotes (an upper bound on) the amount of time the code takes to execute upon the platform.

Determining the exact WCET of an arbitrary piece of code is provably an undecidable problem. Devising analytical techniques for obtaining tight upper bounds on WCET is currently a very active area of research, and sophisticated tools incorporating the latest results of such research have been developed (see [16] for an excellent survey). WCET tools require that some assumptions be made about the run-time behavior of the processor upon which the code is to execute; for example, the *clock speed* of the processor during run-time must be known in order to be able to determine the rate at which instructions will execute. However, conditions during run-time, such as changes to the ambient temperature, the supply voltage, etc., may result in variations in the clock speed. For instance, a system may be designed to have its CPU clock speed(s) reduced into a certain value temporarily whenever it is detected that core temperature gets higher than a threshold. Such variation is likely to be further exacerbated in the future, with the increasing trend in computer architecture towards *Globally Asynchronous Locally Synchronous*, or GALS, circuit designs. In order to be able to guarantee that the values they compute are correct under all run-time conditions, a WCET tool must make the most pessimistic assumptions regarding clock speed: that during run-time *the clock speed takes on the lowest possible value.* If this lowest possible value is highly unlikely to be reached in practice during actual runs, then a significant under-utilization of the CPU's computing capacity will be observed during run-time.

Mixed-criticality Systems. In safety-critical hard-real-time systems, there is little that can be done about such under-utilization of platform resources. But as stated above, another increasing trend in embedded computing is the move towards mixed-criticality (MC) systems, in which functionalities of different degrees of importance or *criticalities* are implemented upon a common platform. As a consequence the real-time systems research community has recently devoted much attention to better understanding the challenges that arise in implementing such MC systems (see [5] for a review of some of this work). The typical approach has been to validate the correctness of highly critical functionalities under *more pessimistic assumptions* than the assumptions used in validating the correctness of less critical functionalities. For instance, a piece of code may be characterized by a larger WCET in the more pessimistic analysis and a smaller WCET in the "normal" (less pessimistic) analysis [15]. All the functionalities are expected to be demonstrated correct under the normal analysis, whereas the analysis under the more pessimistic assumptions need only demonstrate the correctness of the more critical functionalities.

The results reported in this paper fall within the same framework as this prior work. However, rather than considering variations in estimating WCET, we assume that each piece of code is characterized by a single WCET, and focus instead on the variations in run-time speed of the processing platform. As in earlier work, the mixed-criticality nature of the system that is considered in this paper is reflected in the fact that while we would like all functionalities to execute correctly under normal circumstances, it is essential that the more critical functionalities execute correctly even under pathological conditions which, while extremely unlikely to occur in practice, cannot be entirely ruled out. To express this formally, we model the workload of a MC system as being comprised of a collection of real-time jobs – these jobs may be independent, or they may be generated by recurrent tasks. Each job is characterized by a release date, a (single) WCET, a deadline, and a criticality level $\in \{1, 2, \ldots, m\}$ expressing its degree of importance, with larger values denoting greater importance. We desire to schedule the system upon a single preemptive processor. This processor is a varying-speed one that is characterized by a sequence of m speeds $1 = s_1 > s_2 > \ldots > s_m$. The run-time behavior of this processor is as follows: while under normal circumstances it completes at least one unit of execution during each time unit (equivalently, it executes as a speed-1, or faster, processor), its speed may degrade to lower values during run-time. The precise manner in which the speed will vary during run-time is not *a priori*

known. We seek a scheduling strategy that for all $l, 1 \leq l \leq m$ *guarantees to correctly execute all those jobs that have criticality* $\geq l$*, provided the processor speed never falls below* s_l *during run-time.*

The following example illustrates this model.

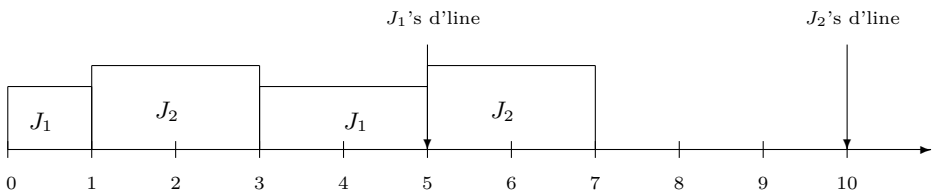
▶ **Example 1.** Consider the following collection of two jobs, to be scheduled on a preemptive processor with specified speeds $s_1 = 1$ and $s_2 = \frac{1}{2}$:

Job	Criticality	Release date	WCET	Deadline
J_1	LO	0	3	5
J_2	HI	1	4	10

An Earliest Deadline First (EDF) [12] schedule for this system prioritizes J_1 over J_2. This is fine if the processor does not degrade: J_1 executes over the interval $[0, 3)$ and J_2 over $[3, 7)$, thereby resulting in both deadlines being met.

Now suppose that the processor were to degrade at some instant within the time-interval $[0, 10]$: a correct scheduling strategy should execute the HI-criticality job J_2 to complete by its deadline (although it may fail to execute J_1 correctly). But consider the scenario where the processor degrades to some speed $s' < \frac{4}{7}$, or ≈ 0.55) starting at time-instant 3: in the EDF schedule J_2 would obtain merely $(10 - 3) \times s' < 4$ units of execution prior to its deadline at time-instant 10. We therefore conclude that EDF does not schedule this system correctly.

An alternative scheduling strategy could instead execute jobs as follows on a normal (speed-1) processor: J_1 over the interval $[0, 1)$; J_2 over $[1, 3)$; J_1 again, over $[3, 5)$; and finally J_2 over $[5, 7)$:



If the processor degrades to a speed < 1 at any instant during this execution then the processor immediately switches to executing J_2 until it completes.

It may be verified that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades to any speed $\geq \frac{1}{2}$, and in both deadlines being met if the processor remains normal (or degrades at any instant ≥ 5).

Contributions and Organization. As mixed-criticality (MC) systems increasingly come to be implemented upon commodity processors, we believe it imperative that real-time scheduling theory provide an understanding of how to implement these systems to meet the twin goals of providing *correctness guarantees at high levels of assurance* to the more critical functionalities while simultaneously making *efficient use of platform resources.* As discussed above, commodity processors tend to execute at varying speeds as ambient conditions change; in order to make correctness guarantees at very high levels of assurance upon such varying-speed processors, it may be necessary to consider the possibility that the processor is executing at a very low speed. In this paper, we seek to define a formal framework for the scheduling-based analysis of MC systems that execute upon CPUs which may be modeled as varying-speed processors. To this end, in Section 2 we describe a very simple model for representing MC systems. In Section 3 we propose, analyze, and evaluate an algorithm for the preemptive uniprocessor scheduling of MC systems that can be represented using this model. In Section 4 we consider the special case where there are only two criticality levels (such MC systems have been called *dual-criticality*

systems in the literature), and provide a more efficient algorithm for this restricted case. In Section 5, we discuss the computational complexity of the problem when preemption is forbidden. In Section 6, recurrent tasks are considered and a scheduling strategy is provided when unbounded preemption is permitted. We conclude in Section 7 by placing this work within the larger context of mixed-criticality scheduling, and briefly enumerate some important and interesting directions for further research.

Relationship to Prior Work. The years since Vestal's seminal paper in 2007 [15] have seen a large amount of research in mixed-criticality scheduling. Much of this research considers a model in which each job is characterized by multiple WCETs. The results from this prior research can be directly applied to our problem, in the following manner. Consider a job in our setting that has WCET c and is being scheduled on a varying-speed processor with normal speed $s_1 = 1$ and degraded speeds $s_2, \ldots, s_m$. This job may be represented in the multiple-WCET model as a job with a normal WCET of c and more pessimistic WCETs of $c/s_2, \ldots, c/s_m$; if all jobs execute for no more than their normal WCETs then all jobs should execute correctly, while if some jobs execute beyond their normal WCETs then only some of the jobs (those with criticality levels exceeding a particular value) are required to execute correctly. It is not difficult to show that the algorithms proposed in prior work for scheduling MC systems with multiple WCET specifications can be used to schedule this transformed system, and that the resulting scheduling strategy correctly schedules our (original) system upon the varying-speed processor. Hence, all the problems considered in this paper could in principle be solved by simply transforming to the earlier, multiple-WCET, model, and applying the previously-proposed solution techniques.

However, in [2] we showed that one can sometimes do better than such an approach. This was observed to be because the problem we are considering here, of MC scheduling on varying-speed processors, is *simpler* (from a computational complexity perspective) than the previously-considered problem of MC scheduling with multiple-WCETs specified. For instance, whereas determining preemptive uniprocessor feasibility for a collection of independent MC jobs specified according to the multiple-WCET model is known [3] to be NP-hard in the strong sense, in Section 3 we will present an optimal polynomial-time algorithm for solving the same problem in our model. For the case of dual-criticality systems of implicit-deadline sporadic tasks on preemptive uniprocessors, a speedup lower bound of 4/3 had been established [4] for the multiple-WCETs model, whereas [2] had provided an optimal (speedup-1) algorithm.

This paper extends our recent work [2] in several significant directions. First (as stated above), the results in [2] were only shown to hold for mixed-criticality systems that are implemented upon varying-speed processors for which just two speeds are specified; this paper extends these results to be applicable to mixed-criticality systems implemented upon varying-speed processors with an arbitrary number of speeds specified. Second, [2] had derived a linear-programming (LP) approach to solving the problem in the two-level case (thereby establishing that the problem could be solved in polynomial time). In this paper, we derive an altogether different algorithm for solving the two-speed case, that has a worst-case run-time of $\mathcal{O}(n \log n)$ where n is the number of jobs in the instance; this is more efficient than the earlier LP-based approach. And finally, the concept of *self-monitoring* by processors was introduced [8] as a means of distinguishing between processors that do or do not "know" at each instant during run-time, what their precise speeds are. While the algorithms derived in [2] assume that the processor possesses the self-monitoring property, the algorithms we derive here do not require this property to hold.

A Note. Although we have chosen to model the problem in terms of real-time jobs executing on varying-speed processors, the model (and our results) are also applicable to the transmission of

time-sensitive data on potentially bandwidth-varying communication media. Specifically, they are particularly relevant to data-communication problems in which time-sensitive data and data-streams must be transmitted over communications media which can provide a high bandwidth under most circumstances but can only *guarantee* some lower bandwidths: the high bandwidth would correspond to the normal processor speed, and the lower bandwidths to the degraded speeds. We therefore believe that this work is relevant to problems of factory communication, communication within automobiles or aircraft, wireless sensor networks, etc., in addition to processor scheduling of mixed-criticality workloads.

2 Model

We start out considering mixed-criticality systems that can be modeled as collections of *independent jobs*; a model for *recurrent tasks* is considered in Section 6. In our model, a mixed-criticality real-time workload consists of basic units of work known as mixed-criticality jobs. Each mixed-criticality (MC) job J_i is characterized by a 4-tuple of parameters: a release date a_i, a WCET c_i, a deadline d_i, and a criticality level $\chi_i \in \{1, 2, \ldots, m\}$. Note that this WCET c_i is measured based upon some constant unit-speed processor – a job with WCET of c_i may require a period of length c_i/s when executing on a speed-s processor.

Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in increasing order (i. e., $t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\left[\min_i\{a_i\}, \max_i\{d_i\}\right)$ into k *intervals*, which we will denote as $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

A mixed-criticality *instance* I is specified by specifying

- a finite collection of MC jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and
- a varying-speed processor that is characterized by a normal speed s_1 (without loss of generality, assumed to be 1) and some specified *degraded processor speeds* $s_2, \ldots, s_m$ in strictly decreasing order; i. e., $s_m < s_{m-1} < \ldots < s_2 < 1$.

The interpretation is that the jobs in $\mathcal{J}$ are to execute on a single shared processor that has m modes: a *normal* mode and $(m-1)$ *degraded* modes. In the normal mode, the processor executes as a unit-speed processor and hence completes one unit of execution per unit time, whereas in degraded mode l it completes fewer than s_{l-1}, but at least s_l, units of execution per unit time, for $l = 2, \ldots, m$.

The processor starts out executing at its normal speed. It is not *a priori* known when, if at all, the processor will degrade: this information only becomes revealed during run-time when the processor actually begins executing at a slower speed. We seek to determine a ***correct scheduling strategy***, which is formally defined as follows:

▶ **Definition 2** (Correct Scheduling Strategy)**.** A scheduling strategy for MC instances is *correct* if it possesses the property that upon scheduling any MC instance $I = (\mathcal{J} = \{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$, each job J_i completes by its deadline if the processor executes at speeds $\geq s_{\chi_i}$ throughout its scheduling window $[a_i, d_i)$.

3 A Scheduling Algorithm

In this section we present efficient strategies for scheduling preemptable mixed-criticality instances. We start out with a general **overview** of our strategy. Given an instance I, prior to run-time we will construct a scheduling table $S(I)$ which prescribes the amounts of execution to be received by each job during each interval. During run-time, scheduling decisions are made according to this scheduling table. Amounts within each interval are executed in the decreasing order of criticality

levels (greater criticality first). A job is dropped at its deadline if it has not completed execution by then. Note that we do not discard a job with criticality level lower than ℓ even when the processing speed is detected to have fallen to some value in the range $(s_{\ell+1}, s_\ell])$ – such a mechanism improves the likelihood of lower criticality jobs meeting their deadlines despite processor degradation[1]

In the remainder of this section we present, and prove the correctness of, a simple linear-programming based algorithm for constructing the scheduling table $S(I)$ optimally. By *optimal*, we mean that if there is a correct scheduling strategy (Definition 2 above) for an instance I, then the scheduling strategy described above is a correct scheduling strategy with the scheduling table we will construct. We start out identifying the following (obvious) necessary condition for MC-schedulability:

▶ **Lemma 3.** *In order that a correct scheduling strategy exists for MC instance $I = (\mathcal{J}, s_1, \ldots, s_m)$, it is* necessary *that for each criticality level $l = 1, \ldots, m$, EDF correctly schedules all the jobs in I with criticality level $\geq l$ upon a speed-s_l uniprocessor.*

Given any instance I, it can be efficiently determined whether I satisfies the necessary conditions of Lemma 3: for each l, simply simulate the EDF scheduling of all the jobs in I with criticality-level $\geq l$ upon a speed-s_l processor. In the remainder of this section, let us therefore assume that any instance under consideration satisfies these necessary conditions. (I.e., any instance that fails these conditions can obviously not have a correct scheduling strategy, and is therefore flagged as being unschedulable.)

Given an MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$ that satisfies the conditions of Lemma 3, we now describe how to construct a linear program (LP) such that a feasible solution for this linear program can be used to construct scheduling table $S(I)$.

To construct our linear program we define $n \times k$ variables $x_{i,j}$, $1 \leq i \leq n; 1 \leq j \leq k$. Variable $x_{i,j}$ denotes the amount of execution we will assign to job J_i in the interval I_j, in the scheduling table that we are seeking to build.

The following n constraints specify that each job receives adequate execution in the normal schedule:

$$\left(\sum_{j | t_j \geq a_i \wedge d_i \geq t_{j+1}} x_{i,j} \right) \geq c_i, \text{ for each } i, 1 \leq i \leq n; \tag{1}$$

while the following k constraints specify the capacity constraints of the intervals:

$$\left(\sum_{i=1}^{n} x_{i,j} \right) \leq s_1(t_{j+1} - t_j), \text{ for each } j, 1 \leq j \leq k. \tag{2}$$

Within each interval, jobs will be executed in the priority order of their criticality levels; i. e., amounts from higher criticality level jobs get executed first. (That is, the interval I_j will have a block of level-m criticality execution of duration $\sum_{i:\chi_i=m}^{n} x_{i,j}$, followed by blocks of l-criticality execution of duration $\sum_{i:\chi_i=l} x_{i,j}$ with l from $m-1$ down to 1, in order.) It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above $(n+k)$ constraints will execute all jobs to completion upon a normal (non-degraded) processor. It now remains to write constraints for specifying the requirements with respect to degraded conditions – that the higher-criticality jobs complete execution even in the event of the processor degrading into corresponding modes.

[1] An example of such benefit will be shown in the execution analysis (Item 2) of Example 4, where J_2 with criticality level of 2 may meet its deadline despite the processor speed falling to below s_2 during $[a_2, d_2)$.

Since within each interval, amounts are executed in decreasing order of criticality level, we observe that the worst-case scenarios occur when the processing speed drops at the very *beginning* of a time interval, since that would leave the minimum computing capacity. For each $\{p, l\}$, $1 \leq p \leq k, 2 \leq l \leq m$, we represent the possibility that the processor degrades into speed-s_l mode at the start of the interval I_p in the following manner:

(i) Suppose that the processor degrades into speed-s_l mode at time-instant t_p; i. e., the start of the interval I_p. Henceforth, only jobs of criticality $\geq l$ must be fully executed in order to meet their deadlines.

(ii) Hence for each $t_q \in \{t_{p+1}, t_{p+2}, \cdots, t_{k+1}\}$, constraints must be introduced to ensure that the cumulative remaining execution requirement of all jobs of criticality $\geq l$ with deadline at or prior to t_q can complete execution by t_q on a speed-s_l processor.

(iii) This is ensured by writing a constraint

$$\left(\sum_{i|(\chi_i \geq l) \wedge (d_i \leq t_q)} \left(\sum_{j=p}^{q-1} x_{i,j} \right) \right) \leq s_l(t_q - t_p). \tag{3}$$

Note that for any job J_i with $d_i \leq t_q$, $\left(\sum_{j=p}^{q-1} x_{i,j}\right)$ represents the remaining execution requirement of job J_i at time-instant t_p. The outer summation on the left-hand side is simply summing this remaining execution requirement over all the jobs of criticality $\geq l$ that have deadlines at or prior to t_q.

(iv) A moment's thought should convince the reader that rather than considering all t_q's in $\{t_{p+1}, t_{p+2}, \cdots, t_{k+1}\}$ as stated in (2) above, it suffices to only consider those that are deadlines for some job of criticality $\geq l$.

(v) The Constraints (3) above only prevent missing deadlines after t_p when the (degraded) processor is continually busy over the interval between t_p and the missed deadline; what about deadline misses when the processor is not continually busy over this interval (and the right-hand side of the inequality of Constraints (3) therefore does not reflect the actual amount of execution received)? We point out that for such a deadline miss to occur, it must be the case that there is a subset of jobs of criticality $\geq l$ – those with release dates and deadlines between the last idle instant prior to the deadline miss and the deadline miss itself – that miss their deadlines on a speed-s_l processor. But this would contradict our assumption that the instance passes the necessary conditions of Lemma 3, i. e., all the jobs of criticality $\geq l$ together (and therefore, every subset of these jobs) execute successfully on a speed-s_l processor.

The entire linear program is listed in Figure 1, and the steps of our LP-based table-driven mixed-criticality scheduling approach, titled Algorithm TDMC-LP, is described in Figure 2.

It is evident that during run-time Algorithm TDMC-LP is performing a typical interval-by-interval execution – unless idleness is detected, no amount of execution that is assigned in later intervals can be "promoted" (executed in an earlier interval).

Note that due to processor degradation, it is possible that some amounts of execution that were assigned to an interval may not have completed by the end of the interval. In such a case, we do not simply drop these execution amounts, but pass them over into the subsequent interval. The reason for this additional modification during run time is that Constraints (3) only provide guarantees as to the total amount of execution provided for each job *until its deadline*. This can be done by adding the unfinished part of the amounts into the corresponding rows in the column of the scheduling table at the end of each interval (as described in Step 2b)[2]. The rationale behind such maintenance during run-time will also be shown in Example 4.

[2] Note that here $Ex(i, j)$ does not denote the total execution *time* of job J_i within Interval I_j – the processing speed during run-time needs to be considered as well.

Given: MC instance $(\{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$, with job release-dates and deadlines partitioning the time-line over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$.
Determine values for the x_{ij} variables, $i = 1, \ldots, n, j = 1, \ldots, k$ satisfying the following **constraints:**

- For each i, $1 \leq i \leq n$,

$$\left(\sum_{j \mid t_j \geq a_i \wedge d_i \geq t_{j+1}} x_{i,j} \right) \geq c_i. \tag{1}$$

- For each j, $1 \leq j \leq k$,

$$\left(\sum_{i=1}^{n} x_{i,j} \right) \leq s_1(t_{j+1} - t_j). \tag{2}$$

- For each p, $1 \leq p \leq k$, for each l, $2 \leq l \leq m$, and for each q, $p < q \leq (k+1)$

$$\left(\sum_{i \mid (\chi_i \geq l) \wedge (d_i \leq t_q)} \left(\sum_{j=p}^{q-1} x_{i,j} \right) \right) \leq s_l(t_q - t_p). \tag{3}$$

Figure 1 Linear program for constructing the scheduling table.

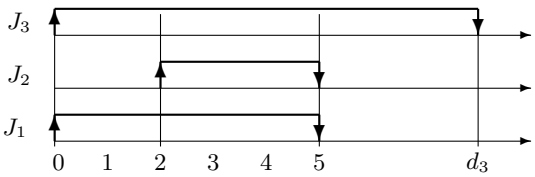
Given: $J = \cup_{i=1}^{n}\{J_i\}$ to be scheduled on a varying-speed processor with speed thresholds $s_1, \ldots, s_m$.

- Construct the scheduling table S according to Figure 1, with $x_{i,j}$ denoting the amount of execution assigned to job J_i during the interval I_j, for each pair (i, j).
- **For** each interval $I_j, j = 1$ up to k:
  1. Higher-criticality execution is performed before lower-criticality ones within each interval, while amounts with the same criticality level may be executed in any order.
  2. At the end of the interval; i. e., at time $t = t_j$
     a. **If** t_j is some unfinished job's deadline, **then** the job is dropped; this is indicated by setting $x_{i,j} \leftarrow -1$ $\forall i$ for which $d_i = t_j$.
     b. Other unfinished executions (if any) need to be carried over into the next interval; i. e., $\forall i$ such that $d_i > t_j$, $x_{i,j+1} \leftarrow x_{i,j+1} + x_{i,j} - Ex(i,j)$, where $Ex(i,j)$ denotes the *amount* of execution that job J_i received within Interval I_j.
  3. Whenever an idleness is detected, we may execute the (released) jobs with amounts assigned to later interval(s) in the same priority order described in Step 1.

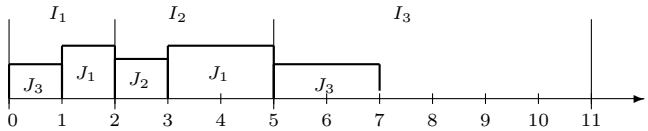
Figure 2 Basic steps of the proposed scheduling algorithm TDMC-LP.

J_i	a_i	c_i	d_i	χ_i
J_1	0	3	5	1
J_2	2	1	5	2
J_3	0	c_3	d_3	3

(a)



(b)



(c)

Figure 3 Illustrating Example 4. The jobs are listed in (a), and depicted graphically in (b). The scheduling table that is constructed is depicted in (c).

(We also point out that the execution order when an idleness is detected, as described in Step 3, represents an optimization in run-time behavior that has nothing to do with correctness – the proof of Theorem 5 will go through even if the processor is left idled until the end of such an interval.)

Before proving its correctness and optimality, we first illustrate the operation of Algorithm TDMC-LP by means of a simple example.

▶ **Example 4.** We will consider a MC instance I consisting of three jobs with parameters as depicted in Figure 3(a), with c_3's value left unspecified for now, and d_3 assumed to be larger than 5.

The release dates and deadlines of these three jobs define three intervals: $I_1 = [0, 3)$; $I_2 = [3, 5)$; $I_3 = [5, d_3)$, as illustrated in Figure 3(b).

Since there are three jobs in I ($n = 3$), Constraints (1) of the LP will be instantiated to the following three inequalities, specifying that all three jobs receive adequate execution in the scheduling table $S(I)$ to execute correctly on a normal (non-degraded) processor:

$$\begin{aligned} x_{11} + x_{12} &\geq 3; \\ x_{22} &\geq 1; \\ x_{31} + x_{32} + x_{33} &\geq c_3. \end{aligned}$$

There are also three intervals I_1, I_2, and I_3. Constraints 2 of the LP will therefore yield the following three inequalities, specifying that the capacity constraints of the intervals are met:

$$\begin{aligned} x_{11} + x_{21} + x_{31} &\leq 2; \\ x_{12} + x_{22} + x_{32} &\leq 3; \\ x_{13} + x_{23} + x_{33} &\leq d_3 - 5. \end{aligned}$$

It remains to instantiate the Constraints (3), that were introduced to ensure correct behavior in the event of processor degradation. In this example there are three criticality levels, and thus a need to consider degradation cases of both speed-s_2 and speed-s_3. These must be separately instantiated to model the possibility of the processor degrading at the start of each of the three intervals I_1, I_2 and I_3. We consider these separately:

- **Degradation at the start of I_1.** In this case, Constraints (3) is instantiated three times: speed-s_2 for $t_m = 5$, and both speed-s_2 and speed-s_3 for $t_m = d_3$:
$$\begin{aligned} x_{21} + x_{22} &\leq (5-0)\, s_2; \\ \big(x_{21} + x_{22} + x_{23}\big) + \big(x_{31} + x_{32} + x_{33}\big) &\leq (d_3 - 0)\, s_2; \\ x_{31} + x_{32} + x_{33} &\leq (d_3 - 0)\, s_3. \end{aligned}$$

- **Degradation at the start of I_2.** This case is similar as the above one that Constraints (3) is instantiated once for $t_m = 5$ and twice for $t_m = d_3$:
$$\begin{aligned} x_{22} &\leq (5-2)\, s_2; \\ \big(x_{22} + x_{23}\big) + \big(x_{32} + x_{33}\big) &\leq (d_3 - 2)\, s_2; \\ x_{32} + x_{33} &\leq (d_3 - 2)\, s_3. \end{aligned}$$

- **Degradation at the start of I_3.** In this case, Constraints (3) is instantiated twice, for $t_m = d_3$ with speeds s_2 and s_3:
$$\begin{aligned} x_{33} &\leq (d_3 - 5)\, s_2; \\ x_{33} &\leq (d_3 - 5)\, s_3. \end{aligned}$$

  (Note that there are nine variables and fourteen constraints in this particular example.)

Continuing with this example, suppose that c_3 and d_3 are 3 and 11 respectively, with degraded speeds $s_2 = 1/2$ and $s_3 = 1/3$. A possible solution to the LP would assign the x_{ij} variables the following values:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \end{bmatrix}.$$

As a consequence, the scheduling table would be as depicted in Figure 3(c).

We can see that this scheduling table yields a correct scheduling strategy: observe that there are three contiguous blocks of execution of criticality-level 2 or greater: $[0, 1)$, $[2, 3)$, and $[5, 7)$, and consider the possibility of the processor degrading during each:

- If the processor degrades to speed-s_2 during $[0, 2)$, then J_3 will execute over $[0, 2)$ and $[5, 9)$, while J_2 can execute over $[2, 4)$. Both jobs of criticality ≥ 2 would thus meet their deadlines on the speed-1/2 processor. J_1 is executed over $[4, 5)$ and dropped at $t = 5$.
- If the processor degrades to speed-s_3 during $[0, 2)$, then for the first interval $[0, 2)$, J_3 will be executed. However the assigned amount $x_{31} = 1$ may not be finished in case the processor degrades early, say at $t = 0$. As a result, the scheduling table needs to be updated at time $t = 2$ according to Step 2b in Figure 2: $x_{32} \leftarrow (0 + 1 - 2/3)$, or $1/3$. J_3 will therefore get to execute over $[2, 3)$ and $[5, 11)$, and meet its deadline, on the speed-1/3 processor. Time interval $[3, 5)$ will be used to execute J_2, and both J_1 and J_2 will be dropped at time $t = 5$ in the worst case, leaving x_{12} and x_{22} the value of -1 for reference. In case the processor degrades to speed-s_3 late, say at $t = 0.5$ (while remaining at unit-speed beforehand), the assigned amount $x_{31} = 1$ can be finished upon $t = 2$, and thus although under a slowest speed condition, J_2 may finish on time be executing over $[2, 5)$.

- If the processor degrades to a speed of either s_2 or s_3 during $[2, 5)$, then J_2 would execute prior to J_1 within this interval and gets finished on time. Job J_3 will not continue its execution until $t = 5$ since $x_{32} = 0$ – it only needs two additional units of execution which will be obtained by executing over the third interval $[5, 9)$.
- If the processor degrades to speed-s_2 (or s_3) during $[5, 7)$, J_3 will still meet its deadline since it has completed one unit of execution prior to the processor degradation – it needs two more units, which will be obtained by executing over $[5, 9)$ (or $[5, 11)$) on the speed-1/2 (or 1/3) processor.

We thus see that the solution of the LP does indeed yield a feasible scheduling strategy according to the proposed run time strategies in TDMC-LP. ◀

Observe that Algorithm TDMC-LP is performing "best-effort execution" – it only discards a job if it has not completed by its deadline, and not merely because a processor degradation is detected. We now formally show that it is guaranteed that the assigned execution amounts with criticality level no lower than ℓ will nevertheless get executed so long as processing speed remains at least as large as s_ℓ (as required under the correctness definition).

▶ **Theorem 5.** *Algorithm* TDMC-LP *is correct.*

Proof. The proof is by contradiction. Assume that some job J_i with criticality level χ_i has not completed by its deadline $d_i = t_q$ (at the end of Interval I_{q-1}), while the processor remains at (or above) a speed of s_{χ_i} over the interval $[a_i, d_i)$.

From constraints (3), we know that total assigned amounts of execution with criticality level no lower than χ_i for intervals that lie within $[a_i, d_i)$ cannot exceed $s_{\chi_i} \times (d_i - a_i)$. Given the fact that no amount with lower criticality level(s) can be executed within the interval $[a_i, d_i)$ (since else J_i would have been assigned and executed during the execution of lower criticality amounts), there must be some "carry-in" amounts of execution with criticality level no lower than χ_i due to Step 2b. Let t_p denote the end of the last interval (before a_i) with either idleness or some execution of amounts with criticality level lower than χ_i (so that no amount assigned before t_p with criticality level $\geq \chi_i$ can be "carried-in"). It is now evident that Constraints (3) must be violated for Interval $[t_p, t_q)$ under speed s_{χ_i}. ◀

▶ **Theorem 6.** *Algorithm* TDMC-LP *is* optimal – *whenever it fails to maintain correctness, no other algorithm can.*

Proof. From Theorem 5, Algorithm TDMC-LP fails only when there is no feasible solution to the LP described in Figure 1. Since the three set of constraints are all necessary ones according to Lemma 3, violations of any of them indicates that the given instance is *not schedulable* under some circumstances (e. g., speed performances during run-time). Thus no other algorithm can maintain correctness as well. ◀

Bounding the Size of This LP. It is not difficult to show that the LP of Figure 1 is of size polynomial in the number of jobs n in MC instance I as well as the number of criticality levels m:

- The number of intervals k is at most $2n - 1$. Hence the number of $x_{i,j}$ variables is $O(n^2)$.
- There are n constraints of the form (1), and k constraints of the form (2). The number of constraints of the form (3) can be bounded from above by (nkm), since for each $p \in \{1, \ldots, k\}$, there can be no more than n t_q's corresponding to deadlines of jobs. Since $k \leq (2n - 1)$, it follows that the number of constraints is $O(n) + O(n) + O(n^2m)$, which is $O(n^2m)$.

Since it is known [10, 9] that a linear program can be solved in time polynomial in its representation, it follows that our algorithm for generating the scheduling tables for a given MC instance I takes time polynomial in the representation of I.

4 The Two-criticality-level Case

In this section, we revisit the same restricted version of the problem that was addressed in [2], and derive a more efficient algorithm for solving it. That is, we consider *dual*-criticality systems executing on a variable-speed processor characterized by just two speeds: a normal speed (assumed as 1) and a degraded speed (designated as s, with $s < 1$). We use the standard designations of LO and HI to denote the lower and higher criticality levels respectively. We propose an alternative method to the Linear Programming approach presented in [2] (and extended for > 2 levels in Section 3 above) for constructing the scheduling table, and show that this new method is computationally very efficient.

At a high level, our algorithm is organized in a manner similar to the one described in Section 3: Given a dual-criticality MC instance I, we will first construct a *scheduling table* $S(I)$, and then make run-time job-dispatch decisions in a manner that is compliant with this scheduling table.

To construct the scheduling table, we first identify (Step 1 below) the latest time intervals during which the HI-criticality jobs must execute if they are to complete execution on a degraded processor; having identified these intervals, we construct (in Step 2) an EDF schedule for the HI-criticality jobs in these intervals.

Step 1. *Considering only the* HI*-criticality jobs in the instance, determine the intervals during which the jobs would execute upon a speed-s processor, if*

1. *each job executes for its* HI*-criticality WCET,*
2. *execution occurs as late as possible.*

It is evident that these intervals may be determined by considering the jobs in non-increasing order of their deadlines (i. e., latest deadline first), and taking the cumulative execution requirements of these jobs. These intervals may therefore be determined in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time (which comes from the time complexity of EDF), where n_{HI} denotes the number of HI-criticality jobs.

Step 2. *Construct an EDF schedule for the* HI*-criticality jobs upon a preemptive processor that has speed s during the intervals determined in Step 1 above, and speed zero elsewhere.*

It follows from the optimality property[3] of EDF that if this step fails to ensure that each HI-criticality job receives adequate execution prior to its deadline, then no scheduling algorithm can guarantee correctness (see Definition 2) for this instance. We would therefore ***report failure***: this MC instance is not feasible. The remainder of this section assumes that Step 2 above was successful in completing each HI-criticality job prior to its deadline.

We now describe how to use this EDF schedule to construct the scheduling table – recall that this scheduling table is used for job dispatch decisions upon both the normal and degraded processor, and is therefore constructed assuming a normal-speed (i. e., speed-1) processor.

Step 3. *To construct the scheduling table, partition the time-line over* $[\min_i\{a_i\}, \max_i\{d_i\}]$ *into the k intervals* $I_1, I_2, \ldots, I_k$*. (Recall, from Section 2, that these are the intervals defined by the release dates and deadlines of all the jobs –* LO*-criticality and* HI*-criticality.)*

3.1 *For each* HI*-criticality job* J_i *and each interval* I_ℓ *in which it is scheduled in the EDF schedule constructed in Step 2 above, execute* J_i *within this interval for an amount* $x_{i\ell}$ *which equals*

[3] Although the optimality proof of EDF in [12], which is based on a swapping argument, assumes that the processor speed remains constant, it is trivial to extend the proof to apply to processors that are only available during limited intervals, or indeed to arbitrary varying-speed processors.

J_i	a_i	c_i	d_i	χ_i
J_1	1	2	10	HI
J_2	5	1	8	HI
J_3	6	2	15	HI
J_4	0	4	6	LO
J_5	1	2	10	LO
J_6	10	3	13	LO

Figure 4 All jobs considered in Example 7, where a_i, c_i, and d_i stands for release date, WCET, and deadline respectively.

to the amount of execution that J_i is allocated during Interval I_ℓ in the EDF scheduled constructed in Step 2 above.

3.2 *Assign* LO*-criticality jobs by simulating the EDF-scheduling of the* LO*-criticality jobs in the remaining capacity of the scheduling table – i. e., in the durations that are not already allocated to the* HI*-criticality jobs during Step 3.1 above.*

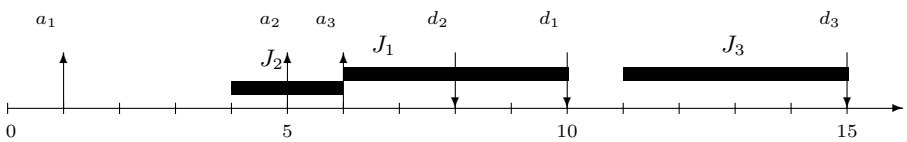
3.3 *If during this EDF simulation there is any capacity left over within an interval (because the supply of currently-active* LO*-criticality jobs has been exhausted), then move over* HI*-criticality jobs, that had been assigned to later intervals in the scheduling table during Step 3.1 above, into the current interval. In so doing favor earlier-deadline jobs over later-deadline ones.*

Note that Step 3.3 is not necessary for correctness; rather, it is an optimization.

We illustrate this table construction process by means of the following example.

▶ **Example 7.** Consider the instance consisting of the six jobs J_1–J_6 shown in tabular form in Figure 4, to be implemented upon a processor of minimum degraded speed $s = 1/2$.

In **Step 1**, we determine the intervals upon which the HI-criticality jobs J_1–J_3 would need to execute if they were to complete as late as possible, upon a degraded processor (one of speed-1/2); this is represented in the following diagram:



In **Step 2**, we construct an EDF schedule of thel HI-criticality jobs J_1–J_3 upon a speed-1/2 processor. Lettting $x_{i,j}$ denote the amount of execution accorded to job J_i in interval I_j, the scheduling table $S(I)$ looks like this:

I_j	$I_1 = [0, 1)$	$I_2 = [1, 5)$	$I_3 = [5, 6)$	$I_4 = [6, 8)$	$I_5 = [8, 10)$	$I_6 = [10, 13)$	$I_7 = [13, 15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	0	1	1

In **Step 3**, we now try to fill in this scheduling table with LO-criticality jobs, interval by interval.

- Interval I_1 will be filled with the job J_4.
- Both J_4 and J_5 are in Interval I_2; J_4 has the earlier deadline. As a result, J_4 receives 3 time units and J_5 takes the remaining 0.5 unit. Here we check that J_4 has received enough execution and meets its deadline.

- Interval I_3 has 0.5 units of execution remaining for job J_5.
- The remaining one time unit capacity in I_4 will be used by J_5. Until now the scheduling table for HI-criticality jobs has remained unchanged from the one constructed in Step 2 (and shown in the above table).
- For the Interval I_5, there is no active LO-criticality job, and the pre-allocated HI-criticality amount $x_{1,5} = 1$ can not fill this up. In this case, we try to move later-assigned HI-criticality amounts into this interval. Specifically, we consider the next interval I_6, where x_{36} should be "promoted" as x_{35}; i. e., the one time unit that originally belongs to Interval $[10, 13)$ will be executed now. Note that after this step, the scheduling table for HI-criticality jobs is changed into the following one (with bold numbers highlighting changes).
- Interval I_6 is now empty and can be fully assigned to job J_6. Here we check that J_6 has received enough execution and meets its deadline.
- Nothing happens to Interval $[13, 15)$.

At the end of Step 3, the scheduling table for all jobs looks like this:

I_j	$[0,1)$	$[1,5)$	$[5,6)$	$[6,8)$	$[8,10)$	$[10,13)$	$[13,15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	***1***	***0***	1
J_4	1	3	0	0	0	0	0
J_5	0	0.5	0.5	1	0	0	0
J_6	0	0	0	0	0	3	0

◀

Computational Complexity. Although an individual job in an EDF schedule for an instance of n jobs may be preempted as many as $(n-1)$ times, it is known (see, e. g., [6]) that the *total* number of preemptions in any EDF schedule for an n-job instance cannot exceed $(n-1)$. In each column of the scheduling table, there should be at least one non-zero element unless all released jobs are finished beforehand. Each more non-zero element denotes that either a job is preempted, or a job finishes its execution within the corresponding interval. Since the number total finishing points is fixed as $n_{\text{HI}} + n_{\text{LO}}$, the total preemption number cannot exceed $(n_{\text{HI}} + n_{\text{LO}} - 1)$, and number of total intervals is no greater than $(2n_{\text{HI}} + 2n_{\text{LO}})$, we know that the total number of non-zero entries in the table of Step 3 cannot exceed $(4n_{\text{HI}} + 4n_{\text{LO}} - 1)$, where n_{HI} (n_{LO}, respectively) denotes the number of HI-criticality (LO-criticality, resp.) jobs in the instance.

We note that standard techniques (see, e. g., [14]) for implementing EDF are known, that allow an EDF schedule for n jobs to be constructed in $\mathcal{O}(n \log n)$ time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time, and the total scheduler overhead during run-time is also bounded from above by $\mathcal{O}(n \log n)$ where $n = n_{\text{HI}} + n_{\text{LO}}$ denotes the total number of jobs.

5 Non-preemptive Scheduling

Recall that the scheduling strategy we adopted in Section 3 above is as follows. Given an instance I, we construct a scheduling table $S(I)$. During run-time scheduling decisions are initially made according to this table. If at any instant it is detected that the processor has transited to degraded mode, the scheduling strategy is *immediately* switched: henceforth, only HI-criticality jobs are executed, and these are executed according to EDF. Such a scheduling strategy requires that the job that is executing at the instant of transition can be preempted, and hence is not applicable for

non-preemptive systems. In this section, we consider the problem of scheduling non-preemptive mixed-criticality instances.

Non-preemptivity mandates that each job receive its execution during one contiguous interval of time. Let us suppose that a LO-criticality job is executing when the processor experiences a degradation in speed. We can specify two different kinds of non-preemptivity requirements:

1. This LO-criticality job does not need to complete – it may immediately be dropped.
2. This LO-criticality job cannot be preempted and discarded – it must complete execution despite that fact that the processor has degraded and this job's completion is not required for correctness.

Although the first requirement – that the LO-criticality job may be dropped – may at first glance seem to be the more reasonable one, implementation considerations may favor the second requirement. For instance, it is possible that the LO-criticality job had been accessing some shared resource within a critical section, and preempting and discarding it would leave the shared resource in an unsafe state.

It has long been known [11] that the problem of scheduling a given collection of independent jobs on a single non-preemptive processor (that does not have a degraded mode) is already NP-hard in the strong sense [11][4]. Since our mixed-criticality problem, under either interpretation of the non-preemptivity requirements, is easily seen to be a generalization, it is also NP-hard. In fact, although determining whether an instance of (regular, not MC) jobs that all share a common release time can be non-preemptively scheduled on a fixed-speed processor is easily solved in polynomial time by EDF, it turns out that even this restricted problem is NP-hard for MC scheduling.

▶ **Theorem 8.** *It is NP-hard to determine whether there is a correct scheduling strategy for scheduling non-preemptive mixed-criticality instances in which all jobs share a common release date.*

Proof Sketch. We prove this first for the second interpretation of non-preemptivity requirements (LO-criticality jobs that have begun execution must be executed to completion), and indicate how to modify the proof for the first interpretation.

This proof consists of a reduction of the partitioning problem [7], which is known to be NP-complete, to the problem of determining whether a given non-preemptive mixed-criticality instance I can be scheduled correctly. The partitioning problem is defined as follows. *Given* a set S of n positive integers $y_1, y_2, \ldots, y_n$ summing to $2B$, *determine* whether there is a subset of S with elements summing to exactly B.

Given an instance S of the partitioning problem, we construct an instance of the mixed-criticality scheduling problem I comprised of $(n+1)$ jobs $J_1, J_2, \ldots, J_{n+1}$. The parameters of the jobs are

$$J_i = \begin{cases} (0, y_i, 5B, \text{HI}), & 1 \le i \le n; \\ (0, B, 2B, \text{LO}), & i = n+1. \end{cases}$$

The normal processor speed is one; the degraded processor speed s is assigned a value equal to half: $s \leftarrow 1/2$.

We will show that there is a partitioning for instance S if and only if there is a correct scheduling strategy for I.

[4] Indeed, it seems that it is difficult to even obtain *approximate* solutions to this problem, to our knowledge, the best polynomial-time algorithm known [1] requires a processor speedup by a factor of 12.

There is a Partitioning for *S*. Let $S' \subseteq S$ denote the subset summing to exactly B. We construct our scheduling table as follows. Jobs corresponding to the elements in S' are scheduled over the interval $[0, B)$, after which J_{n+1} is scheduled over $[B, 2B)$, followed by the scheduling of the jobs corresponding to the elements in $(S \setminus S')$ over $[2B, 3B)$.

- If the processor enters degraded mode prior to time-instant B, then only the HI-criticality jobs need to complete execution; it may be verified that they will do so by their common deadline.
- If the processor enters degraded mode over $[B, 2B)$, then J_{n+1} may execute for no more than the interval $[B, 3B)$. That still leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, on the speed-0.5 processor.
- Otherwise, J_{n+1} completes by time-instant $2B$. That leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, regardless of whether the processor enters degraded mode or not.

There is *No* Partitioning for *S*. In this case, consider the time-instant t_o at which the LO-criticality job J_{n+1} begins execution. We consider three possibilities:

- If $t_o > B$, the processor remains in normal mode but J_{n+1} misses its deadline at time-instant $2B$.
- If $t_o = B$, then the processor must have been idled for some time during $[0, B)$. If the processor were to now enter degraded mode at this time-instant t_o, job J_{n+1} will execute over $[B, 3B)$, after which the strictly more than B units of remaining HI-criticality execution would execute – this cannot complete by the deadline of $5B$ on the speed-1/2 processor.
- Now suppose that that $t_o < B$, and the processor enters degraded mode at this time-instant t_o. It must be the case that $\leq t_o$ units of execution of the HI-criticality jobs has occurred prior to time-instant t_o. Job J_{n+1} will execute over $[t_o, t_o + 2B)$, after which the at least $(2B - t_o)$ remaining units of HI-criticality work must complete. But on the speed-1/2 processor this would not happen prior to the time-instant

$$\begin{aligned} &\geq t_o + 2B + 2(2B - t_o) \\ &= 6B - t_o \\ &> 5B, \end{aligned}$$

  which means that some HI-criticality job misses its deadline.

We have thus shown that there is a correct scheduling strategy for the non-preemptive mixed-criticality instance I if and only if S can be partitioned into two equal subsets.

The proof above assumed the second interpretation of non-preemptivity requirements, in which LO-criticality jobs that begin execution need to complete even if the processor degrades. For the first interpretation of non-preemptivity requirements (LO-criticality jobs that begin execution do not need to complete if the processor degrades while they are executing), we would modify the proof by assigning the jobs $J_1, J_2, \ldots, J_n$ a deadline of $4B$ (rather than $5B$ as above). It may be verified that this modified MC instance can be scheduled correctly if and only if the S can be partitioned into two equal subsets. ◀

The intractability result of Theorem 8 above implies that in contrast to the preemptive case, we are unlikely to be able to obtain efficient (polynomial-time) optimal scheduling strategies for non-preemptive MC scheduling. We are currently working on devising, and evaluating, polynomial-time approximation algorithms for the non-preemptive scheduling of mixed-criticality systems.

6 Recurrent Tasks

In Sections 3-5 above, we have considered mixed-criticality (MC) systems that can be modeled as finite collections of jobs. However, many real-time systems are better modeled as collections of *recurrent processes* that are specified using, e. g., the sporadic tasks model [12, 13]. In this section, we briefly consider this more difficult problem of scheduling mixed-criticality systems modeled as collections of sporadic tasks. As with traditional (i. e., non MC) real-time systems, we will model a MC real-time system τ as being comprised of a finite specified collection of MC recurrent tasks, each of which will generate a potentially infinite sequence of MC jobs. We restrict our attention here to *dual-criticality* systems of *implicit-deadline MC sporadic tasks*. Each task is characterized by a 3-tuple of parameters: $\tau_i = (C_i, T_i, \chi_i)$, with the following interpretation. Task τ_i generates a potentially infinite sequence of jobs, with successive jobs being released at least T_i time units apart. Each such job has a criticality χ_i, a WCET C_i, and a deadline that is T_i time units after its release. The quantity $U_i = C_i/T_i$ is referred to as the *utilization* of τ_i. An *implicit-deadline MC sporadic task system* is specified by specifying a finite number $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of such sporadic tasks, and the degraded processor speed $s < 1$ (as with MC instances of independent jobs, it is assumed that the normal processor speed is one). Such a MC sporadic task system can potentially generate infinitely many different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each sporadic task.

If *unbounded preemption* is permitted, then the scheduling problem for implicit-deadline MC sporadic task systems on uniprocessors is easily and efficiently solved in an optimal manner. We first derive (Theorem 9) a necessary condition for the existence of a correct scheduling strategy. We then present a scheduling strategy, *Algorithm preemptive-MC*, and prove (Theorem 10) that it is optimal.

▶ **Theorem 9.** *A necessary condition for MC sporadic task system (τ, s) to be schedulable by a non-clarivoyant correct scheduling strategy is that*

1. *the sum of the utilizations of all the tasks in τ is no larger than 1, and*
2. *the sum of the utilizations of the* HI*-criticality tasks in τ is no larger than s.*

Proof. It is evident that the first condition is necessary in order that all jobs of all tasks in τ complete execution by their deadlines upon a normal processor, and that the second condition is necessary in order that all jobs of all the HI-criticality tasks in τ complete execution by their deadlines upon a degraded (speed-s) processor. ◀

In order to derive a correct scheduling strategy, we first observe that using preemption we can mimic a *processor-sharing* scheduling strategy, in which several jobs are simultaneously assigned fractional amounts of execution with the constraint that the sum of the fractional allocations should not exceed the capacity of the processor. (This is done by partitioning the time-line into intervals of length Δ where Δ is an arbitrarily small positive number, and using preemption within each such interval to ensure that each job that is assigned a fraction f of the processor capacity gets executed for a duration $f \times \Delta$ within this interval.)

Consider now the following processor-sharing scheduling strategy:

Algorithm Preemptive-MC

1. Initially (i. e., on the normal – non-degradation – processor), assign a share U_i of the processor to each task τ_i during each instant that is active.[5]

[5] A task is defined to be *active* at a time-instant t if it has released a job prior to t and this job has not yet completed execution by time t.

2. If the processor transits to degraded mode at any instant during run-time, immediately discard all LO-criticality tasks and execute the HI-criticality tasks according to EDF.

▶ **Theorem 10.** *Algorithm preemptive-MC is an optimal correct scheduling strategy for the preemptive uniprocessor scheduling of MC sporadic task systems.*

Proof. Let τ denote a MC implicit-deadline sporadic task system satisfying the necessary conditions for schedulability that have been identified in Theorem 9.

It is evident that Algorithm preemptive-MC meets all deadlines if the processor operates at its normal speed, since the processor-sharing schedule ensures that each job of each task τ_i receives exactly C_i units of execution between its release date and its deadline.

Suppose that the processor degrades at some time-instant t_o. If we were to immediately discard all LO-criticality tasks, the second necessary schedulability condition of Theorem 9 ensures that there is sufficient computing capacity on the degraded processor to continue a processor-sharing schedule in which each HI-criticality task τ_i with an active job receives a share U_i of the processor. The correctness of Algorithm preemptive-MC now follows from the existence of this processor-sharing schedule, and the optimality property of preemptive uniprocessor EDF. ◀

If preemption is forbidden, then scheduling of MC sporadic task systems becomes a lot more challenging. As with the collections of independent jobs (Theorem 8), this problem, too, can be shown to be highly intractable.

7 Context and Conclusions

Advanced processors may need to be modeled as *varying-speed* ones: although they are likely to execute at unit speed (or faster) during run-time, we can only *guarantee* that they will execute at lower speeds – the greater the level of assurance at which such a guarantee is sought, the lower the speed that can be guaranteed. Upon such a processor, the scheduling objective is to ensure that all jobs complete in a timely manner if the processor executes at its normal speed, while simultaneously ensuring that more critical jobs complete in a timely manner even if the processor speed falls to below this normal value.

In this paper, we have presented a formal framework for the scheduling-based analysis of MC systems that execute upon CPUs which may be modeled as varying-speed processors. We have defined a very simple model for representing MC systems, and have derived, and proved the correctness of, an optimal algorithm for the preemptive uniprocessor scheduling of MC systems that can be represented using our model. For the special case where there are only two criticality levels (such MC systems have been called *dual-criticality* systems in the literature), we have provided a more efficient scheduling algorithm. We have also cataloged the computational complexity of the problem when preemption is forbidden, and have derived a scheduling strategy for scheduling recurrent mixed-criticality task systems when unbounded preemption is permitted.

Acknowledgements. The research reported here was supported in part by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1409175; and ARO grant W911NF-09-1-0535.

References

1 Nikhil Bansal, Ho-Leung Chan, Rohit Khandekar, Kirk Pruhs, Clifford Stein, and Baruch Schieber. Non-preemptive min-sum scheduling with resource augmentation. In *48th Annual IEEE Symp. on Foundations of Computer Science (FOCS'07), October 20–23, 2007, Providence, RI, USA*, pages 614–624. IEEE Computer Society, 2007. doi:10.1109/FOCS.2007.46.

2 Sanjoy Baruah and Zhishan Guo. Mixed-criticality scheduling upon varying-speed processors. In

IEEE 34th Real-Time Systems Symp. (RTSS'13), Vancouver, BC, Canada, December 3–6, 2013, pages 68–77. IEEE, 2013. doi:10.1109/RTSS.2013.15.

3 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Computers*, 61(8):1140–1152, 2012. doi:10.1109/TC.2011.142.

4 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conf. on Real-Time Systems (ECRTS'12), Pisa, Italy, July 11–13, 2012*, pages 145–154. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.42.

5 Alan Burns and Robert Davis. Mixed-criticality systems: A review. Unpublished manuscript, 4th edition, July 31, 2014. URL: http://www-users.cs.york.ac.uk/~burns/review.pdf.

6 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer US, 2nd edition, 2005. doi:10.1007/978-1-4614-0676-1.

7 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

8 Zhishan Guo and Sanjoy Baruah. Mixed-criticality scheduling upon non-monitored varying-speed processors. In *8th IEEE Int'l Symp. on Industrial Embedded Systems (SIES'13), Porto, Portugal, June 19–21, 2013*, pages 161–167. IEEE, 2013. doi:10.1109/SIES.2013.6601488.

9 Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984. doi:10.1007/BF02579150.

10 L. G. Khachiyan. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096, 1979.

11 Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. doi:10.1016/S0167-5060(08)70743-X.

12 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

13 Aloysius Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-297.pdf.

14 Aloysius Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D. C., May 1988.

15 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE Real-Time Systems Symp. (RTSS'07), December 3–6, 2007, Tucson, Arizona, USA*, pages 239–243. IEEE Computer Society, 2007. doi:10.1109/RTSS.2007.35.

16 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi:10.1145/1347375.1347389.