An Optimal Resource Sharing Protocol for Generalized Multiframe Tasks

Pontus Ekberg[a,*], Nan Guan[a,b], Martin Stigge[a], Wang Yi[a]

[a] *Uppsala University, Department of Information Technology, Box 337, SE-751 05 Uppsala, Sweden*
[b] *Northeastern University, Institute of Embedded Systems, Box 135, Shenyang, China*

Abstract

Many different task models of varying expressiveness have been proposed to capture the timing constraints of real-time workloads. However, techniques for optimal scheduling and exact feasibility analysis of systems with mutually-exclusive shared resources have been available only for relatively simple task models, such as the sporadic task model. We consider sharing of mutually-exclusive resources in task models that are more expressive in terms of the job-release patterns that can be modeled. We propose a new scheduling algorithm and show that it is optimal for scheduling generalized multiframe tasks with shared resources on uniprocessor systems. We also present an efficient feasibility test for such systems, and show that it is both sufficient and necessary.

Keywords: real-time, scheduling, resource sharing

1. Introduction

Processes in real-time systems often compete for shared resources, such as peripheral devices or global data structures that must be accessed in a mutually exclusive manner. To avoid deadlocks and low processor utilization, we need scheduling algorithms that handle the resource sharing.

Well-established solutions to the resource sharing problem exist in the context of *sporadic* [1] task sets. However, these existing solutions are generally either inapplicable or suboptimal for more flexible task models. One generalization of the sporadic task model is provided by the *generalized multiframe* (GMF) task model [2]. With GMF we can model tasks that cycle through a set of different behaviors, and can more precisely represent many systems. Baruah et al. [2] have shown how to efficiently decide feasibility for this model when shared resources are not considered.

*Corresponding author
Email addresses: pontus.ekberg@it.uu.se (Pontus Ekberg), guannan@ise.neu.edu.cn (Nan Guan), martin.stigge@it.uu.se (Martin Stigge), yi@it.uu.se (Wang Yi)

This is the authors' version of DOI: 10.1016/j.jlamp.2014.10.001

The flexible structure of GMF tasks, in combination with shared resources, is the source of difficulty in finding an optimal scheduling strategy. To make optimal scheduling decisions at run-time, we must be aware of the tasks' structures and which behaviors they may display in the near future.

The goal of this work is to show how to analyze and schedule GMF task sets with shared resources. We introduce an efficient technique, which takes the tasks' structures into account, to predict possible resource contention at run-time and thereby determine the urgency of unlocking currently used resources. Based on this technique we propose a new scheduling algorithm and show that it is well suited for scheduling such workloads. The main contributions include:

- We propose the *resource deadline protocol* (RDP) for handling shared resources, and combine it with *earliest deadline first* (EDF) to form the EDF+RDP scheduling algorithm. We prove that EDF+RDP has the following properties:
  - It is optimal for scheduling GMF task sets with shared resources, in the sense that it successfully schedules all feasible task sets.
  - It is deadlock-free, and it enables efficient implementations because there is at most one preemption per job release and all jobs in the system can share a common run-time stack.
- We derive a sufficient and necessary feasibility test for GMF task sets with shared resources. This test is in the same complexity class as the known feasibility test for GMF task sets without resources [2], i.e., pseudo-polynomial for bounded-utilization task sets.

2. Preliminaries

The results of this paper are presented in the context of the GMF task model [2], which we describe in Section 2.1. We extend GMF with the ability to model shared resources in Section 2.2.

2.1. The Generalized Multiframe Task Model

The GMF task model is a generalization of the well-known *sporadic* [1] and *multiframe* [3] task models. Like a sporadic task, a GMF task releases a sequence of *jobs*. However, the jobs released by a GMF task do not all need to have the same parameters (e.g., execution time and deadline). Instead, a GMF task cycles through a sequence of *job types*, which specify the parameters of the jobs that are released.

A natural way of representing a GMF task is to use a directed cycle graph, where the vertices represent the job types, and the arcs specify the order in which jobs are released (as well as the minimum delay between consecutive job

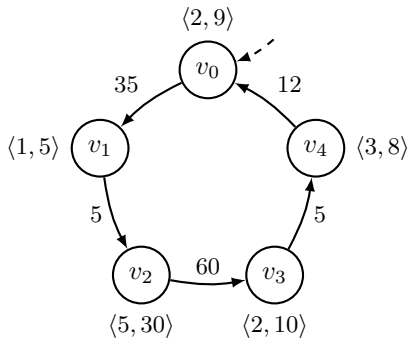

Figure 1: An example GMF task containing five job types.

releases). An example GMF task is depicted in Figure 1. Formally, a GMF task set τ is defined as follows:[1]

- Each task $T \in \tau$ is a directed cycle graph, with vertices $V(T)$ and arcs $A(T)$.
- Each vertex $v \in V(T)$ is called a *job type* and is labeled with a pair $\langle E(v), D(v) \rangle$. For each *job* that is of type v, $E(v) \in \mathbb{N}_{>0}$ is an upper bound on its required execution time, and $D(v) \in \mathbb{N}_{>0}$ is its relative deadline.
- Each arc $(u, v) \in A(T)$ is labeled with a minimum inter-release separation time $P(u, v) \in \mathbb{N}_{\geqslant 0}$.
- One vertex $v_0 \in V(T)$, denoted $S(T)$, is called the *start vertex* of T.

We denote the unique successor of vertex u with $succ(u)$, and note that $succ(u) = v$ if and only if $(u, v) \in A(T)$.

When the system is running, each task T releases a possibly infinite sequence of jobs $[J_0, J_1, J_2, \ldots]$, where each job corresponds to one of T's job types. Intuitively, a job sequence is produced by traversing the graph of T, starting at vertex $S(T)$. Every time a vertex is visited, a job of the corresponding job type is released. Before the next vertex can be visited, the task must wait for at least the minimum inter-release separation time that is labeled on the arc leading there.

Each job J_i in a job sequence is specified by a triple $(r(J_i), e(J_i), d(J_i)) \in \mathbb{R}^3$, where $r(J_i)$ is the job's absolute release time, $d(J_i)$ its absolute deadline and $e(J_i)$ its execution time requirement. A job sequence is said to be *generated*

[1] The notation used for GMF tasks in this paper is different from, though equivalent to, the notation used in [2].

by task T if and only if there is a path $[v_0, v_1, v_2, \ldots]$ through T such that the following hold for all $i \geqslant 0$:

1. $v_0 = S(T)$,
2. $r(J_{i+1}) \geqslant r(J_i) + P(v_i, v_{i+1})$,
3. $e(J_i) \leqslant E(v_i)$,
4. $d(J_i) = r(J_i) + D(v_i)$.

A job sequence is generated by a task set τ if and only if it is an interleaving of job sequences generated by the tasks $T \in \tau$.

We assume that the tasks satisfy the *localized Monotonic Absolute Deadlines* (l-MAD) property described in [2]. That is, we assume $D(u) \leqslant P(u, v) + D(v)$ for all arcs (u, v). This property guarantees that all jobs released by the same task have their (absolute) deadlines ordered in the same order as their release times.

2.2. Modeling Shared Resources

In the plain GMF model described above, all jobs are completely independent; there is no way to model contention between jobs for shared resources. Here we extend the GMF model to include non-preemptable shared resources. The extended model allows us to express which resources may be used by jobs of each job type, and for how long. We refer to the extended model as the GMF-R task model.

When a job is granted access to a resource, we say that it *locks* the resource, and then *holds* it for some time before finally *unlocking* it. If a resource is already held by some job, it cannot be locked again until it has been unlocked by the job holding it. A job trying to lock an already locked resource is said to be *blocked* on that resource, and cannot continue to execute until the resource has been unlocked. Note that a job may be preempted while holding a resource, but no other job may use that resource until it is unlocked.

Each job type has a *worst-case access duration* to each resource. Each time a particular resource is locked by a job, that job will execute for at most this duration before unlocking it again. We do not assume any a priori knowledge about exactly when a job locks a resource. We only assume knowledge about which resources it may lock, and for how long it may execute while holding them in the worst case.

Formally, a GMF-R task set is a triple (τ, ρ, α), such that

- τ is a GMF task set,
- ρ is a set of resources,
- $\alpha : V(\tau) \times \rho \to \mathbb{N}_{\geqslant 0} \cup \{\bot\}$ is a function mapping job types and resources to their worst-case access durations,

where $V(\tau) = \bigcup_{T \in \tau} V(T)$ is the set of all job types in τ.

The worst-case access duration of jobs of type v to resource $R \in \rho$ is given by $\alpha(v, R)$. If $\alpha(v, R) = \perp$, then jobs of type v do not use resource R.[2] Otherwise, $\alpha(v, R) \leqslant E(v)$ is assumed. We let $\alpha^{\max}(T, R)$ denote the maximum access duration to resource R by any job type in task T. Figure 2 shows an example GMF-R task set with two GMF tasks and two shared resources (only $\alpha(v, R) \neq \perp$ are shown).

A single job may use several different resources, possibly at the same time, but resource accesses must be *properly nested*. That is, if a job locks resource R_1 and afterwards locks R_2 before unlocking R_1, it must unlock R_2 before unlocking R_1. A job may lock the same resource R several times during its execution, but each time executing with it held for at most its worst-case access duration to R. A job must unlock all resources that it holds before finishing execution. Any sequence of locking and unlocking events that follows the above rules is called a *valid access pattern*.

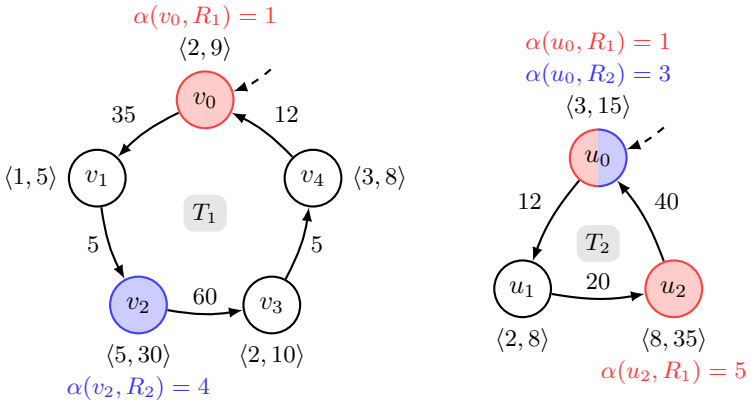

Figure 2: A GMF-R task set with two tasks and shared resources R_1 and R_2.

2.3. Scheduling Algorithms and Feasibility

For a job sequence to be successfully scheduled, all jobs must finish their execution before their deadlines. That is, each job J in the sequence must be exclusively executed for $e(J)$ time units (not necessarily continuously) between $r(J)$ and $d(J)$. A job is said to be *active* during the time interval between its release time and the time point where it finishes execution.

A *scheduling algorithm* decides at each time point which active, non-blocked job (if any) to execute. A scheduling algorithm can know the current system

[2]Note that $\alpha(v, R) = 0$ is useful to express that jobs of type v can be forbidden to execute while some other job holds R, but do not hold R themselves. This can be used to model non-preemptable sections in jobs.

state and how jobs have been released in the past. It does not know what will happen in the future, other than what is specified by the task model.

Definition 2.1 (Feasibility and Optimality)**.** *A* GMF-R *task set* (τ, ρ, α) *is* feasible *if and only if there exists some scheduling algorithm that can successfully schedule all job sequences generated by* τ*, for all valid access patterns to the resources in* ρ *by jobs in the sequence. A scheduling algorithm is* optimal *if and only if it can successfully schedule all feasible task sets.*

2.4. A Motivating Example

Here we present a simple example task set, showed in Figure 3, that illustrates the difficulties in scheduling GMF-R task sets in an optimal way. This task set is feasible, which can be confirmed with the feasibility test presented later in Section 5, but it is not schedulable by any previous scheduling algorithm that we know of. To see why, consider a scenario where a job J from T_1 is released at time t, and locks resource R_1. At $t + 1$, a job J' from T_2 is released. We now have to decide whether J' should preempt J. If J' does preempt J, locks R_2, and immediately afterwards T_3 releases a job of type w_0, then w_0 will be blocked for up to 3 time units waiting for R_1 and R_2 to be unlocked, missing its deadline. If instead J' does not preempt J, and when J is finished T_3 releases a job of type w_1, then one of the remaining two jobs must miss its deadline, because there is only 4 time units available to finish 5 time units of work.

In fact, to make a safe decision about whether J' should preempt J, we must know the state of T_3, and analyze its possible behaviors in the near future. The ability to do so is a key mechanism in the optimal scheduling algorithm that we present in the following section. It can be noted that the lack of branching in GMF-R tasks allows us to efficiently make the necessary predictions to achieve optimal scheduling. While the scheduling algorithm we propose could be applied to more general task models that include branching, such as the digraph real-time task model [4] extended with resources, its optimality would be lost.

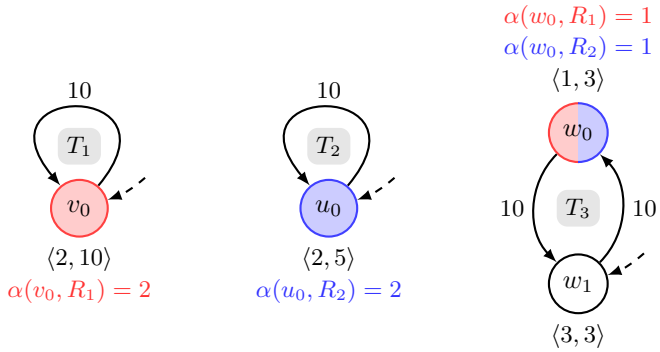

Figure 3: A feasible GMF-R task set.

3. The Resource Deadline Protocol

The resource deadline protocol (RDP) is a resource sharing protocol, designed to extend the earliest deadline first (EDF) scheduling algorithm to handle shared resources. We call the resulting scheduling algorithm EDF+RDP. We will show that EDF+RDP is an optimal scheduling algorithm for GMF-R task sets.

EDF+RDP uses what we call *virtual deadlines* to schedule jobs. It schedules jobs in a similar way to EDF, but uses virtual deadlines instead of absolute deadlines for scheduling decisions. That is, at each time point, EDF+RDP chooses to run the job with the earliest virtual deadline. It is then up to the RDP part of EDF+RDP to assign virtual deadlines to jobs in a way that guarantees the desired properties. It does this by potentially lowering the virtual deadlines (and thereby increasing the priorities) of jobs that are currently holding resources. The virtual deadline of a job therefore represents not only the urgency of the job itself, but also the urgency of releasing the resources that the job is currently holding. To assign virtual deadlines in an optimal way, we must be able to determine how urgent it is that a certain resource becomes unlocked. We capture this urgency by introducing the concept of a *resource deadline*, which is described in the following section.

3.1. Resource Deadlines

RDP relies on the idea that we can predict, at any time, exactly the earliest future time point where a not-yet-released job can have a deadline. In particular, we are interested in the deadlines of future jobs that may need some resource R. The earliest possible such deadline we call the resource deadline of R.

Definition 3.1 (Resource deadline)**.** *The resource deadline $\Delta(R, t)$ of resource R at time point t is exactly the earliest time point when some job that is not yet released at t and that may need R can* potentially *have a deadline, without violating the semantics of the task model.*

In other words, at time t, let $[J, J', \ldots, J'']$ be the job sequence that has been released by a task set (τ, ρ, α) so far (i.e., only containing jobs with release times no larger than t). Then $\Delta(R, t)$ is the smallest value such that the following is satisfied, for some potential future job J''':

1. Some $[J, J', \ldots, J'', \ldots, J''']$ is generated by τ,
2. J''' may use R,
3. $\Delta(R, t) = d(J''')$,
4. $r(J''') \geqslant t$.

Note that no future job that uses R actually has to get a deadline at $\Delta(R, t)$, as long as it is *possible*, given the task model and the system state at time t. We will show how resource deadlines can be efficiently computed at run-time in Section 7.

Example 3.2. *Consider the task system in Figure 2 and the following run-time scenario, illustrated in Figure 4. At time* 115*, we want to know the resource deadline* $\Delta(R_1, 115)$*. The latest job released by task* T_1 *was of type* v_3 *at time* 111*, and the latest job released by task* T_2 *was of type* u_1 *at time* 102*. We can see that the next job of* T_1 *that may need* R_1 *is of type* v_0*. The earliest possible deadline of the next job of type* v_0 *is at* $111 + 5 + 12 + 9 = 137$*. Similarly, the next job of* T_2 *that may need* R_1 *is of type* u_2*, and can have a deadline earliest at time* $102 + 20 + 35 = 157$*. The earliest possible time when some future job that needs* R_1 *may have a deadline is therefore at time* 137*, and* $\Delta(R_1, 115) = 137$*.*

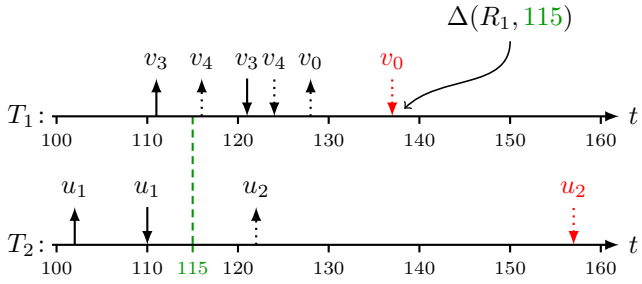

Figure 4: At time 115 we want to know the resource deadline for R_1. The solid arrows indicate release times and deadlines of the latest jobs from T_1 and T_2 (of types v_3 and u_1, respectively). The dotted arrows indicate the earliest possible release times and deadlines of future jobs. We can see that the earliest possible deadline of a future job that uses R_1 is at time 137.

3.2. The EDF+RDP Scheduling Algorithm

In EDF+RDP we use virtual deadlines to represent the urgency of executing jobs. The urgency of executing a job depends not only on the job itself (i.e., its absolute deadline), but also on whether the resources that it holds might be needed by some other job. We introduced resource deadlines to capture this latter aspect of the urgency.

By combining these aspects of urgency, we can now present the complete EDF+RDP scheduling algorithm, which is defined by the following four rules:

1. At each time point, EDF+RDP executes the active job J with the earliest virtual deadline $v(J)$. If several jobs share this earliest virtual deadline, then those jobs are executed in first-come, first-served order.
2. When a job J is released, it gets a virtual deadline equal to its absolute deadline:
$$v(J) \leftarrow d(J).$$
3. When a job J locks a resource R at time t, it gets a virtual deadline based on the resource deadline $\Delta(R, t)$:
$$v(J) \leftarrow \min(v(J),\ \Delta(R, t)).$$

4. When a job unlocks a resource, it regains the virtual deadline that it had before locking that resource.

Note that a virtual deadline only changes when the corresponding job locks or unlocks some resource. In particular, the virtual deadline set in rule 3 depends only on the resource deadline *at the time of locking*.

4. Properties of EDF+RDP

In this section we will show some of the desired properties of EDF+RDP, namely the deadlock freedom, the bounded number of preemptions, and the ability to share a common stack among all jobs. The optimality of EDF+RDP for GMF-R task sets will be established in Section 5. Note that the properties shown in this section hold for any task model that we can compute resource deadlines for, while the optimality is specific for GMF-R (and models generalized by GMF-R).

First we prove an auxiliary lemma.

Lemma 4.1 (No direct blocking)**.** *No job will try to lock a resource that is already held by another job, when scheduled by* EDF+RDP*.*

Proof. We will show that a job J will never try to lock a resource R that is already held by another job J'.

First consider the case where J' already holds R when J is released. We know (by rule 3 of EDF+RDP and the definition of Δ) that when J' locked R, the virtual deadline of J' was set so that $v(J') \leqslant d(J)$. We also know that $v(J) = d(J)$ when J is released (by rule 2). By rule 1, J will not be chosen for execution (and cannot try to lock R) as long as $v(J') \leqslant v(J)$. It is clear from rules 3 and 4 that $v(J') \leqslant v(J)$ will continue to be the case at least until J' unlocks R. Therefore J cannot try to lock R until after R has been unlocked.

Now consider the case where J' locks R after J has been released. In this case, J' must have executed at some time point t after J's release (at the time of locking R). Since J' was executed at t, it must have been chosen at t by EDF+RDP (according to rule 1) in favor of J for execution. By rules 3 and 4, $v(J')$ cannot be later than it was at time t until after R is unlocked. Therefore, J' will continue to be chosen for execution in favor of J while R is locked. □

An interesting observation based on Lemma 4.1 is that a system using EDF+RDP has no need to explicitly enforce mutual exclusion, because the scheduling already provides an implicit guarantee that there will be no conflicts.

From Lemma 4.1 it follows that EDF+RDP will never let the system be idle as long as there is any active job.

Corollary 4.2 (Work-conserving scheduling)**.** *If there are active jobs in the system,* EDF+RDP *will execute one of them.*

Proof. From rule 1 it is clear that as long as there are active jobs, EDF+RDP will pick one for execution, and from Lemma 4.1 we know that this job can always execute. □

We can now see that there can be no deadlocks in a system scheduled by EDF+RDP.

Theorem 4.3 (Deadlock freedom)**.** *If a task set is scheduled by* EDF+RDP*, there can be no deadlocks.*

Proof. From Corollary 4.2 we know that the system can never be in a state where there are active jobs and no job executes. It immediately follows that there cannot be any deadlocks in the system. □

Before we show that the number of context switches is bounded and all jobs can share the same run-time stack, we establish the following auxiliary lemma.

Lemma 4.4 (No preemption by older jobs)**.** *When a job J has started execution, no other job that was active at J's start time will execute until J is finished.*

Proof. Let t_a be the start time of J, i.e., the time when J was first chosen for execution, and t_b be J's finishing time. We will show that no other job J' that was active at t_a will execute in the interval $[t_a, t_b]$.

Assume for the sake of contradiction that such a J' executes in $[t_a, t_b]$, and let $t \in [t_a, t_b]$ be the earliest time when J' was chosen for execution over J. We know that when J started at t_a, it was chosen by EDF+RDP according to rule 1 over all other active jobs (including J'). If J' is to be chosen over J at time t, then either $v(J)$ must have increased (been postponed) or $v(J')$ decreased since t_a. By rules 2-4 of EDF+RDP, $v(J)$ can never increase beyond its initial value at t_a. Also, $v(J')$ can only decrease if J' first executes (and locks a resource), which it did not do in $[t_a, t)$. This contradicts the assumption that J' was chosen over J for execution in $[t_a, t_b]$. □

For any scheduling algorithm it is important to keep the number of preemptions, and thereby the number of context switches, as low as possible. For EDF+RDP, there can be at most one preemption per job release, the same as with EDF.

Theorem 4.5 (At most one preemption)**.** *If a task set is scheduled by* EDF+RDP*, there will be at most one preemption per job release.*

Proof. From Lemma 4.4 it follows that a job J only can preempt jobs that were released before J. It also follows from Lemma 4.4 that as soon as J has started execution, none of the jobs that were released before J will execute until J has finished. Consequently, the only possibility for J to preempt another job is at the time when J first starts execution. □

Similar to the stack resource policy [5], EDF+RDP guarantees that all jobs can share a common run-time stack without conflicts. This can simplify implementations and substantially reduce memory requirements.

Theorem 4.6 (The stack can be shared)**.** *For any task set, all jobs can share a common stack space without conflicts if they are scheduled by* EDF+RDP*.*

Proof. To see that the stack can be shared, it is enough to show that once a job J starts execution (and is allocated space on the stack), no other job that has already been allocated stack space will execute until J is finished. This follows immediately from Lemma 4.4. □

We also note that for GMF-R task sets, EDF+RDP will never execute a job before all other jobs that were previously released by the same task have finished (due to the l-MAD property). This fact can be important in implementations where jobs depend on results produced by previous jobs from the same task.

5. Feasibility and Optimality

Here we present a sufficient and necessary feasibility test for GMF-R task sets. We will show that EDF+RDP meets all deadlines for task sets that pass the test, from which the optimality of EDF+RDP follows.

5.1. A Feasibility Test for GMF-R Task Sets

The test builds on the general feasibility test framework based on *demand-bound functions*. Demand-bound functions have been used for checking feasibility of GMF task sets without resources [2].

Definition 5.1 (Demand-bound function)**.** *A demand-bound function, $dbf(T, \ell)$, gives the maximum total workload of jobs generated by task T that are both released and have deadlines within any time interval of length ℓ.*

In other words,

$$dbf(T,\ell) \;=\; \max_{[J_0,J_1,J_2,\ldots]\in\mathfrak{J}} \left(\max_{t\in\mathbb{R}} \left(\sum \{ e(J_i) \mid r(J_i) \geqslant t \wedge d(J_i) \leqslant t+\ell \} \right) \right),$$

where $\mathfrak{J}$ is the set of all job sequences generated by T. Intuitively, $dbf(T, \ell)$ captures the maximum workload from task T that *must* be executed in any time interval of length ℓ. Note that $dbf(T, \ell)$ is a non-decreasing step function that changes only at integer values of ℓ.

Example 5.2. *Consider task T_1 in Figure 2. The maximum possible workload of jobs from T_1 that are both released and have deadline within any interval of length 40 is given by $dbf(T_1, 40)$. An example workload is given by a sequence of three jobs of types v_3, v_4 and v_0. It is easy to see that these jobs fit into the interval. The maximum total workload of the three jobs is $2+3+2=7$. In fact, this is the maximum workload of any sequence of jobs from T_1 that fit into an interval of length 40, and therefore $dbf(T_1, 40) = 7$.*

To analyze feasibility for GMF-R task sets, we introduce the concept of a *resource-constrained demand-bound function*.

Definition 5.3 (Resource-constrained *dbf*)**.** *The resource-constrained demand-bound function, $dbf(T, R, \ell)$, is defined exactly as the $dbf(T, \ell)$, but with the extra constraint that at least one of the jobs contributing to the workload is of a type that uses resource R.*

The idea behind $dbf(T, R, \ell)$ is that it captures the maximum workload from task T that must be executed in any interval of length ℓ, such that some job of T may have to wait for R in that interval. If $\alpha^{\max}(T, R) = \perp$, then $dbf(T, R, \ell) = 0$ for all ℓ. In Section 6.1 we will describe how the resource-constrained demand-bound function can be computed in polynomial time, by using the algorithm from [2] for computing the standard demand-bound function.

Example 5.4. *Consider again task T_1 from Figure 2. In the previous example we saw that three jobs of types v_3, v_4 and v_0 produce the maximum workload in any interval of length 40. However, none of these job types uses resource R_2, so that job sequence is not considered for $dbf(T_1, R_2, 40)$. Instead, the maximum workload of jobs that fit into the interval, such that at least one job uses resource R_2, comes from a job of type v_1 followed by a job of type v_2. This workload is $1 + 5 = 6$, and therefore $dbf(T_1, R_2, 40) = 6$.*

We can now present the new feasibility test.

Definition 5.5 (Feasibility test)**.** *A* GMF-R *task set (τ, ρ, α) passes the test if and only if both of the following two conditions hold.*

Condition A: *For all $\ell \geqslant 0$:*

$$\sum_{T \in \tau} dbf(T, \ell) \leqslant \ell.$$

Condition B: *For all $\ell \geqslant 0$, all resources $R \in \rho$, and all tasks $T, T' \in \tau$, such that $T \neq T'$, $\alpha^{\max}(T, R) \neq \perp$ and $dbf(T', R, \ell) > 0$:*

$$\alpha^{\max}(T, R) + dbf(T', R, \ell) + \sum_{T'' \in \tau \setminus \{T, T'\}} dbf(T'', \ell) \leqslant \ell.$$

Intuitively, Condition A captures the case when no job waits for shared resources, and Condition B the case when waiting is involved. The three terms in the left-hand side (LHS) of Condition B represent a task T with a job holding a resource R, a task T' with a job that needs R, and all other tasks, respectively.

We will establish the complexity of the feasibility test in Section 6. It is pseudo-polynomial for task sets with a utilization bounded by a constant strictly smaller than 1. This is the same complexity as for the known feasibility tests for sporadic task sets [6] and GMF task sets without resources [2], and is considered tractable for this type of problem.

5.2. Necessity, Sufficiency and Optimality

We show that the feasibility test is both *necessary* and *sufficient*.

Theorem 5.6 (Necessity)**.** *If a* GMF-R *task set* (τ, ρ, α) *fails the feasibility test, then* (τ, ρ, α) *is infeasible.*

Proof. If Condition A fails for some ℓ, then the tasks in τ can together require to be executed for more than ℓ time units in some interval of length ℓ. Clearly, no scheduler can achieve this.

Consider instead the case where Condition B fails for some combination of ℓ, R, T and T', and let $\ell + k$ be the value of the LHS of Condition B. We know that $k > 0$.

We construct a scenario in which no scheduler can meet all deadlines. First, let the only active job be a job from T that may use resource R for $\alpha^{\max}(T, R)$ time units. Sooner or later, this job must be executed (by any scheduler) or it will miss its deadline. When it is executed, it locks resource R at some time point $t - \epsilon$, where $\epsilon < k$ and $\epsilon \leqslant \alpha^{\max}(T, R)$. It will hold R for as long as possible. At time t, task T' starts releasing the job sequence that corresponds to the value of $dbf(T', R, \ell)$. By the definition of the resource-constrained demand-bound function, $dbf(T', R, \ell)$ time units of work from T' *must* then be executed in the interval $[t, t + \ell]$ to avoid a deadline miss. In addition, at least one job from T' has to wait for R to be unlocked before it can finish, so the job from T must also execute for another $\alpha^{\max}(T, R) - \epsilon$ time units in that interval in order to unlock R. Finally, all other tasks T'' release the job sequences that correspond to $dbf(T'', \ell)$, starting at time t. The total workload from all tasks that must be executed in $[t, t + \ell]$ sums up to

$$\underbrace{\alpha^{\max}(T, R) + dbf(T', R, \ell) + \sum_{T'' \in \tau \setminus \{T, T'\}} dbf(T'', \ell)}_{\ell + k} - \epsilon.$$

In other words, $\ell + k - \epsilon$ time units of work must be executed in an interval of length ℓ, which no scheduler can do. □

Before we can prove the sufficiency of the feasibility test, we need to describe the concept of a *busy period*, which is defined whenever a deadline is missed.

Definition 5.7 (Busy period)**.** *Let* t_d *be the first time instant at which any job misses its deadline when scheduled by* EDF+RDP*. The busy period is the longest time interval* $[t_s, t_d]$*, such that in the whole of* $[t_s, t_d]$ *there are active jobs with absolute (non-virtual) deadlines latest at* t_d*.*

The busy period captures the critical time interval leading up to a deadline miss. If a job has an absolute deadline latest at t_d and is active sometime in the busy period, we call it a *pressing job*. Note that a pressing job, by definition, is both released and has its deadline inside the busy period. A job that executes in the busy period with an absolute deadline after t_d is called a *blocking job*. In the following lemma we show some properties of the busy period and of blocking jobs.

Lemma 5.8 (Contention)**.** *If* EDF+RDP *schedules a task set and has a first deadline miss at t_d, and J is a blocking job executing in the busy period $[t_s, t_d]$, then there exists an earliest time point $t_l \leqslant t_s$ such the following hold:*

(i) J locked a resource R at t_l with $\Delta(R, t_l) \leqslant t_d$, and still held R at t_s.
(ii) J only executes in $[t_l, t_d]$ while holding such an R.
(iii) J is the only blocking job.
(iv) All other jobs $J' \neq J$ that execute in $[t_l, t_d]$ have both $r(J') \geqslant t_l$ and $d(J') \leqslant t_d$.

Proof. We prove the four properties in turn. Figure 5 serves as an illustration.

Property *(i)*: By rules 2, 3 and 4 of EDF+RDP, we know that $v(J') \leqslant d(J')$, for all jobs J'. By the definition of the busy period, there must then exist at least one active, pressing job with virtual deadline latest at t_d at all time points in $[t_s, t_d]$. Hence, if a blocking job J executes in the busy period, we know by rule 1 of EDF+RDP that $v(J) \leqslant t_d$ while it executes (or some pressing job would execute instead).

A blocking job J can only have received $v(J) \leqslant t_d$ by locking some resource R at some time t_l, where $\Delta(R, t_l) \leqslant t_d$. In order to lock a resource it must execute, and therefore it must have locked one such resource at a time point $t_l \leqslant t_s$. Furthermore, J must still hold one such resource at t_s to execute in the busy period, since after unlocking a resource J regains the virtual deadline that it had before locking it. This proves property *(i)*.

Property *(ii)*: Let R be the first resource that J locks, such that property *(i)* is satisfied. Immediately before J locked R we had $v(J) > t_d$. This follows from the fact that resource accesses are properly nested, and the assumption that R is the first resource locked by J such that property *(i)* is satisfied. If $v(J) \leqslant t_d$ when J locked R, then J already held a resource satisfying property *(i)*, contradicting our assumption. As soon as R is unlocked we therefore have $v(J) > t_d$ by property 4 of EDF+RDP, which means that J will not execute any more in the busy period. This proves property *(ii)*.

Property *(iii)*: We assume, without loss of generality, that J is the first blocking job to lock a resource R that satisfies property *(i)*. We know then that J holds R, and consequently that $v(J) \leqslant t_d$, in the interval $[t_l, t_s]$. By rule 1 of EDF+RDP, no job J' with $v(J') > t_d$ can execute in $[t_l, t_s]$ or in the busy period $[t_s, t_d]$. In particular, no such job J' can lock a resource and get $v(J') \leqslant t_d$ after t_l. Since J was the first blocking job to lock a resource that allows it to execute in the busy period, and no other job can lock such a resource after J, we know that at most one blocking job executes in the busy period. This proves property *(iii)*.

Property *(iv)*: Now consider the other jobs $J' \neq J$ that execute in $[t_l, t_d]$, where J is the blocking job that locked the first resource R satisfying property *(i)* at t_l. We know that there were no active jobs J' with $v(J') \leqslant t_d$ at

t_l (because J executes at t_l with $v(J) > t_d$). We have also seen that only jobs J' with $v(J') \leqslant t_d$ can execute in $[t_l, t_d]$. Since there were no such jobs active at t_l, only jobs released after t_l will execute in that interval. Also, the jobs J' must have $v(J') \leqslant t_d$ already when they start execution (or they would not start before t_d at all). Since $v(J') = d(J')$ when J' starts, we know that $d(J') \leqslant t_d$. This proves property *(iv)*. □

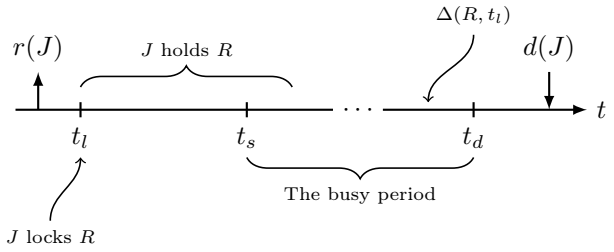

Figure 5: If a blocking job J executes in the busy period, it must have locked some resource R in a way similar to the above, as is claimed by property *(i)* of Lemma 5.8.

With the above lemma we can show that EDF+RDP can miss a deadline only for task sets that fail the feasibility test.

Theorem 5.9 (Sufficiency)**.** *If a* GMF-R *task set* (τ, ρ, α) *passes the feasibility test, then* EDF+RDP *successfully schedules* (τ, ρ, α).

Proof. We prove the contrapositive: if EDF+RDP misses a deadline for a GMF-R task set (τ, ρ, α), then (τ, ρ, α) fails the feasibility test. Let $[t_s, t_d]$ be the busy period, and let J be the job that misses its deadline at t_d. From Lemma 5.8 we know that EDF+RDP executes at most one blocking job in the busy period. We separately consider the cases in which there is no blocking job and exactly one blocking job, respectively.

Only pressing jobs execute in the busy period: Let $\ell = t_d - t_s$ be the length of the busy period. By the definition of the busy period we know that J is a pressing job. Since EDF+RDP is work conserving (by Corollary 4.2) and there are active jobs during the entire busy period, it must be the case that EDF+RDP executes some jobs during the whole of $[t_s, t_d]$. We know that J missed its deadline at t_d, so the total time spent executing *other* jobs during $[t_s, t_d]$ must be strictly more than $\ell - e(J)$.

Since all pressing jobs are both released and have their deadlines in the busy period, we know that the total workload executed in $[t_s, t_d]$ is bounded from above by $\sum_{T \in \tau} dbf(T, \ell)$. The workload of jobs $J' \neq J$ executing in $[t_s, t_d]$ is bounded by $\sum_{T \in \tau} dbf(T, \ell) - e(J)$. It must be the case that

$$\sum_{T \in \tau} dbf(T, \ell) - e(J) > \ell - e(J),$$

and Condition A cannot hold.

Exactly one blocking job executes in the busy period: Let R be the first resource locked by the blocking job that satisfies property *(i)* of Lemma 5.8, and let t_l be the time when it was locked. Let $\ell = t_d - t_l$.

We know that the blocking job is released latest at t_l and has deadline after t_d. Since the tasks satisfy the l-MAD property (described in Section 2.1), no job J' that is from the same task T as the blocking job can have both $r(J') \geqslant t_l$ and $d(J') \leqslant t_d$. The blocking job is therefore the only job from T that can execute in $[t_l, t_d]$ by property *(iv)* of Lemma 5.8. We can see then, by property *(ii)* of Lemma 5.8, that the executed workload of jobs from T in $[t_l, t_d]$ is bounded by $\alpha^{\max}(T, R)$. Since $\Delta(R, t_l) \leqslant t_d$, there must be some other task T' that can release a sequence of jobs from t_l on, in which at least one job needs R and has a deadline before t_d. By property *(iv)* of Lemma 5.8, only jobs with both release time and deadline in $[t_l, t_d]$ execute in that interval, so the workload of task T' in $[t_l, t_d]$ is bounded by $dbf(T', R, \ell)$.

Similarly, the workload of each of the remaining tasks T'' in the interval $[t_l, t_d]$ is bounded by $dbf(T'', \ell)$.

The total workload of jobs executing in $[t_l, t_d]$ is thus bounded by

$$W = \alpha^{\max}(T, R) + dbf(T', R, \ell) + \sum_{T'' \in \tau \setminus \{T, T'\}} dbf(T'', \ell).$$

Note that W is equal to the LHS of Condition B. We know that EDF+RDP is work conserving and that there are active jobs in the entire interval $[t_l, t_d]$. Since J missed its deadline, strictly more than $\ell - e(J)$ time units must therefore have been spent executing other jobs than J in $[t_l, t_d]$. The total workload of the other jobs is bounded from above by $W - e(J)$. Therefore, $W - e(J) > \ell - e(J)$, and Condition B cannot hold. □

We can now establish the optimality of EDF+RDP for GMF-R tasks sets.

Theorem 5.10 (Optimality)**.** EDF+RDP *successfully schedules all feasible* GMF-R *task sets.*

Proof. From Theorems 5.6 and 5.9 we know that the feasibility test is both necessary and sufficient. Further, we know from Theorem 5.9 that EDF+RDP successfully schedules all task sets that pass the test. The optimality of EDF+RDP follows directly. □

6. Complexity of the Test

The following two questions must be answered in order to evaluate the feasibility test from Section 5:

1. How do we compute resource-constrained demand-bound functions?
2. For which values of ℓ do we need to evaluate Conditions A and B in the feasibility test?

We show how resource-constrained demand-bound functions can be computed in polynomial time in Section 6.1. In Section 6.2 we show that it is enough to evaluate Conditions A and B for pseudo-polynomially many different values for ℓ, if the utilization of the task set is bounded by a constant strictly smaller than 1.

The feasibility test can therefore be evaluated in pseudo-polynomial time for such bounded-utilization task sets, similarly to feasibility tests for sporadic task sets [6] and GMF task sets without resources [2].

6.1. Computing Resource-Constrained dbfs

Here we describe a polynomial-time algorithm (Figure 6) for computing resource-constrained demand-bound functions. We make use of the fact that it is already known how to compute the standard demand-bound function in polynomial time [2], by invoking it as a subroutine in the algorithm.

```
1:  if α^max(T, R) = ⊥ then
2:    return 0
3:  end if
4:
5:  if ℓ ⩾ Σ_{(u,v)∈A(T)} P(u, v) + max_{v∈V(T)}(D(v)) then
6:    return dbf(T, ℓ)
7:  end if
8:
9:  e^max ← 0
10: for v ∈ V(T) do
11:   e ← 0
12:   t ← ℓ
13:   r ← false
14:   while t ⩾ D(v) do
15:     e ← e + E(v)
16:     t ← t − P(v, succ(v))
17:     if α(v, R) ≠ ⊥ then
18:       r ← true
19:     end if
20:     v ← succ(v)
21:   end while
22:   if (r = true) ∧ (e > e^max) then
23:     e^max ← e
24:   end if
25: end for
26: return e^max
```

Figure 6: Algorithm for computing $dbf(T, R, \ell)$.

The algorithm in Figure 6 first checks whether any jobs from T uses resource R at all (lines 1-3), otherwise it returns 0. It then checks (lines 5-7) whether

the value of ℓ is large enough to cycle through the entire graph of T and include each job type at least once. If that is the case, some job type using resource R will for sure be included in the interval, and hence $dbf(T, R, \ell) = dbf(T, \ell)$. We can use the known polynomial-time algorithm [2] to compute $dbf(T, \ell)$.

For smaller ℓ we cannot use the algorithm for computing $dbf(T, \ell)$ because we must make sure that some job type that uses R is included. To compute the $dbf(T, R, \ell)$ in this case, we try all possible start vertices (the first job type to be released in the interval of length ℓ) and walk forward in the graph. For each start vertex, we simply add up as much execution demand as possible in an interval of size ℓ, and check if we visit at least one job type that uses R. Note that the inner while-loop will iterate no more than $\mathcal{O}(|V(T)|)$ times (since ℓ is small), and the complexity of the algorithm is $\mathcal{O}(|V(T)|^2)$ in this case.

6.2. Bounding the Values of ℓ in the Test

In the feasibility test, the inequalities in Conditions A and B must hold for all $\ell \geqslant 0$. The LHS of both inequalities change only at integer-valued ℓ, so it is enough to check integers ℓ. Still, there are infinitely many such ℓ, and we cannot check them all. We show here that it is enough to check the inequalities for ℓ smaller than a given bound.

We derive such upper bounds $\ell_A^{\max}$ and $\ell_B^{\max}$ on the values of ℓ that must be checked in Conditions A and B, respectively. Both $\ell_A^{\max}$ and $\ell_B^{\max}$ are pseudo-polynomial in the task set representation, if the *utilization* of the task set is bounded from above by a constant strictly smaller than 1. The total number of different values for ℓ that needs to be checked is therefore pseudo-polynomial.

Definition 6.1 (Utilization)**.** *The utilization of a task set is the maximum execution demand that it can create per time unit, asymptotically.*

For a GMF *task set τ, the utilization $U(\tau)$ is computed as*

$$\begin{aligned} E^{\text{sum}}(T) &= \textstyle\sum_{v \in V(T)} E(v), \\ P^{\text{sum}}(T) &= \textstyle\sum_{(u,v) \in A(T)} P(u, v), \\ U(T) &= E^{\text{sum}}(T)/P^{\text{sum}}(T), \\ U(\tau) &= \textstyle\sum_{T \in \tau} U(T). \end{aligned}$$

We first derive $\ell_A^{\max}$. Consider any sequence of jobs released by a single task T in an interval of length ℓ. If the job sequence is long enough, some job types will be represented several times in the sequence. We can divide the sequence into one part consisting of *full cycles* through the graph of T, and one part consisting of jobs that are *not* part of a full cycle. If we make the part with full cycles as large as possible, then at most one job per job type can be in the other part.

The value $\ell \cdot U(T)$ is an upper bound of the workload of jobs in the full cycles (because full cycles generate at most $U(T)$ amount of work per time unit). Also, $E^{\text{sum}}(T)$ is clearly an upper bound of the workload that is not part of a full cycle. Therefore, it must be the case that

$$E^{\text{sum}}(T) + \ell \cdot U(T) \geqslant dbf(T, \ell)$$

and

$$\sum_{T \in \tau} [E^{\text{sum}}(T) + \ell \cdot U(T)] \geqslant \sum_{T \in \tau} dbf(T, \ell).$$

Note that the RHS of the inequality above is equal to the LHS in Condition A. Any ℓ that violates the inequality in Condition A must therefore be bounded so that

$$\sum_{T \in \tau} [E^{\text{sum}}(T) + \ell \cdot U(T)] > \ell.$$

By rearranging some terms in the above equation we get

$$\frac{\sum_{T \in \tau} E^{\text{sum}}(T)}{1 - U(\tau)} > \ell,$$

which gives our bound

$$\ell_A^{\max} = \frac{\sum_{T \in \tau} E^{\text{sum}}(T)}{1 - U(\tau)}.$$

The value of $\ell_A^{\max}$ is clearly pseudo-polynomial if $U(\tau)$ is bounded by some constant strictly smaller than one. This bound is in fact equivalent to the bound derived in [4] for a similar problem concerning the more general *digraph real-time task model*.

Now consider $\ell_B^{\max}$. One can see that $\ell_B^{\max} = \ell_A^{\max}$ is a valid bound also for Condition B. However, that bound is unnecessarily large for Condition B, so we instead derive another, much smaller $\ell_B^{\max}$. This is not necessary to establish the complexity of the feasibility test, but will substantially speed up implementations.

Let $E^{\max}(T)$ be the largest execution time requirement of all job types in task T, and let $D^{\max}(T)$ be the longest (relative) deadline. By definition we have $\alpha^{\max}(T, R) \leqslant E^{\max}(T)$ and $E^{\max}(T) \leqslant dbf(T, D^{\max}(T))$. Since $dbf(T, \ell)$ is monotonically non-decreasing in ℓ, we have for all $\ell \geqslant D^{\max}(T)$:

$$\alpha^{\max}(T, R) \leqslant dbf(T, \ell).$$

We also know, by definition, that for all ℓ:

$$dbf(T, R, \ell) \leqslant dbf(T, \ell).$$

It follows that the LHS of Condition A is at least as big as the LHS of Condition B for any $\ell \geqslant \max_{T \in \tau}(D^{\max}(T))$. If there exists such an ℓ for which Condition B does not hold, we can be sure that Condition A will not hold for this ℓ either. It is therefore not necessary to check these values for ℓ also in Condition B, and as the upper bound we can use

$$\ell_B^{\max} = \max_{T \in \tau}(D^{\max}(T)).$$

7. Implementing EDF+RDP

To implement EDF+RDP we must be able to compute resource deadlines efficiently at run-time. Here we show how this can be done for GMF-R task sets.

For any GMF-R task, we can pick two job types from that task and statically calculate the minimum amount of time that must pass between a release of a job of the first type and a deadline of a job of the second type. We can use such pre-computed values to efficiently evaluate resource deadlines at run-time. In particular, for each job type v and resource R, we want to pre-compute the minimum amount of time that must pass between a release of v and a deadline of some job (from the same task) that is released no earlier and can use resource R. We denote this value $\delta(v, R)$.

Example 7.1. *Consider task T_2 in Figure 2. After the release of a job of type u_1, the next job from T_2 that can use R_1 is of type u_2. Between a release of u_1 and the next deadline of u_2 there must be at least $20 + 35 = 55$ time units. Hence, $\delta(u_1, R_1) = 55$.*

Formally, $\delta(v, R)$ is defined as follows. Let $succ^i(v)$ denote the ith successor of v, i.e., $succ^0(v) = v$ and $succ^i(v) = succ(succ^{i-1}(v))$ if $i > 0$. If there are no job types in v's task T that use R, i.e., if $\alpha^{\max}(T, R) = \bot$, then $\delta(v, R) = \infty$. Otherwise,

$$\begin{aligned} k &= \min\{i \geqslant 0 \mid \alpha(succ^i(v), R) \neq \bot\}, \\ \delta(v, R) &= \sum_{i=0}^{k-1} P(succ^i(v), succ^{i+1}(v)) + D(succ^k(v)). \end{aligned}$$

Clearly, each $\delta(v, R)$ can be computed in $\mathcal{O}(|V(T)|)$ time, where $v \in V(T)$.

Now consider how these pre-computed values can be used to evaluate resource deadlines at run-time. If the next job to be released by a task T is of type v, then at least $\delta(v, R)$ time must pass after that release before a deadline of any future job from T that needs R. To find the earliest possible absolute time point for this deadline, we need to keep track of two things:

1. The type of T's next job.
2. The earliest time point where that job can be released.

We keep a vector G containing the relevant system state, indexed by the tasks $T \in \tau$. Each entry $G[T]$ is a pair (v, t), containing the type of T's next job and the earliest time point where it could possibly be released without violating the inter-release separation constraints. To keep the vector updated, we simply update one entry per job release. If T releases a job of type v at time t:

$$G[T] \leftarrow (succ(v), t + P(v, succ(v))).$$

To compute a resource deadline $\Delta(R, t)$ at run-time, we traverse G, and for each entry calculate the earliest possible time point where a job using R from the corresponding task can have a deadline. If the current time is t and

$G[T] = (v, t')$, then the earliest time point where the next job from T can be released is $\max(t, t')$, and the earliest deadline of a future job from T that needs R is $\max(t, t') + \delta(v, R)$. The resource deadline can therefore be computed as

$$\Delta(R, t) \leftarrow \min_{(v,t') \in \mathfrak{G}} (\max(t, t') + \delta(v, R)),$$

where $\mathfrak{G} = \{G[T] \mid T \in \tau\}$.

When the system is first started at time t, the entries of G are initialized such that $G[T] \leftarrow (S(T), t)$, where $S(T)$ is the start vertex of T.

Following the EDF+RDP rules, we need to evaluate a resource deadline only when a resource is being locked. The overhead of computing resource deadlines is therefore $\mathcal{O}(|\tau|)$ per lock operation.

8. Related Work

Sha et al. proposed the classic *priority inheritance* and *priority ceiling protocols* [7]. They are mainly used to improve fixed-priority scheduling of systems with shared resources by effectively bounding the worst-case blocking times.

Baker proposed the *stack resource policy* (SRP) [5], which works well with dynamic-priority scheduling algorithms. An interesting property of SRP is that it allows all jobs to share a common run-time stack without conflicts, a property which is shared with the algorithm proposed in this paper. *Multi-unit* resources are also supported by SRP. Baruah [8] later showed that a particular instantiation of SRP combined with EDF is optimal for scheduling sporadic task sets with shared resources. Unfortunately, this optimality does not carry over to GMF-R task sets for the reasons discussed in Section 2.4.

Other relevant work include the *dynamic priority ceiling protocol* by Chen and Lin [9], and the *dynamic deadline modification strategy* by Jeffay [10]. Both essentially target a special case of the sporadic task model, where relative deadlines are equal to periods. The scheduling approach taken in this paper resembles Jeffay's, but targets the more general GMF task model.

The *absolute-time ceiling protocol* by Guan et al. [11] handles resource sharing with the digraph real-time task model [4], of which GMF is a special case. That protocol does not provide optimal scheduling, and restricts itself to systems without nested resource accesses.

9. Conclusions

We have introduced the EDF+RDP algorithm and a feasibility test for GMF task sets with shared resources. We have shown that EDF+RDP is optimal for this task model, and has a range of other desirable properties. For the feasibility test, we have shown that it is both sufficient and necessary, and runs as efficiently as the known feasibility test that does not consider shared resources. Previously, optimal scheduling algorithms have not been known for flexible task models such as GMF when shared resources are used.

The key to the optimality of EDF+RDP is its ability to predict possible resource contention in future system states, based on the current system state. The GMF task model is well suited for this because the order in which jobs are released by a single task is fixed, even though the jobs may have differing parameters and non-deterministic release times. This can be contrasted to even more general task models, such as the digraph real-time task model [4], which expresses job releases with an arbitrary directed graph. The non-deterministic branching inherent to such models makes optimal scheduling very hard to achieve when shared resources are considered. Indeed, while EDF+RDP is applicable to such systems (resource deadlines can be computed in a very similar way), the optimality is lost.

It is interesting to note that the results of this paper can be used in frameworks for hierarchical scheduling and compositional analysis, where several task sets are executed on the same platform but on isolated virtual servers. The feasibility test described earlier in this paper can easily be extended to this case by replacing the RHS of both Conditions A and B (i.e., the ℓ) with a supply-bound function $sbf(\ell)$, which lower-bounds the processor supply to the server in any time window of size ℓ. In such a setting we might have to use another value for the bound $\ell_A^{\max}$ though. We can construct a component that contains a GMF task set with (internally) shared resources. If the component is scheduled by EDF+RDP on top of a virtual server, it is guaranteed to meet all deadlines if the extended feasibility test is passed.

Acknowledgements

This work was supported in part by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence. We would like to thank the anonymous reviewers for their helpful and insightful comments.

References

[1] A. Mok, Fundamental design problems of distributed systems for the hard real-time environment, Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1983).

[2] S. Baruah, D. Chen, S. Gorinsky, A. Mok, Generalized multiframe tasks, Real-Time Systems 17 (1999) 5–22.

[3] A. Mok, D. Chen, A multiframe model for real-time tasks, in: RTSS, 1996, pp. 22–29.

[4] M. Stigge, P. Ekberg, N. Guan, W. Yi, The digraph real-time task model, in: RTAS, 2011, pp. 71–80.

[5] T. Baker, A stack-based resource allocation policy for realtime processes, in: RTSS, 1990, pp. 191–200.

[6] S. Baruah, A. Mok, L. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, in: RTSS, 1990, pp. 182–190.

[7] L. Sha, R. Rajkumar, J. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, Computers, IEEE Transactions on 39 (9) (1990) 1175–1185.

[8] S. Baruah, Resource sharing in EDF-scheduled systems: A closer look, in: RTSS, 2006, pp. 379–387.

[9] M.-I. Chen, K.-J. Lin, Dynamic priority ceilings: A concurrency control protocol for real-time systems, Real-Time Systems 2 (1990) 325–346.

[10] K. Jeffay, Scheduling sporadic tasks with shared resources in hard-real-time systems, in: RTSS, 1992, pp. 89–99.

[11] N. Guan, P. Ekberg, M. Stigge, W. Yi, Resource sharing protocols for graph-based task systems, in: ECRTS, 2011, pp. 272–281.