Resource Sharing Protocols for Real-Time Task Graph Systems

Nan Guan*†, Pontus Ekberg*, Martin Stigge*, Wang Yi*†
*Uppsala University, Sweden
†Northeastern University, China

***Abstract*—Previous works on real-time task graph models have ignored the crucial resource sharing problem. Due to the non-deterministic branching behavior, resource sharing in graph-based task models is significantly more difficult than in the simple periodic or sporadic task models. In this work we address this problem with several different scheduling strategies, and quantitatively evaluate their performance. We first show that a direct application of the well-known EDF+SRP strategy to graph-based task models leads to an unbounded speedup factor. By slightly modifying EDF+SRP, we obtain a new scheduling strategy, called EDF+saSRP, which has a speedup factor of 2. Then we propose a novel resource sharing protocol, called ACP, to better manage resource sharing in the presence of branching structures. The scheduling strategy EDF+ACP, which applies ACP to EDF, can achieve a speedup factor of $\frac{\sqrt{5}+1}{2} \approx 1.618$, the *golden ratio*.**

I. INTRODUCTION

Embedded real-time processes are typically implemented as event-driven code within an infinite loop. In many cases, a process may contain conditional code, where the branch to be taken is determined by external events at runtime, and the timing requirement along each branch is different. These systems can not be precisely modeled by the simple periodic or sporadic task models. Instead, a natural representation of these processes is a *task graph*: a directed graph in which each vertex represents a code block and each edge represents a possible control flow. Over the years, there have been many efforts to study more and more general graph-based real-time task models to precisely represent complex embedded real-time systems [3], [1], [9], [5], [12].

A common restriction in all these graph-based real-time task models is that the processes coexisting in the system are assumed to be completely independent from each other. However, this assumption rarely holds in actual embedded systems. Usually a process needs to use shared resources (e.g., some peripheral devices or global data structures) to perform functions like sampling, control and communication, or to coordinate with other processes. The practical significance of these graph-based models would be considerably limited without the capability of modeling shared resources.

The shared resource problem has been intensively studied in the context of the simple periodic and sporadic task models. Many protocols have been designed to systematically manage resource contention in scheduling, in order to improve system predictability and resource utilization. Priority Inheritance Protocol [10], Priority Ceiling Protocol [10] and Stack Resource Policy (SRP) [2] are three well-known examples. The main idea behind these protocols is to predict, and to some extent prevent, the potential resource blocking that could happen to important or urgent processes.

To our best knowledge, there has been no previous work addressing the resource sharing problem in graph-based real-time task models. This problem is fundamentally different from, and significantly more difficult than, the resource sharing problem for periodic or sporadic task models. This is mainly because of the "branching" behavior of the graph-based models: a process generally has different resource requirements along different branches, and it only becomes revealed during run-time which branch will be taken. Therefore, it can be very difficult, or even impossible, to make scheduling decisions such that the potentially "bad" behaviors due to resource contention are avoided.

In this paper we study the resource sharing problem for real-time task graph systems. We first study the application of the well-known SRP protocol to task graph systems, and quantitatively evaluate its performance. Then we propose a novel resource sharing protocol, called ACP, to better handle the complex issues arising due to the "branching" in task graph systems. The main results of this paper can be summarized as:

- We show that directly applying EDF+SRP (EDF scheduling extended with the SRP rules) to task graph systems leads to an unbounded *speedup factor*[1].
- By slightly modifying SRP we obtain a new protocol, called saSRP. We show that the EDF+saSRP scheduling strategy has a speedup factor of 2.
- We propose a novel protocol, called ACP, and show that EDF+ACP can achieve a speedup factor of $\frac{1+\sqrt{5}}{2}$, which is the well-known constant known as the *golden ratio*.

This work is presented in the context of the Digraph Real-Time (DRT) task model [12], which is the most general among all the real-time task graph models that are known to be tractable (that can be efficiently analyzed). Since DRT generalizes other models such as RRT [3], non-cyclic GMF [9] and non-cyclic RRT [5] (they can be viewed as special cases of DRT), all the results in this paper are directly applicable also to these models. Further, we assume the shared resources are non-nested, i.e., each task can not hold more than one

[1] Speedup factor is formally defined in Section III-C. We use it to quantitatively evaluate the "quality" of a scheduling strategy: the smaller the better.

resource at the same time. Nested resource accesses can be handled by, for example, group locks [6].

II. Related Work

A naive way of handling resource sharing is, as proposed by Mok [8], to non-preemptively execute the critical sections. However, this approach has the drawback that even the jobs that do not require shared resources suffer the extra blocking. The Priority Inheritance Protocol (PIP) [10] designed by Sha, Rajkumar and Lehoczky can avoid the so-called "priority inversion", in order to guarantee the responsiveness of important tasks in fixed priority scheduling. Under PIP, the worst-case number of blocking suffered by a task is bounded by both the number of lower-priority tasks and resource types used by this task. Priority Ceiling Protocol (PCP) [10] is a deadlock free protocol which also works with fixed-priority scheduling, can limit the blocking to be at most the duration of one outer-most critical section. Baker's Stack Resource Policy (SRP) [2] is a more general protocol that can be used for not only fixed-priority, but also dynamic-priority scheduling algorithms like EDF. The same as PCP, SRP is also deadlock free, and the blocking under SRP is also at most the duration of one outer-most critical section. Under certain conditions, EDF+SRP is the optimal scheduling strategy for sporadic task models. The Deadline Dynamic Modification strategy (DDM) by Jeffay [7] is designed to work with EDF for the sporadic task model with a slight extension that each task may have multiple sequential execution-phases with different resource requirements in each period. Spuri and Stankovic [11] considered the scheduling of task systems with both shared resources and precedence constraints. They identify that EDF+SRP/PCP strategies are "quasi-normal", and work correctly even for task systems with precedence constraints.

III. Model and Background

A. The Digraph Real-Time Task Model

We first introduce the Digraph Real-Time (DRT) task model. A DRT task system τ consists of N DRT tasks $\{\tau_1, \cdots, \tau_N\}$. A DRT task τ_i is characterized by a directed graph $G(\tau_i)$. The set $\{J_i^1, \cdots, J_i^{N_i}\}$ of vertices in $G(\tau_i)$ represents the different types of jobs that can be released by task τ_i, where N_i is the number of job types in task τ_i (the number of vertices in $G(\tau_i)$). Each job type J_i^u is labeled a pair $\langle C_i^u, D_i^u \rangle$, where C_i^u is the worst-case execution time (WCET) and D_i^u is the relative deadline of jobs of this type. Each edge (J_i^u, J_i^v) is labeled with a non-negative integer $p(J_i^u, J_i^v)$ for the minimum job inter-released separation time. Further, we assume in this paper that the task system satisfies the *frame separation property*, by which all jobs' deadlines do not exceed the inter-release separation times: for all vertices J_i^u and their outgoing edges (J_i^u, J_i^v) we require $d_i^u \leq p(J_i^u, J_i^v)$. By the frame separation property we know that at any time instant each job type has at most one active *job instance*. For simplicity of presentation, the term *job* means either a *job type* or an instance of a job type, depending on the context. For example, Figure 1 shows a DRT task consisting of 4 jobs (the resource

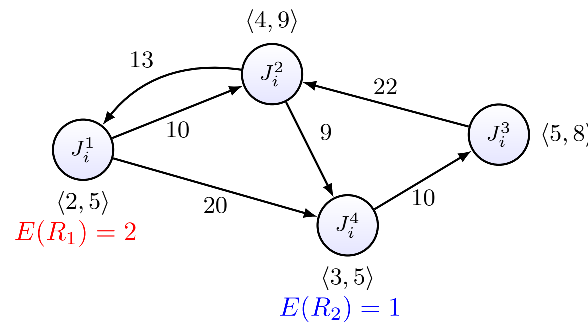


Fig. 1. A DRT task which uses shared resources.

access parameters, $E(R_1)$ and $E(R_2)$, in the figure will be introduced in Section III-B).

The semantics of a DRT task system is as follows. Each DRT task releases a potentially infinite sequence of jobs, by "walking" through the graph and releasing a job of the specified job type every time a vertex is visited. Before a new vertex is visited, it must wait for at least as long as the inter-release separation time labeled on the corresponding edge.

Feasibility analysis of DRT: Although the DRT task model offers very high expressiveness, the preemptive uniprocessor feasibility problem[2] for DRT is still tractable (of pseudo-polynomial complexity) [12]. Now we will very briefly review the key idea of the feasibility analysis of DRT and some meaningful concepts that will be used in this paper.

A crucial concept in the feasibility analysis of DRT task systems is the *demand bound function* $\mathsf{DBF}(\tau_i, l)$, which gives the maximum workload of jobs with both release time and deadline within any interval of length l, over all job sequences potentially generated by task τ_i. Intuitively, the demand bound function represents the worst-case workload that "must be executed" within the time interval to meet all deadlines. A sufficient and necessary condition for a DRT task system to be preemptive uniprocessor feasible is [12]:

$$\forall l > 0 : \sum_{\tau_i \in \tau} \mathsf{DBF}(\tau_i, l) \leq l \tag{1}$$

There are two main questions to be answered when using condition (1) to check feasibility: (i) How to compute $\mathsf{DBF}(\tau_i, l)$ for a given l? (ii) For which l do we need to compute $\mathsf{DBF}(\tau_i, l)$? In [12], techniques are presented for computing $\mathsf{DBF}(\tau_i, l)$ in pseudo-polynomial time. Also, a method to compute an upper bound L^{max} on the values of l that need to be checked is presented. For bounded-utilization task systems (i.e., with a utilization bounded by a constant smaller than 1), L^{max} is polynomial in the values of the task system representation. For these task systems, condition (1) can therefore be used to decide feasibility in pseudo-polynomial time. We use these facts for the schedulability tests later in this paper. Whenever DBF or L^{max} is referred to, they can be computed exactly as in [12].

[2]Determining the feasibility of a DRT task system equals to determining its schedulability under EDF scheduling.

B. Shared Resources

We extend the DRT model by adding non-preemptable, serially-reusable resources. A system can contain several such shared resources $R_1, R_2, ..., R_M$. To use a resource R_r, a job J_i^u first *locks* R_r, and then *holds* it for a certain amount of time, during which no other job can lock R_r. After having finished using R_r, the job will *unlock* it, and afterwards other jobs can lock and hold this resource. A job J_i^u could request resource R_r for multiple occasions during its execution. We use $E_i^u(R_r)$ to denote the maximal resource access duration of J_i^u to R_r, which means that for each time J_i^u requests R_r, the maximal *execution* time during which J_i^u is holding R_r is at most $E_i^u(R_r)$. In the example task system in Figure 1, J_i^1 may need resource R_1 and may execute for at most 2 time units while holding it; J_i^4 may need another resource R_2 and execute for at most 1 time unit while holding it (the task and job indices of $E_i^u(R_r)$ are omitted in the figure as they are clearly indicated by the positions). A job J_i^u could use more than one resource during its execution. We use $\mathcal{R}_i^u$ to denote the set of resources used by J_i^u. We do not assume any precise information about at what time a resource is requested, neither any particular order of the requests to these resources. We assume that the resource accesses are *non-nested*, i.e., a job can not lock a resource while holding another. Nested resource accesses can be handled using group locks [6].

C. The Problem with Branching: an Example

We use the example task system in Figure 2 to illustrate the new challenge that we face in the resource sharing problem for task graph systems.

This task system is a simple special case of DRT: each job only executes once. In τ_3, either J_3^2 or J_3^3 is released after the dummy job J_3^1. If we would know a priori which one of the two jobs will be released in the system, i.e., if the scheduler is *clairvoyant*, then all deadlines can be met:

- If we know that J_3^2 will be released, then we schedule the system by EDF scheduling. There are no resource conflicts in this case, and all deadlines can be met.
- If we know that J_3^3 will be released, then we schedule the system by non-preemptive EDF scheduling. This then makes sure that J_3^3 can be blocked by at most one of R_1 and R_2, and all deadlines can be met.

However, the information about which job will be released in the system is only revealed when the branching is actually taken at run-time, and a realistic scheduler does not have this knowledge beforehand. As we will show in the following, without the "clairvoyant" capability, no scheduler can successfully schedule this task set in all situations.

We focus on a particular scenario: all jobs will execute for their WCET and all resource accesses will last for their worst-case durations (both J_1^1 and J_2^1 will immediately lock the needed resource as soon as they start execution). J_1^1 is released first. J_2^1 is released immediately after J_1^1 locked R_1, and without loss of generality we let this time point be 0. Let the dummy job J_3^1 be released at time 6 and let it take the

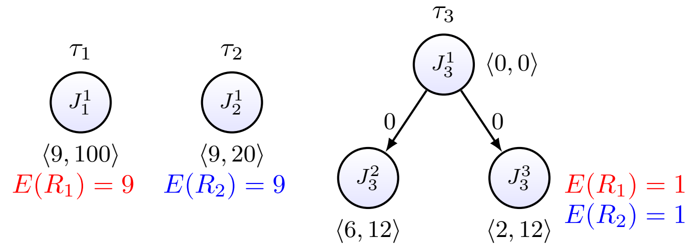


Fig. 2. A task system illustrating the new challenge due to "branching".

minimal inter-release separation 0 to release the next job, i.e., at time 6 either J_3^2 or J_3^3 will be released in the system. Now we look into the scheduling:

During the time interval $[0, 6)$, both J_1^1 and J_2^1 are active, the scheduler may let either of them to execute at any time instant in $[0, 6)$. We categorize all possibilities into two cases:

- J_2^1 has started execution in $[0, 6)$. In this case, the system will fail if J_3^3 is actually released: both R_1 and R_2 have been locked, so J_3^3 needs to wait until both of them are unlocked. The total workload that needs to be finished in the time interval $[0, 18]$ is $9 + 9 + 2 = 20$ which is larger than the interval length 18. A deadline miss in unavoidable.
- J_2^1 has *not* started execution in $[0, 6)$. In this case, the system will fail if J_3^2 is actually released: both J_2^1 and J_3^2 need to be finished before J_2^1's deadline at time 20, so the total work that needs to be done in the time interval $[6, 20]$ is $9 + 6 = 15$, which is larger than the interval length 14. Again a deadline miss is unavoidable.

In summary, there can be a deadline miss no matter how the scheduler behaves during $[0, 6)$.

As seen in this example, when scheduling DRT systems with shared resources, it indeed makes a difference whether the scheduler is clairvoyant or not. Recall that this difference does not exist for periodic or sporadic task systems with the same resource model: EDF+SRP is as powerful as clairvoyant scheduling there. Also for plain DRT systems (without shared resources) EDF is as powerful as any clairvoyant scheduling algorithm.

D. On-line feasibility and speedup factor

Clairvoyant schedulers are unrealistic. We are only interested in *on-line* (i.e., non-clairvoyant) scheduling strategies, which make scheduling decisions only based on task parameters and run-time information revealed in the past. In the remainder of this paper, when we refer to a scheduling strategy, we always mean an on-line scheduling strategy. We say that a task set τ is *feasible* if and only if τ is schedulable by some *on-line* algorithm.

Different metrics can be used to evaluate the worst-case performance guarantee of a real-time scheduling algorithm. The most widely used ones are *utilization bound* and *speedup factor*. It is easy to see that due to the non-preemptive resource access, any on-line algorithm may fail to schedule a task set with utilization arbitrarily close to 0, so *utilization bound* is not an appropriate metric for our problem.

In this paper we use *speedup factor* to quantitatively evaluate the "quality" of a scheduling algorithm. A scheduling algorithm $\mathcal{A}$ has a speedup factor of s if any task set that is feasible on a 1-speed machine, is also schedulable by $\mathcal{A}$ on an s-speed machine. Note that if the task system runs on a machine with speed s, then the total computation capacity provided by this machine in a time interval of length l is $s \cdot l$. So a job J_i^u can finish its worst-case execution requirement in time C_i^u/s, and its maximal resource usage time to resource R_r is $E_i^u(R_r)/s$.

IV. EDF+SRP FOR DRT TASK SYSTEMS

In this section, we study the performance of the well-known EDF+SRP scheduling strategy for the DRT task model extended with shared resources. In Section IV-A we first show that directly applying EDF+SRP to DRT leads to an unbounded speedup factor. Then in Section IV-B, by slightly revising EDF+SRP, we get a new scheduling strategy called EDF+saSRP, which has a tight speedup factor of 2.

A. EDF+SRP

It is easy to see that EDF+SRP can be directly applied to DRT task systems (only with a notation change that *job types* that may use a resource instead of *tasks* that may do so). Now we present the EDF+SRP scheduling rules[3]:

1) Each resource R_r is statically assigned a *level* ψ_r, which is set equal to the minimum relative deadline of any job in the system that may use it:
$$\psi_r = \min\{D_i^u | R_r \in \mathcal{R}_i^u\}$$
2) At runtime, each resource R_r has a *ceiling*:
$$\Psi_r = \begin{cases} \psi_r & \text{if } R_r \text{ is currently held by some job} \\ +\infty & \text{if } R_r \text{ is currently free} \end{cases}$$
3) The *system ceiling* at each time instant is the minimum among all the current resource ceilings.
4) The system is scheduled by EDF (with some deterministic priority order for two jobs with the same absolute deadline). In addition, a job is allowed to initially *start* execution only if its relative deadline is strictly smaller than the current system ceiling.

The speedup factor is unbounded: Unfortunately, directly applying the EDF+SRP strategy will lead to an unbounded speedup factor, as witnessed by the task set in Figure 3.

Clearly this task system is feasible: there is no resource conflict and each job can meet its deadline under EDF.

Now we schedule it by EDF+SRP, and consider a particular scenario: J_2^2 is released and starts execution at time 0, and immediately locks R_1. J_1^1 is released at time 1. By the EDF+SRP strategy, when J_2^2 locked R_1, the system ceiling was set to 1, so J_1^1 can not start execution until J_2^2 unlocks R_1.

[3]The rules are presented in a slightly different way from [2], [4], to keep the notation consistent with later algorithms in this paper. However, the scheduling behavior defined by the rules presented here is exactly the same as in [2], [4].

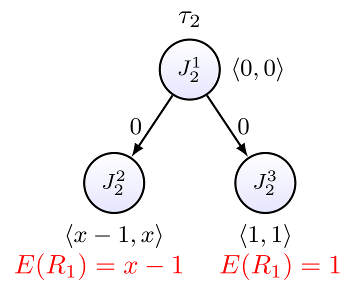


Fig. 3. A task system with the speedup factor of EDF+SRP is unbounded.

In the worst case, the total workload that needs to be executed between time 0 and J_1^1's deadline at time 3 is $x - 1 + 1 = x$. In order to make sure that J_1^1 meets its deadline, the machine speed should be at least $x/3$. As x approaches infinity, an infinitely fast machine is required for the task set to meet all deadlines under EDF+SRP.

B. EDF+saSRP

The reason for the unbounded speedup factor in the above example is that EDF+SRP ignores the knowledge that J_2^2 and J_2^3 are from the same task, so J_2^3 will never be released while J_2^2 is holding R_1 (remember that the tasks are assumed to have the frame separation property, so the same is true also for more complex tasks). This problem can easily be fixed by revising the first and second rules of EDF+SRP as follows:

1) Each resource-task pair (R_r, τ_i) has a static *self-exclusive level* $\psi_{r,i}$, which equals the minimum relative deadline of any job that may use it but is not in task τ_i.
$$\psi_{r,i} = \min\{D_j^u | R_r \in \mathcal{R}_j^u \wedge j \neq i\} \quad (2)$$
Note that there must be at least one job in the system satisfying $R_r \in \mathcal{R}_j^u \wedge j \neq i$. Otherwise there is no conflict on R_r, and R_r can be excluded from our consideration.
2) At runtime, each resource R_r has a *ceiling*:
$$\Psi_r = \begin{cases} \psi_{r,i} & \text{if } R_r \text{ is currently held by a job from } \tau_i \\ +\infty & \text{if } R_r \text{ is currently free} \end{cases}$$

The third and fourth rules are unchanged. We call this revised strategy *self-aware EDF+SRP* (EDF+saSRP for short). Intuitively, while a resource R_r is held by some job from task τ_i, EDF+saSRP will recognize that it is not possible for other jobs in τ_i to be released, and only get ceiling information from other tasks. EDF+SRP's properties, e.g., deadlock avoidance and multiple-blocking prevention, still hold for EDF+saSRP, since EDF+saSRP only safely excludes impossible system behaviors comparing with EDF+SRP.

Schedulability Analysis: We now present a sufficient schedulability test for EDF+saSRP on an s-speed machine.

Theorem 1. *A DRT task system τ with resources is schedulable by EDF+saSRP on an s-speed machine if both of the following two conditions are satisfied for all $l \in (0, L^{max}]$:*
$$\sum_{\tau_j \in \tau} \mathsf{DBF}(\tau_j, l) \leq s \cdot l \quad (3)$$

$$\forall \tau_i \in \tau : \left(B(\tau_i, l) + \sum_{i \neq j} \mathsf{DBF}(\tau_j, l) \leq s \cdot l \right) \quad (4)$$

where $B(\tau_i, l) = \max\{E_i^u(R_r) \mid D_i^u > l \wedge \psi_{r,i} \leq l\}$. If no job in τ_i satisfies $D_i^u > l \wedge \psi_{r,i} \leq l$, then $B(\tau_i, l) = 0$.

Proof Sketch: Condition (3) is for the case that only jobs with deadlines in the interval executes in it. Condition (4) is for the case when some job executes in the interval (while holding some resource), even though its deadline is out of this interval. In this case we enumerate each task τ_i that may cause resource blocking, and consider the longest blocking time, $B(\tau_i, l)$, that any job from that task could introduce in the interval. For blocking on some resource to occur for jobs that fit into the interval, the ceiling of that resource when held by τ_i must necessarily be smaller than the length of the interval. □

As we discussed above, the multiple-blocking prevention property of EDF+SRP still holds for EDF+saSRP, so it is enough to consider one blocking job in each interval. Note that this schedulability test is of pseudo-polynomial complexity, since for each l, DBF can be computed in pseudo-polynomial time and L^{max} is also pseudo-polynomially bounded, as we discussed in Section III-A.

The speedup factor is 2: By adding a simple self-awareness feature, the speedup factor is reduced from unbounded to 2, as shown in the following two lemmas:

Lemma 1. *Any task system that is feasible on a 1-speed machine, is also schedulable by EDF+saSRP on an s-speed machine with $s \geq 2$.*

Proof: Supposing τ is feasible on a 1-speed machine, but does not pass the schedulability test for EDF+saSRP above on an s-speed machine, we want to show that $s < 2$, by which the lemma is proved.

Since τ is feasible on the 1-speed machine, we claim that $\forall l \in (0, L^{max}]$, the following conditions must both be true:

$$\sum_{\tau_i \in \tau} \mathsf{DBF}(\tau_i, l) \leq l \quad (5)$$

$$B(\tau_i, l) \leq l \quad (6)$$

Condition (5) is the necessary condition for a DRT system to be feasible without considering the shared resources, so it must also be true here. To see that condition (6) is true, we recall that $B(\tau_i, l)$ is the longest access duration, $E_i^u(R_r)$, of some job J_i^u to some resource R_r, such that there exists a job J_j^w in another task that needs resource R_r and has a relative deadline of at most l. Since J_i^u and J_j^w are from different tasks, J_j^w may be released right after J_i^u locked R_r. In this case, if J_j^w can meet its deadline, $E_i^u(R_r)$ must be no larger than l, i.e., $B(\tau_i, l) \leq l$ must be true.

Then we consider the s-speed machine. We know by assumption that there must exist some l such that at least one of (3) or (4) is violated. If (3) is violated, then by (5) we have $s < 1 < 2$.

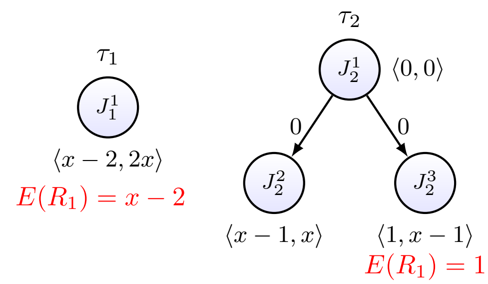


Fig. 4. An example task system illustrating that the speedup factor 2 for EDF+saSRP is tight.

If (4) is violated, then there exists some $l \in (0, L^{max}]$ and some task τ_i such that

$$B(\tau_i, l) + \sum_{i \neq j} \mathsf{DBF}(\tau_j, l) > s \cdot l \quad (7)$$

Since $\sum_{i \neq j} \mathsf{DBF}(\tau_j, l)$ is bounded by $\sum_{\tau_i \in \tau} \mathsf{DBF}(\tau_i, l)$, we get by (5) and (6) that $l + l > s \cdot l$,, i.e., $s < 2$. ■

Actually, the speedup factor 2 derived above is tight for EDF+saSRP, as shown in the following lemma:

Lemma 2. *There exists a task system which is feasible on a 1-speed machine, but not schedulable by EDF+saSRP on any s-speed machine with $s < 2$.*

Proof: Consider the task system in Figure 4 where x is a positive integer. The task system is feasible on a 1-speed machine: we schedule it by EDF plus the rule that J_1^1 and J_2^3 never preempt each other, then all deadlines can be met.

Now we consider the s-speed machine. We assume $s = 2-\xi$ where ξ is positive. We consider a particular scenario: J_2^2 is released just after J_1^1 locked resource R_1. We consider the time interval between J_2^2's release time and absolute deadline, which is of length x. According to the EDF+saSRP rule, after J_1^1 locked R_1, the system ceiling is $x-1$ (J_2^3's relative deadline), which is smaller than J_2^2's relative deadline x, so J_2^2 can not preempt J_1^1, and waits until J_1^1 is finished. So a necessary condition for the task set to be schedulable is that the worst-case access duration of J_1^1 to R_1 plus J_2^2's worst-case execution time is smaller than the total capacity of the interval of length x, i.e.,

$$x - 2 + x - 1 \leq (2 - \xi) \cdot x$$
$$\xi \cdot x \leq 3.$$

So we know that if $\xi \cdot x > 3$, τ is not schedulable by EDF+saSRP. In other words, for any $s = 2 - \xi < 2$, we can find an x satisfying $\xi \cdot x > 3$ to construct a task set as in Figure 4, which is feasible on the 1-speed machine but not schedulable by EDF+saSRP on the s-speed machine. ■

V. ACP: Absolute-time Ceiling Protocol

In this section we present a new protocol, the *Absolute-time Ceiling Protocol* (ACP) to better handle the resource sharing in task graph systems. The new scheduling strategy EDF+ACP (EDF scheduling with the Absolute-time Ceiling Protocol) has a speedup factor no larger than $\frac{1+\sqrt{5}}{2} \approx 1.618$, which is the famous constant commonly known as the *golden ratio*.

A. EDF+ACP

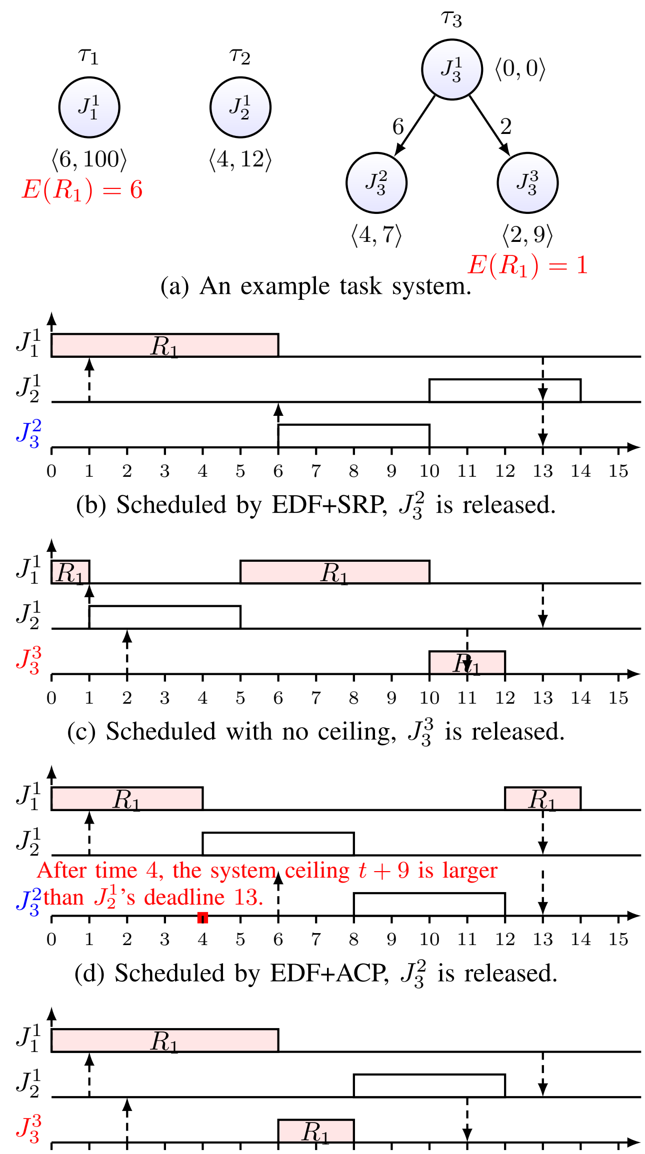
As suggested by its name, the "ceiling" concept in ACP is about absolute time (recall that the ceiling in SRP is about relative time). In the following we will define the scheduling rules of EDF+ACP. For simplicity of presentation, the data structures are defined in the form of functions with respect to time t. Later, in Section V-D, we will introduce how to implement EDF+ACP such that these data structures only need to be updated at certain time points, but not continuously at each time point.

1) At each time instant t, each resource R_r has a *ceiling*:
$$\Psi_r(t) = \begin{cases} t + \psi_{r,i} & \text{if } R_r \text{ is held by } \tau_i \text{ at } t \\ +\infty & \text{if } R_r \text{ is free at } t \end{cases}$$
where $\psi_{r,i}$ is the *self-exclusive level* defined in (2).
2) At each time instant t, each resource R_r has a *request deadline*:
$$\Pi_r(t) = \begin{cases} \mathsf{earliest\_d}(R_r) & \text{if } R_r \text{ is held by a job at } t \\ +\infty & \text{if } R_r \text{ is free at } t \end{cases}$$
where $\mathsf{earliest\_d}(R_r)$ is the earliest *absolute* deadline among all the *active* jobs who may need R_r.
3) At each time instant t, the *system ceiling* is
$$\Upsilon(t) = \min\left\{ \min(\Psi_r(t), \Pi_r(t)) \mid R_r \text{ is a resource} \right\}.$$
4) The system is scheduled by EDF (with some deterministic priority order for two jobs with the same absolute deadline). In addition, a job can initially *start* execution only if its *absolute* deadline is strictly smaller than the current system ceiling.

We use the example task system in Figure 5-(a) to illustrate the crucial difference between EDF+ACP and EDF+SRP (EDF+saSRP), and disclose the main idea behind EDF+ACP. We assume both J_1^1 and J_3^1 are released at time 0, and J_2^1 is released at time 1. By the inter-release separation constraints in τ_3, J_3^2 can at earliest be released at 6 and J_3^3 earliest at 2.

Figure 5-(b) shows how would the task set be scheduled by EDF+SRP if J_3^2 is released at time 6. At time 0, J_1^1 locked resource R_1, and the system ceiling is assigned by ψ_1, which is equal to J_3^3's relative deadline 9. At time 1, J_2^1 is released. Since its relative deadline 12 is larger than the current system ceiling 9, it can not start and preempt J_1^1. At time 6, J_1^1 is finished and J_3^2 is released. The total amount of work that must be finished in $[6, 13]$ is now 8, so there will be a deadline miss no later than at time 13.

We can see the main cause for the deadline miss under EDF+SRP in this example: When J_1^1 locked R_1, EDF+SRP used a ceiling of 9 to prevent potential priority-inversion blocking to J_3^3 who also needs R_1, and may sometimes be released right after the resource was locked. This system ceiling of 9 makes perfect sense for sporadic task systems since the release of such a job exactly forms the worst case: if EDF+SRP can not avoid the deadline miss in this scenario,



(a) An example task system.

(b) Scheduled by EDF+SRP, J_3^2 is released.

(c) Scheduled with no ceiling, J_3^3 is released.

(d) Scheduled by EDF+ACP, J_3^2 is released.

(e) Scheduled by EDF+ACP, J_3^3 is released.

Fig. 5. Illustrating the difference between EDF+SRP and EDF+ACP.

then the task system is not feasible anyway. However, in DRT task systems this is not necessarily the worst case due to the branching structure of τ_3. In this example, it might be J_3^2, instead of J_3^3, that is released in the system, in which case preventing priority-inversion blocking of J_3^3 makes no sense. The price paid for this meaningless system ceiling is to impose on J_2^1 additional blocking from J_1^1, which ultimately causes a missed deadline.

On the other hand, if the scheduler ignores the ceiling at time 1 and allows J_2^1 to preempt J_1^1, as shown in Figure 5-(c), a deadline also might be missed: If later J_3^3 is released, it will suffer the priority-inversion blocking from J_2^1 and miss its deadline. So we can see that due to the "branching" nature of τ_3, it is impossible for the scheduler to make a "correct decision" at time 1 about whether J_1^1 or J_2^1 should run to completion first.

The main idea behind ACP is to always make decisions to safely prevent any potential priority-inversion blocking, and correct a "wrong" decision as soon as it is clear that a predicted priority-inversion blocking no longer is possible.

Figure 5-(d) shows how the task set would be scheduled by EDF+ACP in the case where J_3^2 is released at time 6. Since R_1 is held by J_1^1, during the time interval $[1,4]$ the system ceiling $(t+9)$ is no larger than J_2^1's absolute deadline 13, so J_2^1 can not start. This corresponds to the fact that J_3^3, who needs R_1, may be released at some time point before 4 such that J_3^3's absolute deadline is smaller than J_2^1's. So the system ceiling during the time interval $[1,4]$ prevents the potential priority-inversion blocking from J_2^1 to J_3^3. After time 4, the system ceiling $(t+9)$ is larger than J_2^1's absolute deadline 13, so J_2^1 preempts J_1^1 and starts execution. This corresponds to the fact that after time 4, even if J_3^3 is released, its deadline is later than J_2^1's, so J_2^1 should not be blocked any longer. At time 6, τ_3 releases J_3^2, who does not need R_1 but has more workload than J_3^3. However, thanks to the "correction" at time 4, both J_3^2 and J_2^1 can meet their deadlines.

In the other case, where J_3^3 is released at time 2, the system is also schedulable by EDF+ACP as shown in Figure 5-(e). During the interval $[1,2]$, the system ceiling is no larger than J_2^1's absolute deadline, so J_2^1 is blocked. At time 2, J_3^3 is released, and R_1's *request deadline* $\Pi_1(t)$ is set to equal J_3^3's absolute deadline 11, so J_3^3 has to wait until J_1^1 unlocks R_1. One can see that the *request deadline* $\Pi_r(t)$ is used to capture this direct resource conflict, and enforces the rule that a job can start execution only if all its needed resources are free. Since the system ceiling prevents the priority-inversion blocking from J_2^1 to J_3^3, J_3^3 can meet its deadline. Although the release of J_3^3 prevents J_2^1 from preempting J_1^1 as it did in the example 5-(d), J_3^3 itself has a low workload, so J_2^1 can still meet its deadline.

Correctness: One can see that there are no explicit semantics for resource conflict resolving in the EDF+ACP rules introduced above, and the jobs are scheduled only based on priorities and an extra starting control mechanism based on the system ceiling. Now we show that EDF+ACP can correctly resolve resource conflicts, as stated in the following lemma.

Lemma 3. *A job which has started execution will not be blocked on any resource.*

Proof: We prove by contradiction. Suppose a job J_i^u starts executing at time t_u, and at some point during its execution a resource R_r that J_i^u may need is being held by some other job J_j^w. We know that J_i^u has higher priority than J_j^w (because J_i^u is the one executing), so J_j^w can not execute after J_i^u starts executing and before it is finished. By this we know that R_r must have been held by J_j^w already at t_u, so we know $\Upsilon(t_u) \leq \Pi_r(t_u)$. Since J_i^u is active at t_u, $\Pi_r(t_u)$ is at most J_i^u's absolute deadline. Therefore we know $\Upsilon(t_u)$ is no larger than J_i^u's absolute deadline, which contradicts the assumption that J_i^u starts executing at t_u. ■

We can also see that the EDF+ACP strategy is work-conserving: if there are jobs already started in the system, then the job with the highest priority will execute. If there are no jobs already started in the system, which implies no resource is in use, then the system ceiling is $+\infty$, and the highest-priority one among all jobs that have not started will execute. By the above reasoning, we have the following lemma:

Lemma 4. *There must be a job executing whenever there are pending jobs in the system.*

B. Schedulability Analysis

We first introduce the framework we use to derive our *sufficient* schedulability test for EDF+ACP. We assume a task system τ is not schedulable by EDF+ACP on an s-speed machine, and let t_d be the earliest time instant when some job misses its deadline. Let t_s be the earliest time instant before t_d such that at each time instant in the *busy period* $[t_s, t_d]$, there is at least one active job with deadline no later than t_d. Let $l = t_d - t_s$.

We will derive an upper bound on the total workload $W(l)$ that EDF+ACP executes in $[t_s, t_d]$. Since some job does not meet its deadline at t_d and the scheduling algorithm is work-conserving (by Lemma 4), this upper bound must be larger than the total capacity of $[t_s, t_d]$, which is $s \cdot l$. By negating the above statement, we know that if for all $l \in (0, L^{max}]$ it is true that $W(l) \leq s \cdot l$, then there does not exist a busy period causing the deadline miss, which implies that τ is schedulable by EDF+ACP on an s-speed machine. We first introduce two extra notations.

Definition 1 (Pressing jobs and blocking jobs)**.** *The jobs with both release time and absolute deadline in $[t_s, t_d]$ are called* pressing jobs. *The jobs with absolute deadline later than t_d but that executes in $[t_s, t_d]$ are called* blocking jobs.

By the definition of the busy period $[t_s, t_d]$, a job that executes in $[t_s, t_d]$ is either a pressing job or a blocking job.

In the following we will derive upper bounds for $W(l)$. First consider a simple case: *Only* pressing jobs execute in $[t_s, t_d]$. In this case, the workload of each task τ_i is bounded by its demand bound function $\mathsf{DBF}(\tau_i, l)$, so we have:

$$W(l) \leq \sum_{\tau_i \in \tau} \mathsf{DBF}(\tau_i, l). \quad (8)$$

In the following we consider the difficult case: There are also blocking jobs executing in $[t_s, t_d]$. In this case, $W(l)$ consists of workload from both blocking and pressing jobs. We start with bounding the workload of the blocking jobs.

Workload bounds for blocking jobs: Just as EDF+SRP, the new protocol EDF+ACP can also prevent multiple priority-inversion blocking, as stated in the following lemma:

Lemma 5. *There is at most one blocking job that executes in $[t_s, t_d]$, and the blocking duration is at most the worst-case resource access duration of this blocking job to some resource.*

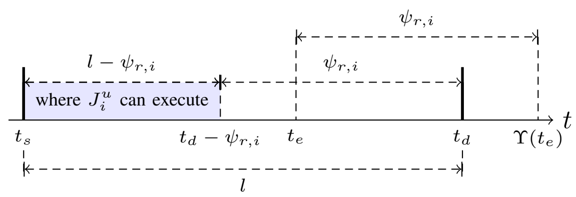


Fig. 6. Intuition of Lemma 6.

Proof: If a blocking job executes at $t_e \in [t_s, t_d]$, then there must be some resource R_r held at t_e that causes $\Upsilon(t_e) \leq t_d$, or otherwise some pressing job would execute at t_e instead. Also, R_r can not be held by a pressing job, because then that pressing job would execute instead. Consequently, R_r, a resource that gives rise to a value of $\Upsilon(t_e) \leq t_d$, must have been locked before t_s. Let $t_b < t_s$ be the earliest time point where such a resource R_r was last locked, by job J_i^u say. Now, J_i^u still holds R_r at t_e, so no jobs with deadline after t_d can start in the interval $[t_b, t_e]$ (because of the ceiling of R_r). Note that, according to the EDF+ACP rules, the value contributed by resource R_r to the system ceiling will never decrease, as long as R_r is being held. This is because the resource ceiling $t + \psi_{r,i}$ provides a safe lower bound of the deadline of any future job which may need R_r.

Also, no job that was released before t_b and has a deadline later than t_d can execute in $[t_b, t_e]$, except J_i^u. This is because at t_b, J_i^u must have had the highest priority among the started jobs, and it has not yet finished by t_e. It follows then that no other job than J_i^u, with deadline later than t_d, can execute in $[t_b, t_e]$, for any $t_e \in [t_s, t_d]$. It also follows that while J_i^u executes in $[t_s, t_d]$, it is holding R_r. ■

In the following, we use J_i^u and R_r to denote the job and the resource contributing to the blocking, and we use $\Delta(J_i^u, R_r)$ to denote the length of the blocking. By Lemma 5 we get an upper bound for $\Delta(J_i^u, R_r)$:

$$\Delta(J_i^u, R_r) \leq E_i^u(R_r) \tag{9}$$

Apart from the above upper bound, we will give another upper bound of $\Delta(J_i^u, R_r)$, which is only applicable under the condition that no pressing job needs R_r:

Lemma 6. *If no pressing job needs R_r, then*

$$\Delta(J_i^u, R_r) \leq (l - \psi_{r,i}) \cdot s \tag{10}$$

Proof: Let J_i^u be the blocking job that executes in $[t_s, t_d]$, and let R_r be the resource that it holds. We know by Lemma 5 that there is at most one such job and resource. Since no pressing jobs need R_r, we know that $\Pi_r(t_e) > t_d$ for all $t_e \in [t_s, t_d]$. It follows then that at any $t_e > t_d - \psi_{r,i}$, the system ceiling $\Upsilon(t_e) > t_d$. The pressing jobs can then be blocked only during the interval $[t_s, t_d - \psi_{r,i}]$, which is of length $l - \psi_{r,i}$. ■

An illustration of the intuition behind Lemma 6 is shown in Figure 6.

Workload bounds for pressing jobs: Lemma 6 provides an extra workload bound of blocking jobs for the special case that no pressing job needs R_r. Correspondingly, we will bound the workload of pressing jobs of a task τ_i in two cases, depending on whether it contains any job which needs R_r. We define a new demand bound function for each case:

- $\mathsf{DBF}_N(\tau_i, R_r, l)$ is the maximum workload of jobs with both release time and deadline within any interval of length l, over all job sequences generated by τ_i in which *none of the jobs needs R_r*.
- $\mathsf{DBF}_Y(\tau_i, R_r, l)$ is the maximum workload of jobs with both release time and deadline within any interval of length l, over all job sequences generated by τ_i in which *at least one job needs R_r*.

As an example, we calculate $\mathsf{DBF}_N(\tau_i, R_2, 20)$ and $\mathsf{DBF}_Y(\tau_i, R_2, 20)$ for the task in Figure 1.

- $\mathsf{DBF}_N(\tau_i, R_2, 20)$: We exclude job J_i^4 who needs R_2, as well as the edges connecting it. Among the remaining nodes and paths, we can see that the sequence $\{J_i^1, J_i^2\}$ (or $\{J_i^2, J_i^1\}$) leads to the maximal $\mathsf{DBF}_N(\tau_i, R_2, 20) = 6$.
- $\mathsf{DBF}_Y(\tau_i, R_2, 20)$: The job sequences must contain J_i^4, and we observe that the sequence $\{J_i^4, J_i^3\}$ leads to the maximal $\mathsf{DBF}_Y(\tau_i, R_2, 20) = 8$.

$\mathsf{DBF}_N(\tau_i, R_r, l)$ and $\mathsf{DBF}_Y(\tau_i, R_r, l)$ can be easily computed by slightly modifying the DBF computation algorithm in [12], using an extra bit to record whether a job that needs R_r has been visited during the graph exploration. The complexity of computing $\mathsf{DBF}_N(\tau_i, R_r, l)$ and $\mathsf{DBF}_Y(\tau_i, R_r, l)$ is still pseudo-polynomial.

Bounding the total workload: By now, we have derived workload bounds for both blocking and pressing jobs. Now we will combine them to get upper bounds of the total workload $W(l)$. This is also done in two cases, depending on whether there exists a pressing job that needs R_r or not:

In the first case, no pressing job needs R_r. In this case, the blocking job's workload is bounded by both $E_i^u(R_r)$ (Lemma 5) and $s(l - \psi_{r,i})$ (Lemma 6). The workload of the pressing jobs from each task $\tau_j \in \tau \setminus \{\tau_i\}$ is bounded by $\mathsf{DBF}_N(\tau_j, R_r, l)$, which only counts the paths that do not include any jobs which need R_r. If J_i^u is the blocking job and R_r the resource causing the blocking, then the total workload in $[t_s, t_d]$ (on an s-speed machine) is bounded by:

$$\mathsf{UB}_N(J_i^u, R_r, l) = \min\big(E_i^u(R_r), s \cdot (l - \psi_{r,i})\big) + \sum_{j \neq i} \mathsf{DBF}_N(\tau_j, R_r, l)$$

By enumerating all the possible candidates for J_i^u (a job with relative deadline larger than l) and R_r (a resource needed by J_i^u), and selecting the maximum, we get an upper bound:

$$W(l) \leq \max\{\mathsf{UB}_N(J_i^u, R_r, l) \mid D_i^u > l \wedge R_r \in \mathcal{R}_i^u\} \tag{11}$$

In the second case, at least one pressing job needs R_r. In this case, we know there is at least one task τ_j that includes a pressing job which needs R_r in its workload in the busy period. That task's workload is bounded by $\mathsf{DBF}_Y(\tau_j, R_r, l)$. Other tasks τ_k may or may not include a pressing job which needs R_r, so the workload of each is bounded by the normal demand bound function $\mathsf{DBF}(\tau_k, l)$. If J_i^u is the blocking job and R_r the resource it holds, then we enumerate all possibilities of choosing τ_j, and the maximal one is an upper bound of the workload (on an s-speed machine):

$$\mathsf{UB}_Y(J_i^u, R_r, l) = \min(E_i^u(R_r), s \cdot l) + \max_{\substack{j \neq i \ \wedge \\ \mathsf{DBF}_Y(\tau_j, R_r, l) \neq 0}} \left\{ \mathsf{DBF}_Y(\tau_j, R_r, l) + \sum_{\substack{k \neq i \\ k \neq j}} \mathsf{DBF}(\tau_k, l) \right\}$$

Note that there may not exist $\tau_j \in \tau \setminus \{\tau_i\}$ such that $\mathsf{DBF}_Y(\tau_j, R_r, l) \neq 0$, which means that it is not possible for any task to include a pressing job which needs R_r in the busy period. In this case, the second item in the RHS of the above definition is 0, and $\mathsf{UB}_Y(J_i^u, l, R_r)$ is bounded by $s \cdot l$.

Again, by enumerating all the possible candidates for J_i^u and R_r, and selecting the maximum, we get an upper bound of $W(l)$ for this case:

$$W(l) \leq \max\{\mathsf{UB}_Y(J_i^u, R_r, l) \mid D_i^u > l \wedge R_r \in \mathcal{R}_i^u\} \quad (12)$$

Schedulability test condition: We have derived upper bounds of $W(l)$ for all cases:

- If no resource blocking occurs in the busy period, $W(l)$ is bounded by (8).
- If there is resource blocking in the busy period, there are two possible cases:
  - If no pressing job needs the blocking resource, $W(l)$ is bounded by (11).
  - If at least one pressing job needs the blocking resource, $W(l)$ is bounded by (12).

We put them together to get a sufficient schedulability test:

Theorem 2. *A task set τ is schedulable by EDF+ACP on an s-speed machine if for all $l \in (0, L^{max}]$, all of the following three conditions are satisfied:*

$$\sum_{\tau_j \in \tau} \mathsf{DBF}(\tau_j, l) \leq s \cdot l \quad (13)$$

$$\max\{\mathsf{UB}_Y(J_i^u, R_r, l) \mid D_i^u > l \wedge R_r \in \mathcal{R}_i^u\} \leq s \cdot l \quad (14)$$

$$\max\{\mathsf{UB}_N(J_i^u, R_r, l) \mid D_i^u > l \wedge R_r \in \mathcal{R}_i^u\} \leq s \cdot l \quad (15)$$

Given an l, DBF can be computed in pseudo-polynomial time [12], and so can DBF_N and DBF_Y as we discussed above. So for each l the complexity of verifying all the three conditions in Theorem (2) is pseudo-polynomial. Recall that L^{max} is a pseudo-polynomial upper bound on the values of l that need to be checked. The overall complexity of the schedulability test in Theorem (2) is therefore pseudo-polynomial.

C. Speedup factor

In this section, we will show that the new EDF+ACP scheduling strategy has a speedup factor of $\frac{\sqrt{5}+1}{2}$. We start from a necessary condition for a task system to be feasible on a 1-speed machine.

Lemma 7. *If a task system τ is feasible on a 1-speed machine, then for all $l \in (0, L^{max}]$, condition (13) and (14) must be true with $s = 1$ and the following condition must also be true:*

$$\forall (J_i^u, R_r) : E_i^u(R_r) \leq \psi_{r,i} \quad (16)$$

Proof: Condition (13) is known to be a necessary condition for the task system to be feasible without considering any shared resources [12]. So it must be true also here.

We indirectly prove that condition (14) is true. Assume there exists a configuration (l, J_i^u, R_r, τ_j) to violate (14), such that $D_i^u > l$ and $R_r \in \mathcal{R}_i^u$ and τ_j can generate workload $\mathsf{DBF}_Y(\tau_j, R_r, l)$, which includes at least one job who needs R_r, in a time interval of length l. We consider a particular scenario: right after J_i^u locked R_r, τ_j releases the job sequence of workload $\mathsf{DBF}_Y(\tau_j, R_r, l)$ with minimal inter-release separation, and all the other jobs release the job sequence of their worst-case workload in l, which cause a total workload of $\sum_{k \neq i, k \neq j} \mathsf{DBF}(\tau_k, l)$. It is clear that the worst-case total workload that needs to be executed in the time interval includes $\mathsf{DBF}_Y(\tau_j, R_r, l)$ and $\sum_{k \neq i, k \neq j} \mathsf{DBF}(\tau_k, l)$. It also includes the worst-case duration for J_i^u to access R_r, since the job who needs R_r in $\mathsf{DBF}_Y(\tau_j, R_r, l)$ can not start until J_i^u unlocks R_r. Since (14) is violated, we know the workload in this interval exceeds the total capacity, so the deadline miss is unavoidable under any on-line scheduling algorithm. So (14) is necessary for τ to be feasible.

To see that condition (16) is also true, we recall that $\psi_{r,i}$ is the minimal relative deadline of some job J_j^w which needs R_r but is not in τ_i. At run time, it is possible that J_j^w is released right after J_i^u locked R_r. In this case, if (16) is not true, clearly no on-line scheduling algorithm can ensure that J_j^w meets its deadline. So we know (16) is also necessary for the task system to be feasible. ■

Now we establish the speedup factor for EDF+ACP.

Theorem 3. *Any task system τ that is feasible on a 1-speed machine, is deemed to be schedulable by EDF+ACP according to Theorem 2 on an s-speed machine with $s \geq \frac{\sqrt{5}+1}{2}$.*

Proof: We prove by contradiction. We assume a task system is feasible on a 1-speed machine, and is not guaranteed to be schedulable by the sufficient schedulability test in Theorem 2. We will prove that it must then hold that $s < \frac{\sqrt{5}+1}{2}$.

We consider the s-speed machine. We know that at least one of the conditions (13), (14) or (15) is violated.

Since τ is feasible on a 1-speed machine, we know that if (13) or (14) is violated, then $s < 1 < \frac{\sqrt{5}+1}{2}$.

In the following we consider the remaining case that (15) is violated, i.e., there exists (J_i^u, R_r, l) such that

$$\mathsf{UB}_N(J_i^u, R_r, l) > s \cdot l \quad (17)$$

For simplicity, we let $A = \min\left(E_i^u(R_r), s(l - \psi_{r,i})\right)$ and $B = \sum_{j \neq i} \mathsf{DBF}_N(\tau_j, R_r, l)$ so (17) can be rewritten as:

$$A + B > s \cdot l \tag{18}$$

Now we will find upper bounds for A and B, respectively. We first consider A. Let $x = \psi_{r,i}$. By definition, it is clear that

$$A \leq s(l - x) \tag{19}$$

$$A \leq E_i^u(R_r). \tag{20}$$

Since τ is feasible on a 1-speed machine, by Lemma 7 we know that $E_i^u(R_r) \leq \psi_{r,i}$, i.e., $E_i^u(R_r) \leq x$, and so (20) can be rewritten as

$$A \leq x \tag{21}$$

We observe that the RHS of (19) is a decreasing function with respect to x, while the RHS of (21) is an increasing function with respect to x. So A is bounded by the value of the point where these two functions intersect. We let $s(l - x) = x$ and get $x = \frac{s \cdot l}{s+1}$. By the reasoning above we have

$$A \leq \frac{s \cdot l}{s + 1}. \tag{22}$$

Now we consider B. Since $\mathsf{DBF}_N(\tau_j, R_r, l) \leq \mathsf{DBF}(\tau_j, l)$, we get $B \leq \sum_{\tau_j \in \tau} \mathsf{DBF}(\tau_j, l)$. Since τ is feasible on a 1-speed machine, by Lemma 7 we know that

$$B \leq l. \tag{23}$$

By applying (22) and (23) to (18), we get $\frac{s \cdot l}{s+1} + l > s \cdot l$, by which we have $s < \frac{\sqrt{5}+1}{2}$. ■

D. Implementation and overhead

By definition, the system ceiling $\Upsilon(t)$ is a function with respect to time t. However, the EDF+ACP strategy can be implemented *without* updating $\Upsilon(t)$ at each time instant, and the number of extra scheduler invocations for maintaining $\Upsilon(t)$ can be very well bounded. The crucial observation is that the system ceiling is important to check only at the time instants where some job may get an absolute deadline smaller than the system ceiling.

Firstly, we may have to check (and update) the system ceiling at the time points where the scheduler would normally be invoked anyway: when a job is released or finished and when the unlocking of a resource enables some new job to start. Updating the system ceiling at those points require no extra scheduler invocation.

Secondly, we may have to invoke the scheduler at some time point when the system ceiling has grown larger than the deadline of some job that was not allowed to start due to the ceiling. Let $d(t)$ be the earliest deadline of all active jobs at time t. If the job that has deadline $d(t)$ has not yet been allowed to start (because $\Upsilon(t) < d(t)$), we may have to invoke the scheduler at a time point t' where the ceiling has grown so that $\Upsilon(t') = d(t')$.

When is this time point t'? If $\min_r\{\Pi_r(t)\} \leq d(t)$, then nothing except one of the normal scheduling events described above can lead to a t' where $\Upsilon(t') = d(t')$, so we can safely skip invoking the scheduler until one of these events occur. However, if $\min_r\{\Pi_r(t)\} > d(t)$ (which means that $\min_r\{\Psi_r(t)\} < d(t)$), the passing of time can bring us to such a t'. If none of the normal scheduling events occur before, or any new resource is locked, that t' will be exactly at $t + d(t) - \min_r\{\Psi_r(t)\}$. We can set a timer to fire at this time point. If one of those events occurs before t', we can simply remove or recalculate the timer as needed. When a timer eventually fires, we know that the job with deadline $d(t)$ can start, and consequently that job will never be blocked again. Clearly, the firing of the timer will happen then at most once per job, and therefore the number of extra scheduler invocations will be at most once per job.

VI. Conclusions

In this paper we studied the non-nested resource sharing problem for real-time task graph systems. Due to the branching structures in such graphs, this problem is fundamentally different from, and significantly more difficult than, for the simple periodic and sporadic models. We first considered applying EDF+SRP, which is the optimal scheduling strategy for simple sporadic models, to task graph systems. We showed that a direct application of EDF+SRP leads to an unbounded speedup factor, and it can be reduced to 2 by a slight modification of the EDF+SRP rules. We also proposed ACP, a novel resource sharing protocol,and the scheduling strategy EDF+ACP, achieves a speedup factor of $\frac{\sqrt{5}+1}{2} \approx 1.618$, which is the well-known constant *golden ratio*. As future work, we seek to design scheduling strategies with smaller speedup factors. The potential is to trade scheduler complexity for a more precise resource blocking prediction.

References

[1] M. Anand, A. Easwaran, S. Fischmeister, and I. Lee. Compositional feasibility analysis of conditional real-time task models. In *ISORC*, 2008.

[2] T. P. Baker. Stack-based scheduling of realtime processes. In *Real-Time Systems*, 1991.

[3] S. Baruah. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks. In *Real-Time Systems*, 2003.

[4] S. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *RTSS*, 2006.

[5] S. Baruah. The non-cyclic recurring real-time task model. In *RTSS*, 2010.

[6] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.

[7] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *RTSS*, 1992.

[8] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. In *PhD thesis, Massachusetts Institute of Technology*, 1983.

[9] N. T. Moyo, E. Nicollet, F. Lafaye, and C. Moy. On schedulability analysis of non-cyclic generalized multiframe tasks. In *ECRTS*, 2010.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers*, 1990.

[11] M. Spuri and J. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. In *IEEE Trans. Computers*, 1994.

[12] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The Digraph Real-Time Task Model. In *RTAS 2011*.