GEDF Tardiness: Open Problems Involving Uniform Multiprocessors and Affinity Masks Resolved

Stephen Tang
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
sytang@cs.unc.edu

Sergey Voronov
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
rdkl@cs.unc.edu

James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA
anderson@cs.unc.edu

Abstract

Prior work has shown that the global earliest-deadline-first (GEDF) scheduler is *soft real-time (SRT)-optimal* for sporadic task systems in a variety of contexts, meaning that bounded deadline tardiness can be guaranteed under it for any task system that does not cause platform overutilization. However, one particularly compelling context has remained elusive: multiprocessor platforms in which tasks have *affinity masks* that determine the processors where they may execute. Actual GEDF implementations, such as the `SCHED_DEADLINE` class in Linux, have dealt with this unresolved question by foregoing SRT guarantees once affinity masks are set. This unresolved question, as it pertains to `SCHED_DEADLINE`, was included by Peter Zijlstra in a list of important open problems affecting Linux in his keynote talk at ECRTS 2017. In this paper, this question is resolved along with another open problem that at first blush seems unrelated but actually is. Specifically, both problems are closed by establishing two results. First, a proof strategy used previously to establish GEDF tardiness bounds that are exponential in size on heterogeneous uniform multiprocessors is generalized to show that polynomial bounds exist on a wider class of platforms. Second, both uniform multiprocessors and identical multiprocessors with affinities are shown to be within this class. These results yield the first polynomial GEDF tardiness bounds for the uniform case and the first such bounds of any kind for the identical-with-affinities case.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases scheduling theory, multicore, processor affinity masks, GEDF, uniform multiprocessors

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.6

Related Version A full version of the paper is available at `http://www.cs.unc.edu/~anderson/papers.html`.

Funding Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

1 Introduction

The global earliest-deadline-first (GEDF) scheduler has received considerable prior attention. One attractive property of GEDF is that its use ensures guaranteed bounded deadline tardiness on certain multiprocessor platform types for any sporadic task system that does not cause platform overutilization [5, 8]. In this sense, GEDF is considered an *optimal soft real-time (SRT)* scheduler. This SRT-optimality is significant enough to warrant mention in

the documentation of `SCHED_DEADLINE` [2], Linux's implementation of GEDF. A practical use by `SCHED_DEADLINE` of this optimality is the ability to perform online admission control.

A major caveat of GEDF's SRT-optimality is that it was originally proven only for processors with identical speeds [5]. Processor models that break this assumption, called *heterogeneous multiprocessors*, tend to fundamentally break the proof techniques applied to the identical model. Only one prior work [8] has succeeded in extending the SRT-optimality of GEDF to a multiprocessor model with heterogeneity, namely the *uniform model*, which allows speed differences among processors. This work required years of thought and several new proof techniques but only yielded tardiness bounds that are exponential in size.

Affinity masks. Heterogeneity is also introduced by the usage of *affinity masks*. A task's affinity mask (typically a bit vector) indicates the processors upon which it may execute. Affinity masks are useful for preventing excessive task migrations and can be used to take better advantage of cache hierarchies. Also, global, clustered, and partitioned scheduling can all be expressed using affinity masks.

Affinity masks introduce heterogeneity by removing the symmetry among processors. An important property used to prove the SRT-optimality of GEDF without affinities is that the existence of an idle processor is a sufficient condition for a pending task to begin executing. With this property, showing that a task of interest makes progress merely requires showing that *some* processor becomes idle in a bounded amount of time. With affinity masks, this proof strategy does not work. In particular, this strategy's use would require showing that a processor *allowed by the considered task's affinity mask* becomes idle, and doing so over the entire space of arbitrary affinity masks leads to a case explosion.

This difficulty has left unresolved the question of whether GEDF retains its SRT-optimality with the introduction of affinity masks. This unresolved question has rendered systems that support both GEDF and affinity masks incomplete. For example, in the case of `SCHED_DEADLINE`, either admission control is disabled or affinity masks are forbidden altogether. This specific gap between theory and implementation was mentioned by Peter Zijlstra [9] in a keynote talk at ECRTS 2017 and by Luca Abeni [1] at RTSS 2017.

Contributions. In this paper, we close the open problem of whether GEDF retains its SRT-optimality on identical multiprocessors with affinities by showing that it does, provided the GEDF implementation maintains a certain task-migration property. We also establish the first ever polynomial tardiness bounds for GEDF on uniform multiprocessors. While these two platform models (uniform and identical with affinities) may seem unrelated to each other, we shall see that they are.

These results hinge on three proof innovations. First, we introduce a layer of abstraction between the processor models we consider and the tardiness analysis. This abstraction layer is a property we call "HP-LAG," which we prove is satisfied by GEDF on the models we consider for all sporadic task systems that do not cause platform overutilization.[1] This strategy allows us to reason about tardiness without regard for specifics concerning the underlying platform, and as a result, we are able to avoid the aforementioned case explosion.

Second, for the statement made in the prior paragraph concerning GEDF to be valid, the definition of GEDF itself must be tailored for the specific platform type under consideration. Prior work [8] has shown how to do so for the case of a uniform platform, and we show here

[1] The concept of "overutilization" is more nuanced on uniform platforms and platforms with affinities than on identical platforms with no affinities.

how to do so for an identical platform with affinities by specifying rules that GEDF must adhere to in this case. These rules specify when and how tasks must be migrated as scheduling decisions are made. Now that these rules are known, actual implementations of GEDF on systems that support affinity masks should be designed to uphold them. Unfortunately, as we discuss in App. C, the current `SCHED_DEADLINE` implementation does not do so, so a refinement of it is needed if affinity masks are to be used alongside admission control. In App. A, we present an algorithm that implements our GEDF scheduling rules for affinity masks with lower time complexity than `SCHED_DEADLINE`, given some preprocessing.

The last innovation is our improved tardiness analysis with less pessimism than prior work, particularly with respect to uniform multiprocessor platforms. This required maintaining an exponential number of invariants relative to prior work.

Organization. In the rest of this paper, we cover needed background and define the important concept of Lag, which is widely used in tardiness analysis (Sec. 2), list some general Lag properties that are predicated upon tasks being periodic and executing for their worst-case requirements (Sec. 3), define HP-LAG and present our tardiness analysis for HP-LAG-compliant schedulers using said properties (Sec. 4), show that GEDF (if implemented appropriately) is HP-LAG-compliant on both uniform multiprocessors (Sec. 5) and identical multiprocessors with affinities (Sec. 6), show that our results extend to sporadic tasks that may execute for less than their worst-case requirements (Sec. 7), explain why these results cannot be easily extended to more general models (Sec. 8), and conclude (Sec. 9).

2 Background

We consider the problem of scheduling n implicit-deadline sporadic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ on a multiprocessor $\pi = \{\pi_1, \ldots, \pi_m\}$ with m processors. We consider time to be continuous. Each task τ_i releases a sequence of jobs with a minimum separation of T_i time units between releases; T_i is called τ_i's *period*. The j^{th} released job of τ_i is denoted $J_{i,j}$, and its release time is denoted $r_{i,j}$. When this separation between the jobs of each task τ_i is exactly T_i, the system is called *periodic*. With the assumption of implicit deadlines, the *deadline* of a job of τ_i is exactly T_i time units after its release; the deadline of job $J_{i,j}$ is denoted $d_{i,j} = r_{i,j} + T_i$. The amount of work that is needed to complete a job of τ_i is bounded by τ_i's *worst-case execution requirement (WCER)* C_i, the largest of which among the tasks in τ we denote as C_{max}. Note that C_i is typically called the worst-case execution *time* in the literature. This is because much of the literature assumes that processors complete one unit of execution in one unit of time. This assumption does not hold for some of the hardware platform models we consider in this work (see Sec. 2.1). τ_i's *utilization* is defined as $u_i = C_i/T_i$. We let u_{min} be the smallest utilization among the tasks in τ. For any $\tau' \subseteq \tau$, we let $U_{\tau'}$ denote $\sum_{\tau_i \in \tau'} u_i$. A job is *pending* at time t if it has been released but has not completed. Likewise, a task is *pending* at t if any of its jobs are pending at t. A job is *ready* at t if it is the earliest released pending job from its task.

▶ **Definition 1.** *If task τ_i is pending at time t, then we define its release time at t, denoted $r_i(t)$, and its deadline at t, denoted $d_i(t)$, as the release time and deadline, respectively, of its ready job at time t. If τ_i is not pending at t, then we define $d_i(t) = \infty$.*

If a job has a deadline at time t_d and completes at time t_c, then its *tardiness* is defined as $\max(0, t_c - t_d)$. The tardiness of a task is the supremum of the tardiness of any of its jobs. If this value is finite, then we say that the task has *bounded tardiness*.

In the real-time literature, both hard real-time (HRT) and soft real-time (SRT) systems are considered. While SRT can be defined in different ways, we adopt the following definitions. A task set τ is HRT-schedulable (resp., SRT-schedulable) under a given scheduling algorithm if each task in τ has 0 (resp., bounded) tardiness in any schedule for τ generated by that algorithm. A task set τ is HRT-feasible (resp., SRT-feasible) if it is HRT-schedulable (resp., SRT-schedulable) under some scheduling algorithm. A scheduler is HRT-optimal (resp., SRT-optimal) if it can schedule any HRT-feasible (resp., SRT-feasible) system.

2.1 Multiprocessor Platform Models

There exist several multiprocessor platform models in the literature that differ by how execution speeds are allowed to vary. We formalize execution speeds as follows. Hereafter, we let $A(\mathcal{S}, \tau_i, t, t')$ denote the cumulative execution allocated to jobs of task τ_i in the schedule $\mathcal{S}$ in the time interval (t, t').

▶ **Definition 2.** *A time interval (t, t') is a continuous scheduling interval if the assignment of tasks to processors at t is maintained throughout (t, t').*

▶ **Definition 3.** *Suppose task τ_i executes on processor π_j over the continuous scheduling interval (t, t') in schedule $\mathcal{S}$. If the speed of τ_i on π_j is s, then $A(\mathcal{S}, \tau_i, t, t') = s(t' - t)$.*

In order of increasing generality, the platform models of relevance to us are as follows.

1. **Identical.** All tasks execute with speed 1.0 on all processors.
2. **Uniform.** Speeds may vary by processor, but not by task. Any task on processor π_i executes with speed s_i, which may differ from 1.0.
3. **Unrelated.** Speeds may vary by processor and by task. Task τ_i executes on processor π_j with speed $s_{i,j}$.

In addition to these models, the migration scheme can be one of two types.

1. **Global.** A task can be scheduled on any processor.
2. **Affinity.** A task can only be scheduled on a specific set of processors as defined by its *affinity mask*. We let $\alpha_i \subseteq \pi$ denote the set of processors allowed by τ_i's affinity mask.

Note that global, clustered, and partitioned scheduling can all be defined using affinity masks. We separate global scheduling as a separate case because some later proofs focus on it exclusively. From this point on, global scheduling is assumed unless the -Aff suffix is appended. For example, Identical and Identical-Aff refer to an identical multiprocessor under global and affinity scheduling, respectively. Note that Unrelated generalizes not only Identical and Uniform, but also affinity scheduling, as $\pi_j \notin \alpha_i$ can be represented by letting $s_{i,j} = 0$. Hence, all combinations of platform and migration scheme are generalized by Unrelated.

GEDF has been proven to be SRT-optimal under Identical, but no prior work has generalized this to Identical-Aff. GEDF's SRT-optimality has been generalized to Uniform [8], but with exponential tardiness bounds. In this work, we establish the SRT-optimality of GEDF with polynomial tardiness under both Uniform and Identical-Aff.

The SRT-optimality of GEDF under Uniform-Aff and Unrelated are difficult or impossible to prove using the techniques of this work. We describe why in Sec. 8.

We will later show that the typical GEDF scheduling rules under Identical may be insufficient to achieve SRT-optimality under the more general models. As such, we will later define extended GEDF scheduling rules for Identical-Aff and Uniform, respectively (our rules for Uniform are actually from [8]).

The standard GEDF scheduler under Identical is:

IG-GEDF:	At every time instant, if more than m tasks are pending, then the m pending tasks with the earliest deadlines are scheduled; otherwise, all pending tasks are scheduled.

The prefix "IG" denotes that this rule applies under Identical with global scheduling. We will keep this notation when extending GEDF, denoting the extended GEDF rules for Identical-Aff and Uniform as IA-GEDF and UG-GEDF, respectively. Under all GEDF variants we consider, we assume deadline ties are broken in some arbitrary but consistent way (e.g., by task index). For any time instant t, for tasks τ_i and τ_j, we let $d_i(t) \prec d_j(t)$ denote that either $d_i(t) < d_j(t)$ holds or $d_i(t) = d_j(t)$ holds with the tie broken in τ_i's favor. As we shall see, IA-GEDF and UG-GEDF reduce to IG-GEDF when the underlying processor model is Identical (note that both Uniform and Identical-Aff generalize Identical).

2.2 Lag

As in [5], our analysis is based on the concept of Lag. Lag compares the execution of a task in a "real" schedule $\mathcal{R}$ to its allocation in an "ideal" schedule $\mathcal{I}$.

▶ **Definition 4.** *A non-fluid schedule is a schedule such that at any time instant t, there exists some $\delta > 0$ such that $(t, t + \delta)$ is a continuous scheduling interval.*

Implementable schedulers are non-fluid.

▶ **Definition 5.** *We let $\mathcal{R}$ denote a non-fluid schedule produced under a considered scheduling algorithm and multiprocessor platform.*

▶ **Definition 6.** *We let $\mathcal{I}$ denote a schedule that executes task τ_i on processor π_i of speed u_i.*

Notice that $\mathcal{I}$ is defined with respect to a Uniform multiprocessor consisting of n processors $\pi_1, ..., \pi_n$ with speeds $u_1, ..., u_n$, respectively, and not the actual platform π.

Under the implicit-deadline sporadic task model, every job executes in $\mathcal{I}$ from its release until its completion without interference from other jobs or tasks (different tasks run on different processors). If a job's execution requirement is smaller than the WCER of its task, then the job completes in the ideal schedule before its deadline; otherwise, it completes exactly at its deadline. Thus, in $\mathcal{I}$, at most one job from every task is ever scheduled.

We are now ready to formally define Lag.

▶ **Definition 7.** *For a single task τ_i, $\mathsf{Lag}_i(t) = A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t)$. For the subset $\tau' \subseteq \tau$, $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} \mathsf{Lag}_i(t)$.*

3 General Lag Properties

In [8], Yang and Anderson showed how to generalize IG-GEDF to obtain a variant, which we denote as UG-GEDF, that is SRT-optimal under Uniform. Yang and Anderson's proof relied on several properties of Lag that we make use of in this work. We repeat these properties and proofs verbatim from [8], with minor wording changes. However, unlike [8], where these properties were considered in the context of Uniform, we consider them in the context of Unrelated. Because Unrelated generalizes all the models in Sec. 2.1, these Lag properties apply to all the models considered in this work.

Lemmas 10–12 rely on the following assumptions, which we henceforth assume until stated otherwise.

Every task is periodic. (P)

Every job $J_{i,j}$ has an execution requirement equal to C_i (the worst case for τ_i). (W)

So as to leave no doubt that the properties considered in this section hold under Unrelated, we begin by listing all of the model-related concepts the proofs below will use:

- the task-system parameters C_i, T_i, u_i, $r_i(t)$, and $d_i(t)$;
- Lag and LAG, as defined in Def. 7;
- the fact that $\mathcal{I}$ continuously executes task τ_i with speed u_i, which follows from (P) and (W), hence the need for these assumptions

[8] showed that removing (P) and (W) cannot cause greater tardiness in UG-GEDF under Uniform. We will show the same for IA-GEDF under Identical-Aff later in Sec. 7.

▶ **Lemma 8** (Property 1 of [8]). $\forall \tau' \subseteq \tau :$ $\mathsf{LAG}(\tau',\ t)$ *is a continuous function of* t.

Proof. By definition, $A(\mathcal{S}, \tau_i, t, t')$ is a continuous function in t and t'. By Def. 7, $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} \mathsf{Lag}_{\tau_i}(t) = \sum_{\tau_i \in \tau'} (A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t))$. Because $\mathsf{LAG}(\tau',\ t)$ is a finite sum of continuous functions in t, it is continuous in t. ◀

▶ **Lemma 9** (Lemma 1 of [8]). $\mathsf{Lag}_i(t) > 0 \Rightarrow \tau_i$ *has a pending job at* t.

Proof. We prove the lemma by contradiction. Suppose that $\mathsf{Lag}_i(t) > 0$ holds but τ_i is not pending at t in $\mathcal{R}$. Then, all jobs of τ_i released before or at t have completed by t. Let W denote the total execution requirement of such jobs such that $W = A(\mathcal{R}, \tau_i, 0, t)$. In $\mathcal{I}$, only released jobs can be scheduled and will not execute for more than their execution requirement. Thus, $A(\mathcal{I}, \tau_i, 0, t) \leq W$ holds as well. Therefore, by Def. 7, we have $\mathsf{Lag}_i(t) = A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t) \leq 0$, a contradiction. ◀

▶ **Lemma 10** (Lemma 2 of [8]). *If task* τ_i *is pending at time* t *in* $\mathcal{R}$*, then*

$$t - \frac{\mathsf{Lag}_i(t)}{u_i} < d_i(t) \leq t - \frac{\mathsf{Lag}_i(t)}{u_i} + T_i. \tag{1}$$

Proof. Let $e_i(t)$ denote the remaining execution requirement for the ready job $J_{i,j}$ of τ_i at time t. Because this job is ready, it must not be complete, hence

$$0 < e_i(t) \leq C_i. \tag{2}$$

All jobs of τ_i prior to $J_{i,j}$ must have been completed by time t. Let E denote the total execution requirement of these jobs. Then,

$$A(\mathcal{R}, \tau_i, 0, t) = E + C_i - e_i(t). \tag{3}$$

In $\mathcal{I}$, all prior jobs of τ_i have completed by $r_i(t)$. Within $(r_i(t), t)$, $\mathcal{I}$ continuously executes τ_i on a processor with speed u_i. Thus,

$$A(\mathcal{I}, \tau_i, 0, t) = E + (t - r_i(t))u_i. \tag{4}$$

Given these facts, an expression for $\mathsf{Lag}_i(t)$ can be derived as follows.

$\mathsf{Lag}_i(t) = \{$by Def. 7$\}$

$$
\begin{aligned}
& A(\mathcal{I}, \tau_i, 0, t) - A(\mathcal{R}, \tau_i, 0, t) \\
= & \{\text{by (3) and (4)}\} \\
& (t - r_i(t))u_i - (C_i - e_i(t)) \\
= & \{\text{because } d_i(t) = r_i(t) + T_i\} \\
& (t - d_i(t) + T_i)u_i - (C_i - e_i(t)) \\
= & \{\text{because } u_i T_i = C_i\} \\
& (t - d_i(t))u_i + e_i(t)
\end{aligned}
$$

By (2) and the above expression, we have

$$(t - d_i(t))u_i < \mathsf{Lag}_i(t) \leq (t - d_i(t))u_i + C_i, \tag{5}$$

which (using $T_i = C_i/u_i$) can be rearranged to obtain (1). ◀

▶ **Corollary 11** (Corollary 1 of [8]). *If for some $L > 0$ we have $\forall t, \mathsf{Lag}_i(t) \leq L$, then the tardiness of task τ_i does not exceed L/u_i.*

Proof. We prove the corollary by contradiction. Suppose that

$$\mathsf{Lag}_i(t) \leq L \tag{6}$$

holds but τ_i has tardiness exceeding L/u_i. Then, there exists a job $J_{i,j}$ that is pending at time $t \geq d_{i,j}$ where

$$t - d_{i,j} > L/u_i. \tag{7}$$

Because $J_{i,j}$ is pending at t, τ_i's ready job cannot have been released later than $J_{i,j}$. Thus, $d_i(t) \leq d_{i,j}$. Therefore,

$$
\begin{aligned}
t - d_i(t) & \geq t - d_{i,j} \\
& > \{\text{by (7)}\} \\
& \quad L/u_i \\
& \geq \{\text{by (6)}\} \\
& \quad \mathsf{Lag}_i(t)/u_i.
\end{aligned}
$$

Rearrangement yields $t - \mathsf{Lag}_i(t)/u_i > d_i(t)$, which contradicts Lemma 10. ◀

▶ **Lemma 12** (Lemma 4 of [8]). *If a task τ_i has a pending job at t and for a task τ_j we have*

$$\frac{1}{u_j}\mathsf{Lag}_j(t) + T_{max} \leq \frac{1}{u_i}\mathsf{Lag}_i(t), \tag{8}$$

then $d_i(t) < d_j(t)$.

Proof. Assume τ_j is pending at t, as otherwise $d_j(t) = \infty$, and we trivially have $d_i(t) < d_j(t)$.

$$
\begin{aligned}
d_i(t) & \leq \{\text{by Lemma 10}\} \\
& \quad t - \frac{\mathsf{Lag}_i(t)}{u_i} + T_i \\
& \leq \{\text{by (8)}\}
\end{aligned}
$$

$$
\begin{aligned}
& t - \frac{\mathsf{Lag}_j(t)}{u_j} - T_{max} + T_i \\
\leq\ & \{\text{because } -T_{max} + T_i \leq 0\} \\
& t - \frac{\mathsf{Lag}_j(t)}{u_j} \\
<\ & \{\text{by Lemma 10}\} \\
& d_j(t)
\end{aligned}
$$

◀

As a reminder, we are considering the properties in this section in the context of Unrelated. This means these properties apply to all of our considered models.

4 Tardiness Bounds for HP-LAG-Compliant Schedulers

In this paper, we consider two Identical generalizations: Uniform and Identical-Aff. In order to prove GEDF's SRT-optimality under these different processor models, we provide an abstraction layer called HP-LAG. Informally, HP-LAG states that the LAG of any subset of tasks τ' with the earliest deadlines is temporarily non-increasing.

HP-LAG:	For any time instant t, if $\tau' \subseteq \tau$ is a set of pending tasks such that $\forall \tau_h \in \tau'$ and $\forall \tau_\ell \in \tau/\tau'$ we have $d_h(t) < d_\ell(t)$, then $\exists \delta > 0$ such that $\forall t' \in (t, t+\delta) :$ $\mathsf{LAG}(\tau',\ t) \geq \mathsf{LAG}(\tau',\ t')$.

▶ **Definition 13.** *We say that a scheduler is HP-LAG-compliant under a given platform if HP-LAG holds for any feasible task system τ under said platform model.*

In this section, we show that every HP-LAG-compliant scheduler ensures bounded tardiness under its considered platform model. We do this by extending the approach of [8] by maintaining as invariants bounds on the LAG of *every* task subset; in [8], only a *linear* (with respect to m) number of invariants is instead maintained. The LAG bound we define for any subset of tasks $\tau' \subseteq \tau$ is

$$\beta(\tau') = \frac{T_{max}}{2u_{min}} U_{\tau'} \left(2U_\tau - U_{\tau'}\right). \tag{9}$$

We will prove by contradiction that $\forall \tau' \subseteq \tau\ \forall t :\ \mathsf{LAG}(\tau',\ t) \leq \beta(\tau')$ holds for any HP-LAG-compliant scheduler. We continue to consider all properties in the context of Unrelated. Thus, Lemmas 16, 17, 18, and 19 below hold for any scheduler that is HP-LAG-compliant under any processor model that Unrelated generalizes. We begin by defining a set of time instants that must exist if our LAG bounds are violated.

▶ **Definition 14.** *We call a time instant t invalid if $\exists \tau' \subseteq \tau$ such that $\forall \delta > 0\ \exists t' \in (t, t+\delta) :$ $\mathsf{LAG}(\tau',\ t') > \beta(\tau')$. τ' is called an attestant set of the invalid instant t.*

Note that for any invalid instant, $\emptyset$ is never an attestant set because for any time instant t we have $\mathsf{LAG}(\emptyset,\ t) = 0$ and $\beta(\emptyset) = 0$.

▶ **Definition 15.** *If at least one invalid time instant exists, then we call the first[2] such instant a boundary instant, denoted t_b. We let τ^b denote an arbitrary attestant set of t_b. We call any task from τ^b a boundary task.*

[2] It can be shown that the infimum of all invalid instants is itself an invalid instant. Hence, the first invalid instant, t_b, is well-defined.

Because t_b is the first invalid instant, we can prove specific bounds on Lag values at t_b.

▶ **Lemma 16.** *For the boundary instant t_b, the following three expressions hold.*

$$\forall \tau' \subseteq \tau : \mathsf{LAG}(\tau',\ t_b) \leq \beta(\tau') \tag{10}$$

$$\mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b) \tag{11}$$

$$\forall \delta > 0\ \exists t' \in (t_b, t_b + \delta) : \mathsf{LAG}(\tau^b,\ t') > \beta(\tau^b) \tag{12}$$

Proof. We prove (10) by contradiction. If for some $\tau' \subseteq \tau$, $\mathsf{LAG}(\tau',\ t_b) > \beta(\tau')$, then, by Lemma 8 (continuity of $\mathsf{LAG}(\tau',\ t)$), $\exists \delta > 0\ \forall t' \in (t_b - \delta, t_b) :\ \mathsf{LAG}(\tau',\ t') > \beta(\tau')$. Thus, time instant $t_b - \delta$ is invalid with attestant set τ', which contradicts Def. 15.

By Defs. 14 and 15, $\forall \delta > 0\ \exists t' \in (t_b, t_b + \delta) :\ \mathsf{LAG}(\tau^b,\ t') > \beta(\tau^b)$. By Lemma 8 (continuity of $\mathsf{LAG}(\tau^b,\ t)$), we have $\mathsf{LAG}(\tau^b,\ t_b) \geq \beta(\tau^b)$. By (10), $\mathsf{LAG}(\tau^b,\ t_b) \leq \beta(\tau^b)$, so (11) holds.

(12) follows from Defs. 14 and 15. ◀

▶ **Lemma 17.** *For any boundary task τ_i at t_b, $\mathsf{Lag}_i(t_b) \geq \dfrac{T_{max}}{2u_{min}}(2u_i U_\tau - 2u_i U_{\tau^b} + u_i^2)$.*

Proof.

$$\begin{aligned}
\mathsf{Lag}_i(t_b) &= \{\text{by Def. 7}\} \\
&\quad \mathsf{LAG}(\tau^b,\ t_b) - \mathsf{LAG}(\tau^b/\{\tau_i\},\ t_b) \\
&\geq \{\text{by (11), } \mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b), \text{ and by (10), } \mathsf{LAG}(\tau^b/\{\tau_i\},\ t_b) \leq \beta(\tau^b/\{\tau_i\})\} \\
&\quad \beta(\tau^b) - \beta(\tau^b/\{\tau_i\}) \\
&= \{\text{by (9)}\} \\
&\quad \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right] - \frac{T_{max}}{2u_{min}}\left[U_{\tau^b/\{\tau_i\}}\left(2U_\tau - U_{\tau^b/\{\tau_i\}}\right)\right] \\
&= \{\text{by the definition of } U_{\tau^b/\{\tau_i\}}\} \\
&\quad \frac{T_{max}}{2u_{min}}\left[U_{\tau^b}\left(2U_\tau - U_{\tau^b}\right)\right] - \frac{T_{max}}{2u_{min}}\left[\left(U_{\tau^b} - u_i\right)\left(2U_\tau - U_{\tau^b} + u_i\right)\right] \\
&= \{\text{rearranging}\} \\
&\quad \frac{T_{max}}{2u_{min}}(2u_i U_\tau - 2u_i U_{\tau^b} + u_i^2)
\end{aligned}$$

◀

Note that the Lag lower bound from Lemma 17 is strictly positive, because $U_\tau \geq U_{\tau^b}$. Thus, by Lemma 9, any boundary task τ_i is pending at t_b. This proves the following lemma.

▶ **Lemma 18.** *At the boundary time instant t_b, every boundary task has a pending job.*

As shown next, similar reasoning as in Lemma 17 can be used to upper bound the Lag of non-boundary tasks. This allows us to establish a relationship between the deadlines of boundary and non-boundary tasks.

▶ **Lemma 19.** *At time instant t_b, every boundary task has an earlier deadline than any non-boundary task (i.e., from τ/τ^b).*

Proof. For any non-boundary task $\tau_j \in \tau/\tau^b$, we have the following.

$$\begin{aligned}
\mathsf{Lag}_j(t_b) &= \{\text{by Def. 7}\} \\
&\quad \mathsf{LAG}(\tau^b \cup \{\tau_j\},\ t_b) - \mathsf{LAG}(\tau^b,\ t_b)
\end{aligned}$$

$$
\begin{aligned}
&\leq \{\text{by (10), } \mathsf{LAG}(\tau^b \cup \{\tau_j\},\ t_b) \leq \beta(\tau^b \cup \{\tau_j\}), \text{ and by (11), } \mathsf{LAG}(\tau^b,\ t_b) = \beta(\tau^b)\} \\
&\quad \beta(\tau^b \cup \{\tau_j\}) - \beta(\tau^b) \\
&= \{\text{by (9)}\} \\
&\quad \frac{T_{max}}{2u_{min}} \left[U_{\tau^b \cup \{\tau_j\}} \left(2U_\tau - U_{\tau^b \cup \{\tau_j\}}\right)\right] - \frac{T_{max}}{2u_{min}} \left[U_{\tau^b} \left(2U_\tau - U_{\tau^b}\right)\right] \\
&= \{\text{by the definition of } U_{\tau^b \cup \{\tau_j\}}\} \\
&\quad \frac{T_{max}}{2u_{min}} \left[\left(U_{\tau^b} + u_j\right) \left(2U_\tau - U_{\tau^b} - u_j\right)\right] - \frac{T_{max}}{2u_{min}} \left[U_{\tau^b} \left(2U_\tau - U_{\tau^b}\right)\right] \\
&= \{\text{rearranging}\} \\
&\quad \frac{T_{max}}{2u_{min}} (2u_j U_\tau - 2u_j U_{\tau^b} - u_j^2) \qquad (13)
\end{aligned}
$$

To conclude the proof, we show that the Lag of a boundary task $\tau_i \in \tau^b$ and the Lag of a non-boundary task $\tau_j \in \tau/\tau^b$ together satisfy the requirement specified in Lemma 12.

$$
\begin{aligned}
\frac{1}{u_j}\mathsf{Lag}_j(t_b) + T_{max} &\leq \{\text{by (13)}\} \\
&\quad \frac{1}{u_j} \frac{T_{max}}{2u_{min}} (2u_j U_\tau - 2u_j U_{\tau^b} - u_j^2) + T_{max} \\
&= \{\text{factor in } 1/u_j \text{ and out } 1/u_i \text{ from } 2u_j U_\tau - 2u_j U_{\tau^b}\} \\
&\quad \frac{1}{u_i} \frac{T_{max}}{2u_{min}} (2u_i U_\tau - 2u_i U_{\tau^b}) + T_{max} \left(1 - \frac{u_j}{2u_{min}}\right) \\
&\leq \{2u_{min} - u_j = u_{min} + (u_{min} - u_j) \leq u_{min} \leq u_i\} \\
&\quad \frac{1}{u_i} \frac{T_{max}}{2u_{min}} (2u_i U_\tau - 2u_i U_{\tau^b}) + T_{max} \frac{1}{u_i} \left(\frac{u_i^2}{2u_{min}}\right) \\
&\leq \{\text{by Lemma 17}\} \\
&\quad \mathsf{Lag}_i(t_b) \qquad (14)
\end{aligned}
$$

By Lemma 18, a boundary task τ_i is pending, and by Lemma 12, its deadline is earlier than task $\tau_j \in \tau/\tau^b$ (if τ_j has no pending job, then $d_j(t) = \infty$ by Def. 1) at time t_b. ◀

▶ **Theorem 20.** *If a scheduler is HP-LAG-compliant under its considered platform, then for any feasible task system τ, the tardiness of task $\tau_i \in \tau$ is at most*

$$\frac{T_{max}}{2u_{min}} (2U_\tau - u_i). \qquad (15)$$

Proof. If there exists at least one invalid instant, we can define the boundary time instant t_b with an attestant set τ^b. By Lemma 19, tasks in τ^b have earlier deadlines than any task in τ/τ^b. Thus, by HP-LAG with $\tau' = \tau^b$,

$$\exists \delta > 0\ \forall t \in (t_b, t_b + \delta) :\ \mathsf{LAG}(\tau^b,\ t_b) \geq \mathsf{LAG}(\tau^b,\ t). \qquad (16)$$

However, by Lemma 16,

$$\forall \delta > 0\ \exists t \in (t_b, t_b + \delta) :\ \mathsf{LAG}(\tau^b,\ t) > \beta(\tau^b) = \mathsf{LAG}(\tau^b,\ t_b),$$

which contradicts (16). Because the existence of t_b leads to a contradiction, there is no first invalid instant, and hence there are no invalid time instants. Thus, by Def. 14,

$$\forall \tau' \subseteq \tau\ \forall t \geq 0\ \exists \delta > 0\ \forall t' \in (t, t + \delta) :\ \mathsf{LAG}(\tau',\ t') \leq \beta(\tau').$$

By Lemma 8, it follows that

$$\forall \tau' \subseteq \tau \; \forall t \geq 0 : \; \mathsf{LAG}(\tau', \; t) \leq \beta(\tau'). \tag{17}$$

Hence, for any task τ_i and any time instant t,

$$\begin{aligned}
\mathsf{Lag}_i(t) &= \{\text{by Def. 7}\} \\
&\quad \mathsf{LAG}(\{\tau_i\}, \; t) \\
&\leq \{\text{by (17) with } \tau' = \{\tau_i\}\} \\
&\quad \beta(\{\tau_i\}) \\
&= \{\text{by (9)}\} \\
&\quad \frac{T_{max}}{2u_{min}} u_i(2U_\tau - u_i).
\end{aligned}$$

Thus, by Corollary 11, task τ_i has maximum tardiness at most $\frac{T_{max}}{2u_{min}}(2U_\tau - u_i)$. ◀

The theorem above is proved under the context of Unrelated. Thus, the theorem holds for HP-LAG-compliant schedulers under Uniform and Identical-Aff because these models are special cases of Unrelated. In Secs. 5 and 6, we demonstrate that the GEDF generalizations discussed in this work are HP-LAG-compliant.

5 GEDF Tardiness Bounds under the Uniform Model

In this section, we show that UG-GEDF, the generalization of IG-GEDF under Uniform in [8], is HP-LAG-compliant. This result enables us to apply Theorem 20 to obtain tardiness bounds for UG-GEDF that are superior to those in [8].

5.1 Refining GEDF for the Uniform Model

IG-GEDF is not SRT-optimal under Uniform. Consider a single-task system $\tau = \{\tau_1\}$ with $C_1 = 1$ and $T_1 = 2$ (hence $u_1 = 0.5$) running on $\pi = \{\pi_1, \pi_2\}$ with $s_1 = 1$ and $s_2 = 0.1$. This system is clearly feasible if τ_1 always executes on the faster processor π_1. Under IG-GEDF, however, it is legal for τ_1 to be continuously scheduled on the slower processor π_2. This would lead to unbounded tardiness, as π_2's speed is lower than τ_1's utilization.

Such counterexamples led Yang and Anderson to define UG-GEDF as below. For the remainder of this section, we assume that processors are indexed by decreasing speed.

UG-GEDF:	At any time instant, the ready job of the pending task with the k^{th} earliest deadline is scheduled on π_k for $k \in [1, m]$.

5.2 HP-LAG-Compliance for UG-GEDF

To prove HP-LAG-compliance, we reference the feasibility condition under Uniform, which references the following definition.

▶ **Definition 21.** *Let* $S_k = \sum_{i=1}^{k} s_i$ *for* $k \leq m$.

When tasks are indexed by decreasing utilization, the feasibility condition is as follows [6, 8].

$$\forall k \leq m : \sum_{i=i}^{k} u_i \leq S_k \tag{18}$$

$$U_\tau \leq S_m \tag{19}$$

In terms of subsets of tasks, this condition is equivalent to

$$\forall \tau' \subseteq \tau : U_{\tau'} \leq S_{\min(|\tau'|,m)}. \tag{20}$$

▶ **Lemma 22.** *UG-GEDF is HP-LAG-compliant under Uniform.*

Proof. By Def. 13, we need only consider the case that task system τ is feasible under Uniform. Let $\tau' \subseteq \tau$ be any subset as defined in HP-LAG for any time instant t. HP-LAG states that the tasks in τ' have earlier deadlines at time t than any tasks outside of τ'. Under UG-GEDF, tasks from τ' occupy the $\min(|\tau'|, m)$ fastest processors. Because UG-GEDF is a non-fluid scheduler (see Def. 4), for any time instant t, for some $\delta > 0$ we have that $(t, t+\delta)$ is a continuous scheduling interval (see Def. 2). For any $t' \in (t, t+\delta)$,

$$\begin{aligned}
\mathsf{LAG}(\tau',\ t') =& \{\text{by Def. 7}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, t, t') - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
=& \{\text{by Defs. 3 and 6}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
=& \{\text{tasks from } \tau' \text{ occupy processors with speeds } s_1, ..., s_{\min(|\tau'|,m)}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - \sum_{i=1}^{\min(|\tau'|,m)} s_i(t'-t) \\
=& \{\text{by Def. 21}\} \\
& \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} u_i(t'-t) - S_{\min(|\tau'|,m)} \cdot (t'-t) \\
=& \mathsf{LAG}(\tau',\ t) + (t'-t)\left[U_{\tau'} - S_{\min(|\tau'|,m)}\right] \\
\leq& \{t' > t \text{ by definition and } U_{\tau'} \leq S_{\min(|\tau'|,m)} \text{ by (20)}\} \\
& \mathsf{LAG}(\tau',\ t).
\end{aligned}$$

Therefore, UG-GEDF is HP-LAG-compliant. ◀

Because we have proven in Lemma 22 that UG-GEDF is HP-LAG-compliant, we have the following corollary of Theorem 20.

▶ **Corollary 23.** *Under UG-GEDF on a Uniform multiprocessor executing a feasible task system, the tardiness of any task τ_i is at most (15).*

We assumed without loss of generality that tasks (processors) are indexed by decreasing utilizations (speeds) to make stating UG-GEDF and the Uniform feasibility condition simpler. We no longer keep these assumptions in the following sections.

6 GEDF Tardiness Bounds under the Identical Model with Affinities

In this section, we generalize the IG-GEDF scheduling rules under the context of Identical-Aff. We show that the resulting scheduling policy, IA-GEDF, is HP-LAG-compliant. Thus, Theorem 20 ensures bounded tardiness because Identical-Aff is a special case of Unrelated.

Under Identical-Aff, all processors have speed equal to 1.0. As in [7], we use in our analysis the concept of an affinity graph.

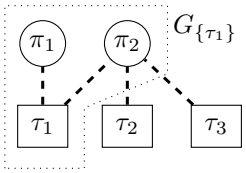


Figure 1 Affinity graph (AG) example.

▶ **Definition 24.** *An affinity graph (AG) G_τ of a task system τ on platform π with affinity masks is a undirected bipartite graph containing one vertex for each task and each processor. An edge exists between τ_i and π_j if and only if $\pi_j \in \alpha_i$. For any $\tau' \subseteq \tau$, we let $G_{\tau'}$ denote the subgraph of G_τ containing only the vertices that correspond to the tasks in τ' and the processors in the union of their affinity masks.*

▶ **Example 25.** An example G_τ for the task system $\tau = \{\tau_1, \tau_2, \tau_3\}$ on $\pi = \{\pi_1, \pi_2\}$ with $\alpha_1 = \{\pi_1, \pi_2\}$ and $\alpha_2 = \alpha_3 = \{\pi_2\}$ is given in Fig. 1. The same figure also shows $G_{\{\tau_1\}}$.

6.1 Refining GEDF for the Identical Model with Affinities

Unlike under Identical or Uniform, it is not always possible to schedule the m tasks with the earliest deadlines under Identical-Aff. Consider the example in Fig. 1 with two processors and three tasks. If all tasks are pending at a time instant t and the deadlines are such that $d_2(t) < d_3(t) < d_1(t)$ holds, then the tasks with the earliest deadlines, τ_2 and τ_3, cannot be simultaneously scheduled because they share a single processor.

The choice of processor assignments can also leave a processor idle, i.e., with no job to execute. Consider again Fig. 1 with the assumption that only τ_1 is pending and is assigned to π_2. Suppose that τ_2 at time instant t releases a job such that $d_1(t) < d_2(t)$. Under IG-GEDF, a lower-priority task such as τ_2 would be scheduled on π_1, but affinity-mask restrictions disallow this. Because $d_1(t) < d_2(t)$, τ_1 has higher priority, and τ_2 is not scheduled, leaving processor π_1 idle. However, forcing τ_1 to migrate to π_1 is a more efficient use of processor capacity in this example. The problem here is that the availability of processors for different tasks under affinity scheduling is not symmetrical.

Under IG-GEDF, a preemption only affects the preempting and preempted tasks and a single processor. We define IA-GEDF to extend this preemption rule to avoid unnecessary idleness. To formally specify IA-GEDF, we require several graph-theory definitions.

▶ **Definition 26.** *A matching of a graph G with edge set E is an edge set $M \subseteq E$ such that no two edges in M share a common vertex. A vertex is matched if it is an endpoint of one of the edges in the matching; otherwise, the vertex is unmatched.*

▶ **Example 27.** An example matching can be found in the left graph of Fig. 2. We have the matching $M = \{(\tau_1, \pi_2), (\tau_3, \pi_3), (\tau_4, \pi_4)\}$, with vertices π_1 and τ_2 being unmatched.

Observe that any assignment of tasks to processors at runtime that obeys the tasks' affinity masks defines a matching of G_τ (and vice versa). While the concept of a matching can be applied to any graph, we restrict attention to only AGs and their subgraphs.

▶ **Definition 28.** *An alternating path p of a matching in a graph is a path that begins with an unmatched "task" vertex, has edges that alternately are in the matching and not in the matching, and ends with a "processor" vertex. An augmenting path p of a matching is an alternating path that ends with an unmatched vertex.*

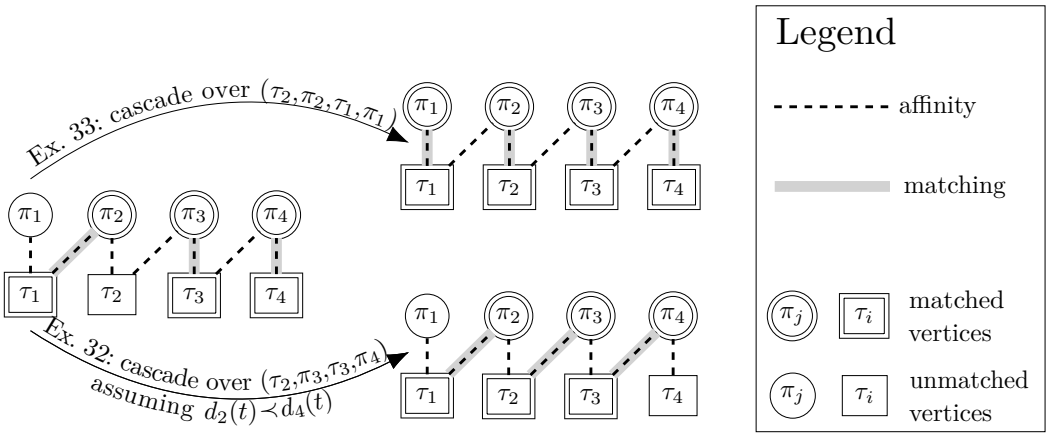


Figure 2 Two examples of a scheduling cascade.

▶ **Example 29.** In the left graph of Fig. 2, $(\tau_2, \pi_3, \tau_3, \pi_4)$ is an alternating path that is not an augmenting path for the given matching, and $(\tau_2,\pi_2,\tau_1,\pi_1)$ is an augmenting path.

Observe that an alternating path must have an odd number of edges because its first and last vertices are task and processor vertices, respectively. The following lemma establishes a relationship between augmenting and alternating paths.

▶ **Lemma 30.** *Consider a matching M for G_τ and $\tau' \subseteq \tau$. Let M' denote the set of edges of M that are present in $G_{\tau'}$. Then, M' is a matching in $G_{\tau'}$. Furthermore, if p is an augmenting path of the matching M' in $G_{\tau'}$, then p is an alternating path of M in G_τ.*

Proof. By Def. 26, no two edges in M share a common vertex. This does not change when removing edges and vertices from G_τ and edges from M, through which we get $G_{\tau'}$ and M', respectively. Thus, M' is also a matching in $G_{\tau'}$.

Let the first vertex of p (as mentioned in the lemma statement) be task τ_i. By Def. 28, τ_i is unmatched in $G_{\tau'}$. By Def. 24, $G_{\tau'}$ contains τ_i and all the processors in its affinity mask. Hence, all edges from τ_i that are in G_τ are also in $G_{\tau'}$. Thus, τ_i is also unmatched in G_τ.

By definition, M' contains all the edges in M that are also in $G_{\tau'}$. Hence, an edge of $G_{\tau'}$ is in M' if and only if it is also in M, which applies to all edges of p because it is contained in $G_{\tau'}$. Because an augmenting path is a special case of an alternating path, edges of p are alternately in and not in M', and hence, alternately in and not in M. This fact and the fact that the first vertex of p, task τ_i, is unmatched in M make p an alternating path in G_τ. ◀

We now have sufficient terminology to define a generalized notion of a preemption that occurs under Identical-Aff and that we call a *scheduling cascade*. Informally, in a scheduling cascade, an unscheduled task is scheduled by making an idle processor busy or by unscheduling a different task, perhaps on a different processor, with a later deadline. Note that this may require several migrations.

▶ **Definition 31.** *A scheduling cascade at time t via the alternating path p in G_τ changes the task-to-processor assignments (i.e., the matching M at t in G_τ) via Alg. 1.*

We can write $p = (\tau_{i_1}, \pi_{j_1}, \tau_{i_2}, \pi_{j_2}, \ldots, \tau_{i_k}, \pi_{j_k})$ for some $k \geq 1$ and task (resp., processor) indicies $i_1, i_2, \ldots, i_k$ (resp., $j_1, j_2, \ldots, j_k$) because the first vertex of p is a task by Def. 28. Because p is alternating, $\forall r \in [1, k-1] : (\tau_{i_{r+1}}, \pi_{j_r}) \in M$ and $\forall r \in [1, k] : (\tau_{i_r}, \pi_{j_r}) \notin M$.

Note that M remains a matching after a scheduling cascade. All tasks and processors in p are unmatched prior to line 6 and each iteration of line 7 matches a distinct task and processor from the other iterations.

Input : Matching M of the AG G_τ at time t;
Alternating path $p = (\tau_{i_1}, \pi_{j_1}, \tau_{i_2}, \pi_{j_2}, \ldots, \tau_{i_k}, \pi_{j_k})$ such that if τ_ℓ exists such that $(\tau_\ell, \pi_{j_k}) \in M$, then $d_{i_1}(t) \prec d_\ell(t)$

1 **if** π_{j_k} *is matched in* M **then**
2 | $\tau_\ell \leftarrow \pi_{j_k}$'s matched task;
3 | Remove edge (τ_ℓ, π_{j_k}) from M;
4 **for** $r \in [1, k-1]$ **do**
5 | Remove edge $(\tau_{i_{r+1}}, \pi_{j_r})$ from M;
6 **for** $r \in [1, k]$ **do**
7 | Add edge (τ_{i_r}, π_{j_r}) to M;
8 **return**

Algorithm 1: The scheduling cascade algorithm.

▶ **Example 32.** Consider the lower scheduling cascade in Fig. 2. In the scheduling cascade, we have $\tau_{i_1} = \tau_2$ and $\pi_{j_k} = \pi_4$. Because π_4 is matched to task τ_4, we have $\tau_\ell = \tau_4$. Thus, the condition $d_{i_1}(t) \prec d_\ell(t)$ becomes $d_2(t) \prec d_4(t)$ in this example, and we remove edge (τ_4, π_4) from the matching (line 3). Of the edges in the alternating path, we remove edge (τ_3, π_3) (line 5) and add edges (τ_2, π_3) and (τ_3, π_4) (line 7). This results in a new matching, as indicated in Fig. 2.

▶ **Example 33.** Consider the higher scheduling cascade in Fig. 2. In the scheduling cascade, we have $\tau_{i_1} = \tau_2$ and $\pi_{j_k} = \pi_1$. We do not execute line 3 because π_1 is unmatched. Of the edges in the alternating path, we remove edge (τ_1, π_2) (line 5) and add edges (τ_2, π_2) and (τ_1, π_1) (line 7). This results in a new matching, as indicated in Fig. 2.

The net result of a scheduling cascade is that task τ_{i_1} that was not scheduled prior to the scheduling cascade is now scheduled, and task τ_ℓ (if it exists) with later deadline than τ_{i_1} is now not scheduled. All other tasks that were scheduled in matching M continue to be scheduled after the scheduling cascade, though the other tasks in p have migrated.

To define our GEDF scheduling policy with affinity masks, we define the task-to-processor assignments at scheduling events, i.e., job releases or job completions, and assume these assignments hold between scheduling events.

IA-GEDF:	At every scheduling event, task-to-processor assignments are made such that afterwards no scheduling cascade is possible. These assignments do not change until the next scheduling event.

One might assume that a simpler scheduling policy may be sufficient, but issues arise when weaker scheduling rules are used. For example, `SCHED_DEADLINE` under Identical-Aff is not HP-LAG-compliant. These details can be found in App. C.

Because we do not explicitly specify the task-to-processor assignments, there may exist multiple assignments that satisfy IA-GEDF at every scheduling event. Note that every scheduling cascade either schedules an additional task or replaces a scheduled task with an unscheduled task with an earlier deadline. Thus, the total number of possible scheduling cascades per scheduling event is finite. To show IA-GEDF is possible to implement, we created an $O(mn)$ algorithm that computes task-to-processor assignments at every scheduling event that obeys IA-GEDF. With offline preprocessing, the time complexity is reduced to $O(m + \log n)$ per scheduling event. The algorithm and preprocessing details are available in App. A. In App. C, we show that `SCHED_DEADLINE` under Identical-Aff has higher time complexity per scheduling event.

Note that a preemption in IG-GEDF can be interpreted as a scheduling cascade with an alternating path of a single edge. Thus, IA-GEDF reduces to IG-GEDF under Identical.

6.2 HP-LAG-Compliance for IA-GEDF

Here we consider only feasible task systems τ because HP-LAG-compliance is defined only for such systems. Exact feasibility conditions under Identical-Aff were established in [7], but in our reasoning about IA-GEDF's HP-LAG-compliance, we only use a necessary condition for a feasible task system, provided in Lemma 35.

▶ **Definition 34.** *A matching M of a graph G is a maximal matching if $|M| \geq |M'|$ for any matching M' of G, where $|M|$ denotes the number of edges of the matching M.*

▶ **Lemma 35.** *If a task system τ is feasible under Identical-Aff, then for any task subset $\tau' \subseteq \tau$, a maximal matching M' under $G_{\tau'}$ has $|M'| \geq U_{\tau'}$ edges.*

Proof. Suppose otherwise that τ is feasible and there exists τ' such that a maximal matching M' under $G_{\tau'}$ has $|M'|$ edges with $|M'| < U_{\tau'}$. Because M' is maximal, the tasks of τ' are scheduled on at most $|M'|$ processors at any time instant, and hence, for any schedule $\mathcal{R}$ and time instant t, $\sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, 0, t) \leq |M'|t$. Hence, by Defs. 3, 6, 7, and the definition of $U_{\tau'}$, we have $\mathsf{LAG}(\tau',\ t) = \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, 0, t) - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, 0, t) \geq U_{\tau'}t - |M'|t$. Because $U_{\tau'} > |M'|$, $U_{\tau'}t - |M'|t \to \infty$ as $t \to \infty$. Thus, $\mathsf{LAG}(\tau',\ t)$ is unbounded under any schedule $\mathcal{R}$, making τ' unfeasible, which contradicts the asumption that τ is feasible. ◀

We use Berge's Theorem to prove that IA-GEDF is HP-LAG-compliant. Berge's definition of an augmenting path reduces to Def. 28 in the context of an AG and its subgraphs.

▶ **Theorem 36** (Theorem 1 of [3], Berge's Theorem)**.** *A matching M of a graph G is maximal if and only if there is no augmenting path for M and G.*

▶ **Lemma 37.** *IA-GEDF is HP-LAG-compliant under Identical-Aff.*

Proof. We use Fig. 3 to illustrate the key steps of the proof. Consider a matching M in G_τ defined by the IA-GEDF task-to-processor assignments at time t. Let τ' be as defined in HP-LAG at time t ($\tau' = \{\tau_1, \tau_2, \tau_3\}$ in Fig. 3). Consider a matching M' in $G_{\tau'}$ defined by the assignments of tasks in τ' to processors ($M' = \{(\tau_1, \pi_1), (\tau_3, \pi_2)\}$).

We will first show that M' is maximal in $G_{\tau'}$. Suppose otherwise that M' is not maximal in $G_{\tau'}$. Then, by Theorem 36, there exists an augmenting path p for M' in $G_{\tau'}$ ($p = (\tau_2, \pi_2, \tau_3, \pi_3)$). By Lemma 30, p is an alternating path for M in G_τ.

Consider the task τ_h that is represented by the first vertex of p (τ_2), and the processor π_j that is represented by the last vertex of p (π_3). If there is a task τ_ℓ that is scheduled on π_j (τ_4 on π_3), then $\tau_h \in \tau'$ and $\tau_\ell \in \tau/\tau'$ because p is an augmenting path in $G_{\tau'}$. By the definition of τ' in HP-LAG, we have $d_h(t) < d_\ell(t)$ ($d_2(t) < d_4(t)$). Thus, because we have satisfied the input requirements of Alg. 1, we can perform a scheduling cascade via p at time t (remove (τ_4, π_3) and (τ_3, π_2) and add (τ_3, π_3) and (τ_2, π_2) to the matching), which contradicts our definition of IA-GEDF. Hence, M' is maximal.

Because IA-GEDF produces a non-fluid schedule (Def. 4), the interval $(t, t+\delta)$ must be a continuous scheduling interval for some $\delta > 0$. Thus, for all $t' \in (t, t+\delta)$,

$\mathsf{LAG}(\tau',\ t') =$ {by Def. 7}

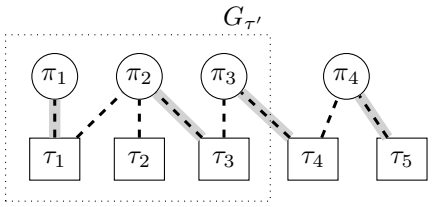


Figure 3 An example graph $G_{\tau'}$ for the proof of Lemma 37. Solid gray edges represent the task-to-processor assignments prior to a scheduling cascade and dashed edges represent affinity.

$$
\begin{aligned}
&\mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} A(\mathcal{I}, \tau_i, t, t') - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
&= \{\text{by Defs. 3 and 6}\} \\
&\quad \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} (t' - t)u_i - \sum_{\tau_i \in \tau'} A(\mathcal{R}, \tau_i, t, t') \\
&= \{\text{the number of processors scheduling tasks of } \tau' \text{ is } |M'|\} \\
&\quad \mathsf{LAG}(\tau',\ t) + \sum_{\tau_i \in \tau'} (t' - t)u_i - (t' - t)|M'| \\
&= \{\sum_{\tau_i \in \tau'} u_i = U_{\tau'}\} \\
&\quad \mathsf{LAG}(\tau',\ t) + (t' - t)(U_{\tau'} - |M'|) \\
&\leq \{t < t' \text{ and by Lemma 35, } U_{\tau'} \leq |M'|\} \\
&\quad \mathsf{LAG}(\tau',\ t).
\end{aligned}
$$

Therefore, IA-GEDF is HP-LAG-compliant. ◀

Applying Theorem 20 yields the following.

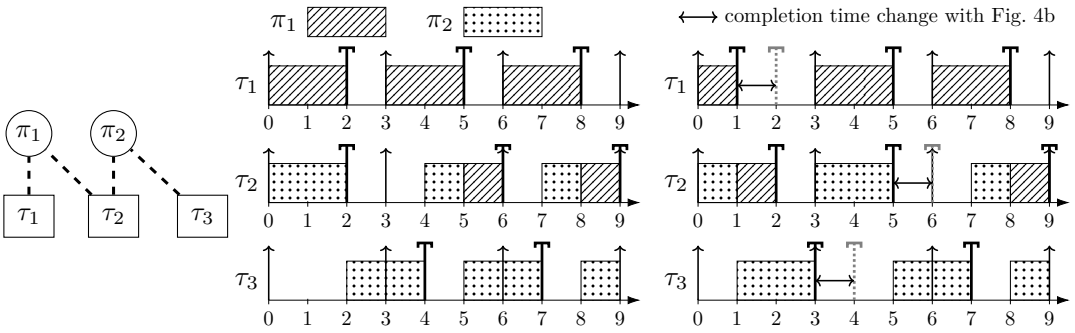
▶ **Corollary 38.** *Under IA-GEDF on a Identical-Aff multiprocessor executing a feasible task system, the tardiness of any task τ_i is at most (15).*

7 Extending to the Sporadic Task Model

As in [8], we made assumptions (P) and (W). It was proven in Theorems 3 and 4 of [8] that any tardiness bounds derived for UG-GEDF under Uniform assuming (P) and (W) hold without these assumptions. Thus, Corollary 23, which establishes tardiness bounds for UG-GEDF under Uniform, applies to sporadic tasks.

It remains to show that these assumptions can be removed from Corollary 38, which pertains to IA-GEDF on Identical-Aff. We use reasoning similar to [8] to show this, though some details are changed due to the different scheduler and platform. Due to space constraints, we present a proof sketch and provide the formal proofs in App. B.

Removing assumption (W). The intuition here is that reducing the execution requirement of a job cannot cause jobs to complete later. Consider Figs. 4b and 4c, which show two schedules under IA-GEDF for two periodic instances of the task system defined in Fig. 4a. In Fig. 4b, assumption (W) is true, while in Fig. 4c, job $J_{1,1}$ completes with 1.0 less execution unit than τ_1's WCER of 2.0. As a result, jobs $J_{2,2}$ and $J_{3,1}$ complete earlier in Fig. 4c than in



(a) Affinity graph. **(b)** IA-GEDF schedule with (W). **(c)** IA-GEDF schedule where the first job of τ_1 completes early.

Figure 4 An example task system where removing assumption (W) only causes jobs to complete earlier. When the completion times in the two schedules differ for a job, a dashed gray marker in 4c indicates its original completion time in 4b. Every task has $(C, T) = (2, 3)$.

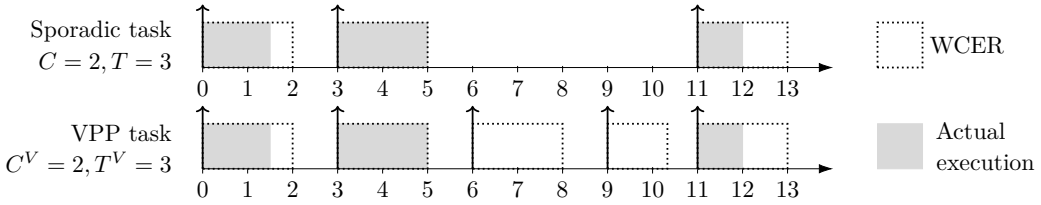


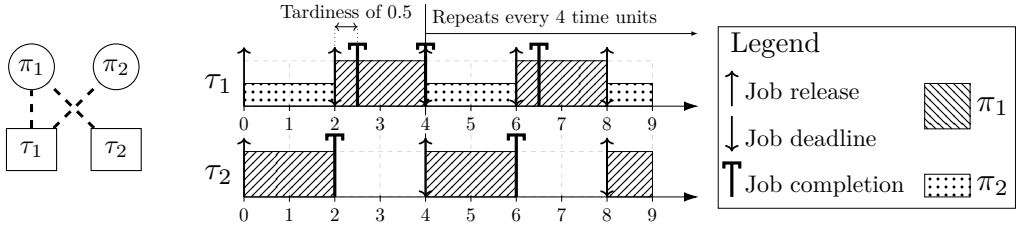
Figure 5 Transformation of a sporadic task to a VPP task.

Fig. 4b, while all other jobs complete at the same time in both schedules (the two schedules converge at time $t = 6$). Thus, reducing the execution time of a job did not increase tardiness for any other job. We prove in App. B that this is always the case. By applying this fact inductively, it follows that tardiness bounds derived assuming (W) hold in systems without (W) under IA-GEDF on Identical-Aff.

Removing assumption (P). In order to remove (P), we use the *varying-period periodic (VPP)* task model, defined in [8]. A VPP task τ_i^V of task set τ^V is defined through its utilization u_i^V. Every job $J_{i,j}^V$ has its own WCER $C_{i,j}^V$. $\max_j C_{i,j}^V$ is denoted as C_i^V. Unlike the sporadic task model, each VPP task τ_i^V releases its first job $J_{i,1}^V$ at time $t = 0$. Afterwards, each job $J_{i,j+1}$ is released *exactly* $T_{i,j}^V = C_{i,j}^V / u_i^V$ time units after the release of $J_{i,j}^V$ for $j \geq 1$. C_i^V / u_i^V is denoted as T_i^V.

It was shown in App. A of [8] that sporadic task systems are special cases of VPP task systems. For example, in Fig. 5, the first timeline represents a sporadic release of jobs by some task τ_i and the second timeline represents a VPP release of jobs by a VPP task τ_i^V with $u_i = u_i^V$ and $C_i = C_i^V$. When the separation between jobs of τ_i is greater than T_i, arbitrarily small VPP jobs with an execution requirement of 0.0 are inserted between the gap in releases, thereby making the second timeline a valid VPP release sequence. Thus, the sporadic release of jobs is also a VPP release of jobs.

Because sporadic task systems are special cases of VPP task systems, our proof obligation is to show that our tardiness bounds apply under the VPP model. The tardiness bounds under IA-GEDF depend only on properties of HP-LAG schedulers, which in turn depend only on the properties in Sec. 3. The fact that these properties hold under the VPP model with assumption (W) (with some reinterpretation of parameters, e.g., substituting u_i with u_i^V and C_i with $C_{i,j}^V$ in proofs) was shown in App. A of [8]. Hence, because removing (W)



(a) Affinity graph. **(b)** A SRT schedule (with a maximum tardiness of 0.5) of a periodic instance of the task system in Theorem 39. The height of a scheduling interval represents the speed of the allocated processor.

Figure 6 Task system considered in the proof of Theorem 39.

does not increase tardiness, Corollary 38 holds with VPP task systems (substituting C_{max} with $\max_i C_i^V$ in (15)). If the VPP task system is also a sporadic system, as in Fig. 5, then $\max_i C_i^V = \max_i C_i = C_{max}$. Thus, the tardiness bound in Corollary 38 for sporadic tasks is exactly as written in (15).

8 Problems with Extending to the Uniform Model with Affinities

Of the models listed in Sec. 2.1, we have not addressed Uniform-Aff and Unrelated. In this section, we explain why extending our proof techniques to these models is difficult. The exact proof strategy used in this work cannot be directly applied to the more general models in Sec. 2.1 because HP-LAG may not hold.

▶ **Theorem 39.** *No non-fluid scheduler always satisfies HP-LAG under Uniform-Aff.*

Proof. We prove the theorem by constructing a feasible task system, deadline ordering, and Uniform-Aff platform for which no scheduler satisfies HP-LAG. Consider the task system $\tau = \{\tau_1, \tau_2\}$ with $(C_1, T_1) = (3, 2)$ and $(C_2, T_2) = (4, 4)$. Then, $u_1 = 1.5$ and $u_2 = 1$. τ runs on two processors $\pi = \{\pi_1, \pi_2\}$ with $s_1 = 2$ and $s_2 = 1$. G_τ is illustrated in Fig. 6a.

We know this system is feasible from the schedule in Fig. 6b, which contains a timeline for each task that describes what processor, if any, schedules the task at any time instant. The schedule repeats every four time units. This schedule provides six units of execution to τ_1 and four units of execution to τ_2 every four time units. Because the schedule provides execution to each task at a long-run rate equal to its utilization ($6/4 = 3/2 = u_1$ and $4/4 = u_2$), both tasks have bounded tardiness.

Let the deadline ordering at some time instant t be $d_1(t) < d_2(t)$ (e.g., $t = 0$). Suppose some non-fluid scheduling algorithm is HP-LAG-compliant at t. By Def. 4, we have that $(t, t + \delta)$ is a continuous scheduling interval for some $\delta > 0$. Note that τ' as defined in HP-LAG for this task system may be either $\{\tau_1\}$ or $\{\tau_1, \tau_2\}$ at time t. HP-LAG states for these two task subsets that $\forall t' \in (t, t + \delta)$:

$$\mathsf{LAG}(\{\tau_1\},\ t') \leq \mathsf{LAG}(\{\tau_1\},\ t) \tag{21}$$

$$\mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') \leq \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t) \tag{22}$$

We show that τ_1 must execute on π_1 over $(t, t + \delta)$ by contradiction. Consider otherwise that there exists a time instant in $(t, t + \delta)$ where τ_1 executes on π_2. Because $(t, t + \delta)$ is a continuous scheduling interval, by Def. 2, τ_1 executes on π_2 for all $t' \in (t, t + \delta)$. Thus,

$\mathsf{LAG}(\{\tau_1\},\ t') =$ {by Def. 7}

$$
\begin{aligned}
& \mathsf{LAG}(\{\tau_1\},\ t) + A(\mathcal{I}, \tau_1, t, t') - A(\mathcal{R}, \tau_1, t, t') \\
= & \{\text{by Defs. 3 and 6}\} \\
& \mathsf{LAG}(\{\tau_1\},\ t) + u_1(t' - t) - A(\mathcal{R}, \tau_1, t, t') \\
= & \{\text{by Def. 3 and the assumption that } \tau_1 \text{ is scheduled on } \pi_2\} \\
& \mathsf{LAG}(\{\tau_1\},\ t) + u_1(t' - t) - s_2(t' - t) \\
= & \{u_1 = 1.5 \text{ and } s_2 = 1\} \\
& \mathsf{LAG}(\{\tau_1\},\ t) + 0.5(t' - t) \\
> & \{t' - t > 0 \text{ by definition}\} \\
& \mathsf{LAG}(\{\tau_1\},\ t),
\end{aligned}
$$

which contradicts (21). A similar contradiction arises when τ_1 is not executing for any instant in $(t, t+\delta)$, so any HP-LAG compliant scheduler *must* schedule τ_1 on π_1 during $(t, t+\delta)$. Scheduling τ_1 on π_1 means τ_2 is not scheduled over this interval, because π_1 is the only processor in τ_2's affinity mask.

Consider the LAG of $\{\tau_1, \tau_2\}$ for all $t' \in (t, t+\delta)$ given that τ_1 executes on π_1 and τ_2 is not executing over this interval. Through reasoning similar to that above, we can conclude $\mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') = \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t) + 0.5(t' - t) > \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t)$, which contradicts (22). Because no non-fluid scheduler can simultaneously satisfy (21) and (22), no non-fluid scheduler can satisfy HP-LAG for this feasible task system. ◀

9 Conclusion

We have derived the first ever GEDF tardiness bounds that are polynomial in the number of processors under Uniform. We have also derived for the first time generalized GEDF scheduling rules that are provably SRT-optimal under Identical-Aff. This result shows that the open problem mentioned by Peter Zijlstra and Luca Abeni can be resolved by altering `SCHED_DEADLINE` to be HP-LAG-compliant. In App. A, we have provided an algorithm that implements our generalized GEDF scheduling rules with lower time complexity per scheduling event than `SCHED_DEADLINE`, given some preprocessing.

Note that the proofs in Sec. 4 only require that the values of β satisfy certain linear constraints. Thus, lower β values than ours can be derived using linear programming (though the constraint set grows exponentially with the task count). This suggests that the properties in Sec. 3 are sufficient to derive tighter analytical tardiness bounds than ours.

In future work, we will investigate dynamic task systems, where tasks may enter or exit the system and affinity masks may change at runtime. The interaction of affinity masks with dynamic task systems is particularly relevant to `SCHED_DEADLINE`, as admission control with affinity masks is currently broken. We plan to investigate what restrictions must be placed on these dynamics to avoid compromising bounded tardiness, as done in prior work [4] on GEDF without affinity masks. We are also interested in how overhead accounting and non-preemptive sections might be handled as well as how algorithms that satisfy IA-GEDF might be constructed with lower time complexity than in App. A.

The SRT-optimality of GEDF on more general platforms also remains an open problem. We have shown via a counterexample that HP-LAG, a property that applies to both Uniform and Identical-Aff individually, does not hold when the models are combined. It is unknown whether a weaker version of HP-LAG exists that applies to the more general processor models while still being sufficient to bound tardiness. If not, these open problems may require new proof techniques.

References

1 Luca Abeni. SCHED_DEADLINE: a real-time CPU scheduler for Linux. 2nd TuTor at the 38th IEEE Real-Time Systems Symposium, 2017. URL: https://tutor2017.inria.fr/sched_deadline/.

2 Luca Abeni et al. Deadline task scheduling. Linux kernel documentation, 2018. URL: https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.txt.

3 Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957.

4 Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Task reweighting under global scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 123–167, 2006.

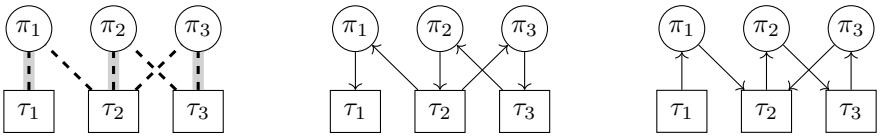
5 UmaMaheswari C. Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

6 Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the 22th IEEE Real-Time Systems Symposium*, pages 183–192, 2001.

7 Sergey Voronov and James H. Anderson. AM-Red: An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 408–420, 2018.

8 Kecheng Yang and James H. Anderson. On the soft real-time optimality of global EDF on uniform multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 319–330, 2017.

9 Peter Zijlstra. An update on real-time scheduling on Linux. Keynote talk at the 29th Euromicro Conference on Real-Time Systems, 2017. URL: https://www.ecrts.org/archives/index.php?id=284.



(a) G_τ and M **(b)** H_τ **(c)** $\overline{H}_\tau$

Figure 7 Reference figure for Exs. 42 and 44.

A Exact IA-GEDF Scheduling Algorithm

In order to show that IA-GEDF is possible to implement efficiently, we present Alg. 2, which computes IA-GEDF task-to-processor assignments with time complexity $O(mn)$ (and $O(m + \log n)$ with some preprocessing) per scheduling event. We first define a special structure used by Alg. 2.

▶ **Definition 40.** *Let the task-to-processor assignment at time instant t be given by the matching M. The runtime affinity graph (H_τ) is defined for time instant t and has the same vertices as the AG G_τ, but has directed edges. The edges of H_τ at t are defined as follows.*

- *$\tau_i \to \pi_j$ is an edge if and only if $\pi_j \in \alpha_i$ and $(\tau_i, \pi_j) \notin M$.*
- *$\pi_j \to \tau_i$ is an edge if and only if $(\tau_i, \pi_j) \in M$.*

▶ **Definition 41.** *For a directed edge e, we denote a directed edge with the same vertices in reversed order as $\overline{e}$. For a path $p = (v_1, v_2, \ldots, v_k)$, we denote the path $(v_k, v_{k-1}, \ldots, v_1)$ as $\overline{p}$. $\overline{H}_\tau$ denotes the reversed H_τ: $e \in \overline{H}_\tau$ if and only if $\overline{e} \in H_\tau$.*

▶ **Example 42.** H_τ for G_τ in Fig. 7a is illustrated in Fig. 7b. $\overline{H}_\tau$ for the same G_τ is illustrated in Fig. 7c.

▶ **Definition 43.** *We say that a vertex v_1 is reachable from another vertex v_2 in a graph if there exists a path over the edges of the graph from v_2 to v_1. Such a path can be found by a variety of linear-time algorithms, e.g., breadth-first search (BFS).*

▶ **Example 44.** In the H_τ from Fig. 7b, τ_1 is reachable from every other vertex, while only τ_1 can be reached from π_1.

The following lemma shows that H_τ is defined in a way such that the reachability between special types of vertices in H_τ is equivalent to the existence of an alternating path in the corresponding AG G_τ.

▶ **Lemma 45.** *If processor π_j is reachable from an unscheduled task τ_i in H_τ via a path p, then p is an alternating path in G_τ. If an unscheduled task τ_i is reachable from a processor π_j in $\overline{H}_\tau$ via a path p, then $\overline{p}$ is an alternating path in G_τ.*

Proof. If π_j is reachable from τ_i, then there is a path of directed edges from τ_i to π_j. By Def. 40, every edge from a task to a processor is not in M and every edge from a processor to a task is in M. Hence, by Def. 28, every path from any task to any processor is alternating, including the path from τ_i to π_j.

If instead τ_i is reachable from π_j by path p in $\overline{H}_\tau$, then π_j is reachable from τ_i by $\overline{p}$ in H_τ, in which case the same reasoning as above applies. ◀

We are now ready to present Alg. 2, which contains two functions, `JobReleased` and `JobCompleted` to be executed per job release and completion, respectively.

1 **Function** `JobReleased`($J_{i,j}$, H_τ, t): `// a job` $J_{i,j}$ `is released at time` t
2 **if** $J_{i,j}$ *is not ready* **then**
3 **return** *Unchanged* H_τ;
4 $d_i(t) \leftarrow d_{i,j}$;
5 run BFS starting from task τ_i of $J_{i,j}$ over H_τ;
6 **if** *no idle processor* π_k *is found* **then**
7 $\tau_\ell \leftarrow$ task with the latest deadline among all reached scheduled tasks;
8 $\pi_k \leftarrow$ processor that scheduled τ_ℓ;
9 **else**
10 $\pi_k \leftarrow$ an idle processor;
11 **if** π_k *is idle OR* (π_k *is busy AND* $d_i(t) \prec d_\ell(t)$) **then**
12 perform scheduling cascade via an alternating path from τ_i to π_k;
13 update H_τ (and $\overline{H}_\tau$) edges along the alternating path;
14 **return** updated H_τ;
15 **Function** `JobCompleted`($J_{i,j}$, H_τ, t): `// a job` $J_{i,j}$ `finishes at time` t
16 **if** *Job* $J_{i,j+1}$ *is released* **then**
17 $d_i(t) \leftarrow d_{i,j+1}$;
18 **else**
19 $d_i(t) \leftarrow \infty$;
20 $\pi_k \leftarrow$ processor where $J_{i,j}$ was scheduled;
21 run BFS starting from π_k over $\overline{H}_\tau$;
22 $\tau_h \leftarrow$ task with the earliest deadline among all reachable unscheduled tasks;
23 **if** $d_h(t) \prec d_i(t)$ **then**
24 unschedule τ_i;
25 perform scheduling cascade via an alternating path from τ_h to π_k;
26 update H_τ (and $\overline{H}_\tau$) edges along the alternating path;
27 **else if** $d_i(t) = \infty$ **then** `// means` τ_i `has no ready jobs (see Def. 1)`
28 unschedule τ_i;
29 **return** updated H_τ;

Algorithm 2: IA-GEDF assignment computation for a scheduling event.

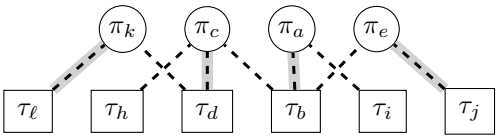
▶ **Lemma 46.** *Alg. 2 is well-defined, i.e., it is guaranteed that every line is possible to execute.*

Proof. All lines of Alg. 2 except 12 and 25 present simple actions or a BFS over a graph, so they are well-defined. The precondition for line 12 is that there exists a processor π_k that is reachable from τ_i in H_τ and that is either idle or schedules a task with later deadline. Thus, by Lemma 45 and Def. 31, a scheduling cascade can be performed.

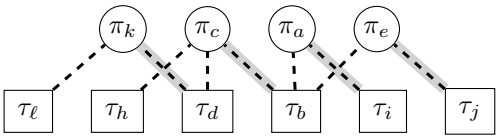
Likewise, the precondition for line 25 is that processor π_k schedules task τ_i such that task τ_h is reachable from π_k and τ_h has an earlier deadline than τ_i. Thus, by Lemma 45 and Def. 31, a scheduling cascade can be performed. ◀

▶ **Lemma 47.** *If the task-to-processor assignments satisfy IA-GEDF before a job* $J_{i,j}$ *is released, then the new task-to-processor assignments after executing* `JobReleased` *on* $J_{i,j}$ *satisfy IA-GEDF.*

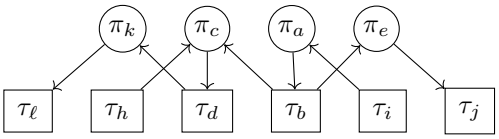
Proof. We use Fig. 8 to illustrate different proof stages. Denote the matchings that represent the before and after task-to-processor assignments as M and M', respectively (matching



(a) Matching M.



(b) Matching M'.



(c) H_τ before Alg. 2 runs.

Figure 8 Lemma 47 reference figure.

examples can be found in Figs. 8a and 8b)..

Case 1: No scheduled task with a larger deadline than $d_i(t)$ was found by a BFS traversal that starts from τ_i. In this case, the task-to-processor assignments remains the same, thus $M = M'$. Because M satisfies IA-GEDF by the lemma statement, so too does M'.

Case 2: A scheduling cascade via an alternating path p that starts from τ_i and ends with a processor π_k was performed. This resulted in some task τ_ℓ being descheduled from this processor. For example, in Fig. 8c, task τ_ℓ was found through the path $(\tau_i, \pi_a, \tau_b, \pi_c, \tau_d, \pi_k, \tau_\ell)$.

Suppose that M' does not satisfy IA-GEDF. Then, a scheduling cascade can be performed under M'. Denote the path of this cascade as $p' = (\tau_h, ..., \pi_e)$ (e.g., the path $(\tau_h, \pi_c, \tau_b, \pi_e)$ in Fig. 8b). Denote the task that is scheduled on π_e in M' as τ_j. (If π_e is instead an idle processor, then we consider π_e as a processor that executes a task τ_j, not in τ, with a deadline equal to ∞ to avoid having to duplicate the reasoning given here for that case. A task with infinite deadline would permit the same scheduling cascades as an idle processor, by Def. 31.) By Def. 31,

$$d_h(t) \prec d_j(t), \tag{23}$$

because a scheduling cascade over p' is possible under M'. Because M' represents the task-to-processor assignments after τ_i releases a job,

$$(\tau_j, \pi_e) \in M'. \tag{24}$$

(24) implies that the directed edge (π_e, τ_j) is in H_τ.

If p' does not have any edges from M'/M, then p' is an alternating path under M, and hence a scheduling cascade was possible in G_τ with M before the execution of `JobReleased`, which contradicts the lemma statement. Thus, p' has at least one edge from M'/M. Denote the last (by order of appearance in p') common edge of p and p' in M'/M as (τ_b, π_c). Denote the first (by the same ordering) common edge of p and p' in M'/M appearing in p' as (τ'_b, π'_c). Note that these edges can be equal, as in Fig. 8. Thus, we can write p' as

$$p' = (\overbrace{\tau_h, ..., \pi'_c}^{\text{no edges in } M'/M, \text{ ends with an edge not in } M}, \tau'_b, ..., \pi_c, \underbrace{\tau_b, ..., \pi_e}_{\text{no edges in } M'/M, \text{ starts with an edge not in } M})$$

(e.g., the path $(\overbrace{\tau_h, \pi_c}^{\text{no edges in } M'/M}, \underbrace{\tau_b, \pi_e}_{\text{no edges in } M'/M})$ in Fig. 8b)

(p' is an alternating path in M', so by Def. 28, the processor π_c would appear first among π_c and τ_b).

M' was the result of a scheduling cascade via an alternating path p, so the set of all odd edges of p is M'/M, and the set of all even edges is M/M' (we consider the edge with vertex τ_i as the first edge). Thus,

$$p = (\overbrace{\tau_i, ..., \tau_b}^{\text{ends with an edge in } M\text{: } (\tau_b,\pi_c)\in M'/M}, \pi_c, ..., \pi_k) \qquad \text{(e.g., the path } (\tau_i, \pi_a, \tau_b, \pi_c, \tau_d, \pi_k) \text{ in Fig. 8a)}$$

(p is an alternating path in M, so by Def. 28, the task τ_b would appear first among π_c and τ_b). The same reasoning applies for the edge (τ'_b, π'_c).

$$p = (\tau_i, ..., \tau'_b, \underbrace{\pi'_c, ..., \pi_k}_{\text{starts with an edge in } M\text{: } (\tau'_b,\pi'_c)\in M'/M}) \qquad \text{(e.g., the path } (\tau_i, \pi_a, \tau_b, \pi_c, \tau_d, \pi_k) \text{ in Fig. 8a)}$$

This reasoning allows us to claim that τ_j was reachable in H_τ from τ_i by the path

$$(\overbrace{\tau_i, ..., \underbrace{\tau_b}_{\text{from } p'}}^{\text{from } p}, ..., \overbrace{\pi_e, \tau_j}^{\text{from (24)}}). \qquad \text{(e.g., the path } (\overbrace{\tau_i, \pi_a, \underbrace{\tau_b}_{\text{from } p'}}^{\text{from } p}, \pi_e, \tau_j) \text{ in Fig. 8c).}$$

This path can be constructed because $(\tau_b, ..., \pi_e)$ is the part of p' with no edges in M', and its edges are alternating to be not in M and in M, where the first one is not in M.

Because task τ_ℓ was the task with the latest deadline among all tasks reachable from τ_i, $d_j(t) \prec d_\ell(t)$. With (23), this implies $d_h(t) \prec d_\ell(t)$. However, a path

$$(\overbrace{\tau_h, ..., \underbrace{\pi'_c}_{\text{from } p}, ..., \pi_k}^{\text{from } p'}). \qquad \text{(e.g., the path } (\overbrace{\tau_h, \underbrace{\pi_c}_{\text{from } p}, \tau_d, \pi_k}^{\text{from } p'} \text{ in Fig. 8a).}$$

exists as an alternating path in M. This path can be constructed because $(\tau_h, ..., \pi'_c)$ is the part of p' with no edges in M', and its edges are alternating to be not in M and in M, where the first one not in M. Because $d_h(t) \prec d_\ell(t)$, it was possible to perform a scheduling cascade in M before the execution of `JobReleased`, which contradicts the lemma statement.

Case 3: A scheduling cascade via an alternating path that starts from τ_i and ends with a processor π_k was performed, and no task was descheduled from π_k. If we consider an idle processor as a processor that executes a task (not in τ) with deadline equal to ∞, then the behavior of the scheduler reduces to that of Case 2 (a task with infinite deadline would permit the same scheduling cascades as τ_ℓ of Case 2). Thus, the reasoning from Case 2 also applies to Case 3. ◀

▶ **Lemma 48.** *If the task-to-processor assignments satisfied IA-GEDF before a job $J_{i,j}$ completes, then the new task-to-processor assignments after executing `JobReleased` on $J_{i,j}$ satisfy IA-GEDF.*

Proof. The proof is analogous to the proof of Lemma 47. Exactly the same reasoning is applied with $\overline{H}_\tau$ instead of H_τ, and processor π_k instead of task τ_i. ◀

Applying Lemmas 47 and 48 together yields the following.

▶ **Corollary 49.** *If the functions of Alg. 2 are applied for every scheduling event, then the resulting schedule satisfies IA-GEDF at every time instant.*

In this paper, we consider the time complexity of our algorithm per scheduling event and per scheduling time instant.

▶ **Definition 50.** *A scheduling time instant is a time instant t such that at least one scheduling event happens at t. Note that several scheduling events can happen at the same scheduling time instant (e.g., several tasks simultaneously release new jobs).*

▶ **Lemma 51.** *If the AG of a feasible task system has V vertices and E edges, then the time complexity of Alg. 2 is $O(E)$ per scheduling event. Expressed in terms of m and n, the time complexity of Alg. 2 is $O(mn)$ per scheduling event*

Proof. Let V be the number of vertices in the AG of a feasible task system. Note that the affinity mask of every task must contain at least one processor, so $V \leq E$. The BFS traversal dominates the time of Alg. 2, because all other lines are executed in O(length of a path), where the length of path is limited by $2V \leq 2E$. The BFS has time complexity $O(V + E) = O(E)$. The number of edges in H_τ and $\overline{H}_\tau$ is at most mn (when tasks are globally scheduled), hence, a single BFS traversal of any function in Alg. 2 has time complexity $O(mn)$. ◀

Because the number of scheduling events at one scheduling time instant does not exceed $O(n)$ (every task can release at most one job and complete at most one job), the previous lemma yields the following.

▶ **Corollary 52.** *Alg. 2 has the time complexity $O(mn^2)$ per scheduling instant.*

The time complexities, provided by Lemma 51 and Corollary 52, can be pessimistic under some contexts (e.g., $m \ll n$). However, as we show next, this time complexity can be significantly lowered by applying certain pre-processing steps offline. Because the BFS traversal dominates the time of Alg. 2, the algorithm performs better when the number of edges in G_τ is relatively small. In [7], Voronov and Anderson presented an $O(m^2n^2)$ affinity-reduction algorithm (Alg. 4 in [7]), which removes edges from G_τ until G_τ becomes acyclic while preserving the feasibility of a given task system (Theorem 9 in [7]). Following [7], we call a task *migrating* if its affinity mask contains more than one processor and *fixed* otherwise. G_τ produced by this affinity-reduction technique as at most $(m+n-1)$ edges and at most $(m-1)$ migrating tasks (Theorem 3 in [7]). Hence, this affinity-reduction algorithm can be used as preprocessing step offline to reduce Alg. 2's time complexity. The following lemma shows an additional way to reduce the time complexity.

▶ **Lemma 53.** *Consider two fixed pending tasks τ_h and τ_ℓ, which share a single-processor affinity mask. If $d_h(t) \prec d_\ell(t)$, then τ_ℓ is not scheduled.*

Proof. We prove the lemma by contradiction. Denote the matching that corresponds to the task-to-processor assignments as M. Consider the case when τ_ℓ has an assigned processor π_l after executing a function of Alg. 2. The task-to-processor assignments respect all affinity-mask restrictions, so $\pi_l \in \alpha_{\tau_\ell}$. Because α_{τ_ℓ} contains only one processor, we have $\alpha_{\tau_\ell} = \{\pi_l\}$. Thus, τ_h is not scheduled because its mask contains only one processor, which is assigned to task τ_ℓ. Consider the single-edge path $p = (\tau_h, \pi_l)$. Vertex τ_h is unmatched, so (τ_h, π_l) is not in the matching M. Thus, p is alternating for M in G_τ. Because $(\tau_\ell, \pi_l) \in M$ and $d_h(t) \prec d_\ell(t)$, a scheduling cascade via p can be performed (see Alg. 1). This last statement contradicts IA-GEDF by Lemmas 47 and 48. Thus, τ_ℓ is not scheduled. ◀

The last lemma states that among all tasks that share the same single-processor mask, only the one with the earliest deadline may be scheduled. Hence, all other tasks with this mask can be excluded from consideration. Thus, we can use two modifications to speed up Alg. 2:

1. Use Alg. 4 from [7] to perform offline affinity-mask reductions. Note that this modification performs certain pre-processing steps, which implies an offline time complexity $O(m^2n^2)$.
2. Let $\tau' \subseteq \tau$ denote the set of fixed tasks on processor π_k and let $\tau_h \in \tau'$ denote the task with earliest deadline in this set. When jobs of tasks in $\tau'/\{\tau_h\}$ release or complete, do not perform a BFS (or scheduling cascade) for these jobs.

In order to be able to find a task with the earliest deadline among fixed tasks on a processor in Modification 2, we need to maintain a min-heap of fixed tasks ordered by deadline for every processor. The time complexitiy for inserting or removing a task to or from a heap is $O(\log n)$.

▶ **Theorem 54.** *Modified as above, Alg. 2 has a time complexity $O(m + \log n)$ per scheduling event and an offline time complexity $O(m^2n^2)$.*

Proof. By Modification 1, we ensure that the number of edges in AG is at most $(m + n - 1)$. By Modification 2, we perform BFS traversals and scheduling cascades for at most $2m - 1$ tasks (at most $m - 1$ migrating tasks and one fixed task per processor). By Lemma 53, we do not have to consider any other task at a scheduling event. Because, for the system to be feasible, every task must have at least one processor in its affinity mask and we have dropped at least $n - (2m - 1)$ tasks from consideration, the number of edges we need to consider in the AG G_τ is at most

$$(m + n - 1) - (n - (2m - 1)) = 3m = O(m).$$

Thus, by Lemma 51, the time complexity of Alg. 2 with Modifications 1 and 2 is $O(m)$.

Unfortunately, maintaining Modification 2 has time complexity $O(\log n)$ at runtime per scheduling event due to heap operations. Hence, the total time complexity per scheduling event is $O(m + \log n)$. Note that Modification 1 only has an offline time complexity $O(m^2n^2)$. ◀

▶ **Corollary 55.** *Modified as above, Alg. 2 has time complexity $O(m^2 + n \log n)$ per scheduling instant.*

Proof. At every scheduling instant, at most $(m + n)$ scheduling events may occur (m jobs can complete and n jobs can be released). Because we do not want to execute the functions of Alg. 2 $(m + n)$ times, we add an additional change to Alg. 2 for Modification 2: we collect the scheduling events of all fixed tasks and insert (resp., remove) these tasks into (resp., from) a heap of the corresponding processor. This requires $O(n \log n)$ time complexity (there are $O(n)$ insertion/deletion operations).

After that, we discard all job release events from fixed tasks that do not have the earliest deadline among the fixed tasks on each processor (such tasks cannot be scheduled by Lemma 53). The remaining number of scheduling events is at most $O(m)$ (for migrating tasks) + $O(m)$ (for job releases of fixed tasks) + $O(m)$ (for job completions of fixed tasks) = $O(m)$.

Applying the modified Alg. 2 to $O(m)$ tasks results in $O(m^2 + m \log n)$ time complexity. Thus, the total time complexity per scheduling event is $O(m^2 + m \log n) + O(n \log n) = O(m^2 + n \log n)$, assuming $n > m$.

Note that the $O(n \log n)$ term can be decreased to $O(n)$ if deadlines are stored in variables with fixed length (like floats or doubles). In this case, we can apply radix sort on arrays of deadlines, thereby building each min-heap in linear time (over a presorted array). This approach results in $O(m^2 + n)$ time complexity per scheduling instant. ◀

B A Formal Extension to Sporadic Task Model

In this appendix, we present the formal proofs that assumptions (P) and (W) can be removed without compromising Corollary 38.

Removing assumption (W). Note that all lemmas in this appendix consider either a single job, or jobs from a set of tasks where, for each task, earlier jobs have earlier deadlines. Thus, these lemmas may be applied to periodic, sporadic, and VPP task systems.

Recall that under (W), the execution requirement of every job $J_{i,j}$ is C_i. Let $\mathcal{R}$ be the original schedule where all jobs execute to their tasks' WCERs and $\mathcal{R}'$ be a schedule with the execution reduction (both schedules produced under IA-GEDF). Consider a job $J_{i,j}$ of task τ_i such that no job with an earlier deadline exists that completes at a later time in $\mathcal{R}'$ than in $\mathcal{R}$. Recall Fig. 4, which shows a schedule that obeys (W) (inset b) and a schedule where job $J_{1,1}$'s execution requirement is reduced by 1.0 (inset c). Observe that in Fig. 4c, every job becomes ready no later than in Fig. 4b. We can formalize this by proving that $J_{i,j}$ is ready in $\mathcal{R}'$ no later than in $\mathcal{R}$ (Lemma 56), and that this implies that Theorem 20 applies without (W) (Lemma 57).

▶ **Lemma 56.** *If at time instant t job $J_{i,j}$ is ready or completed in $\mathcal{R}$, then $J_{i,j}$ is ready or completed in $\mathcal{R}'$.*

Proof. By the definition of $J_{i,j}$, all jobs with a deadline earlier than $J_{i,j}$ complete no later in $\mathcal{R}'$ than in $\mathcal{R}$. Thus, job $J_{i,j-1}$ (if it exists) completes in $\mathcal{R}'$ no later than in $\mathcal{R}$. As a result, job $J_{i,j}$ becomes ready in $\mathcal{R}'$ no later than in $\mathcal{R}$. ◀

▶ **Lemma 57.** *$J_{i,j}$ completes in $\mathcal{R}'$ no later than in $\mathcal{R}$.*

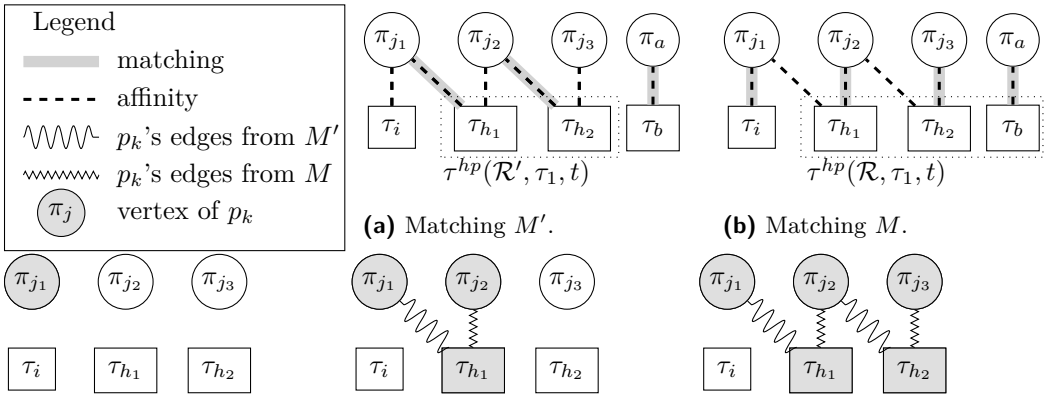
Proof. By Lemma 56, $J_{i,j}$ is ready in $\mathcal{R}'$ no later than in $\mathcal{R}$. Assume that for any time instant t where $J_{i,j}$ is scheduled in $\mathcal{R}$, $J_{i,j}$ is also scheduled or completed in $\mathcal{R}'$ (we prove this fact below in Lemma 61). Because $J_{i,j}$ receives at least as much execution in $\mathcal{R}'$ as in $\mathcal{R}$ at any given time instant, it is impossible for $J_{i,j}$ to complete later in $\mathcal{R}'$ than in $\mathcal{R}$. This contradicts the definition of $J_{i,j}$ as the first job that completes later in $\mathcal{R}'$, so no job completes later in $\mathcal{R}'$. ◀

The lemma above shows that no job's completion time shifts later in time (in $\mathcal{R}'$ compared to $\mathcal{R}$) unless a prior job (by deadline) also shifted later. Hence, with the lemma applied inductively, the reduction of the execution requirement of a single job (which shifts its completion time earlier in time) does not shift the completion time of any job later in time. This statement applied inductively implies that we can reduce the execution requirement of any set of jobs and no job will complete later in $\mathcal{R}'$ than in $\mathcal{R}$ as a result. This allows us to remove assumption W.

In the proof of Lemma 57, we assumed that Lemma 61 was true. While the lemma statement may see obvious, proving it is nontrivial. In order to do so, we need several new definitions.

▶ **Definition 58.** *Let $\tau^{hp}(\mathcal{S}, \tau_i, t) \subset \tau$ be the set of tasks such that for task τ_i, schedule $\mathcal{S}$, and time instant t, $\tau_h \in \tau^{hp}(\mathcal{S}, \tau_i, t)$ implies $d_h(t) \prec d_i(t)$ in $\mathcal{S}$.*

By the definition of $J_{i,j}$, all jobs with earlier deadlines completed in $\mathcal{R}'$ no later than in $\mathcal{R}$. Hence, while $J_{i,j}$ is ready, there is no job with higher priority than $J_{i,j}$ in $\mathcal{R}'$ that did not also have higher priority in $\mathcal{R}$.



(a) Matching M'. **(b)** Matching M.

(c) Path p_1 (base case). **(d)** Path p_2 ((27) with $k = 2$). **(e)** Path p_3 ((27) with $k = 3$).

Figure 9 Example graphs illustrating some of the steps in proving Lemma 61.

▶ **Lemma 59.** *For any time instant t, where $J_{i,j}$ is ready in $\mathcal{R}$ and not completed in $\mathcal{R}'$, we have $\tau^{hp}(\mathcal{R}', \tau_i, t) \subseteq \tau^{hp}(\mathcal{R}, \tau_i, t)$.*

Proof. As shown in Lemma 56, $J_{i,j}$ is ready or completed in $\mathcal{R}'$ at time instant t. The second case ($J_{i,j}$ is completed in $\mathcal{R}'$) is invalid by the lemma statement, hence we need only consider the case that $J_{i,j}$ is ready.

We will prove the lemma by showing $\forall \tau_h \in \tau : \tau_h \in \tau^{hp}(\mathcal{R}', \tau_i, t) \Rightarrow \tau_h \in \tau^{hp}(\mathcal{R}, \tau_i, t)$. To distinguish between task deadlines in $\mathcal{R}$ and $\mathcal{R}'$, we let $d_k(t)$ and $d'_k(t)$ denote the deadline of task τ_k at time t in $\mathcal{R}$ and $\mathcal{R}'$, respectively. If $\tau_h \in \tau^{hp}(\mathcal{R}', \tau_i, t)$, then in $\mathcal{R}'$, we have

$$d'_h(t) \prec d'_i(t). \tag{25}$$

Note that $d'_i(t) = d_{i,j}$ because $J_{i,j}$ is ready at t in $\mathcal{R}'$.

Because jobs with earlier deadlines than $J_{i,j}$ complete in $\mathcal{R}'$ no later than in $\mathcal{R}$, the ready job of τ_h at time t in $\mathcal{R}$ is either the same job that is ready at t in $\mathcal{R}'$ or an earlier job of τ_h. Hence,

$$d_h(t) \leq d'_h(t). \tag{26}$$

Thus, by (25) and (26), $d_h(t) \prec d'_i(t) = d_i(t)$ because $J_{i,j}$ is ready at t in both $\mathcal{R}$ and $\mathcal{R}'$ and has the same deadline under both schedules. Thus, for any time instant t where $J_{i,j}$ is ready, and for any task $\tau_h \in \tau^{hp}(\mathcal{R}', \tau_i, t)$, we have $\tau_h \in \tau^{hp}(\mathcal{R}, \tau_i, t)$. Therefore, for any time instant t where $J_{i,j}$ is ready, we have $\tau^{hp}(\mathcal{R}', \tau_i, t) \subseteq \tau^{hp}(\mathcal{R}, \tau_i, t)$. ◀

▶ **Definition 60.** *For paths p_1 and p_2 in a graph such that the last vertex of p_1 is the first vertex of p_2, we denote the concatenation of p_1 and p_2 as $p_1 + p_2$.*

We are now ready to prove Lemma 61, which completes the proof of Lemma 56.

▶ **Lemma 61.** *If $J_{i,j}$ is scheduled in $\mathcal{R}$ at time t, then $J_{i,j}$ is scheduled or completed at time t in $\mathcal{R}'$.*

Proof. Because $J_{i,j}$ is scheduled in $\mathcal{R}$, $J_{i,j}$ is ready in $\mathcal{R}$, and, by Lemma 56, $J_{i,j}$ is ready or completed in $\mathcal{R}'$. The case, where $J_{i,j}$ is completed in $\mathcal{R}'$ satisfies the lemma statement. Thus, we need only consider the case, where $J_{i,j}$ is ready in $\mathcal{R}'$. In this case, by Lemma 59,

$$\tau^{hp}(\mathcal{R}', \tau_i, t) \subseteq \tau^{hp}(\mathcal{R}, \tau_i, t).$$

To simplify our reasoning, in the rest of the proof, instead of considering the scheduling of job $J_{i,j}$, we consider the scheduling of task τ_i (these considerations are actually identical because τ_i has a ready job $J_{i,j}$ at time instant t in both $\mathcal{R}$ and $\mathcal{R}'$).

We will prove the lemma by contradiction. Suppose otherwise that at time t, τ_i is scheduled in $\mathcal{R}$ and not $\mathcal{R}'$. We will show that under this assumption, we can construct an arbitrarily long path of distinct vertices in G_τ, thereby contradicting the fact that τ and π are finite. Let M' and M denote the matchings associated with the task-to-processor assignments at time t for $\mathcal{R}'$ and $\mathcal{R}$, respectively. An example of these matchings is illustrated in Fig. 9a and Fig. 9b.

We will prove via induction that for $k = [1, \infty)$, there exists path

$$p_k = (\pi_{j_1}, \tau_{h_1}, \pi_{j_2}, \tau_{h_2}, \pi_{j_3}, \ldots, \tau_{h_{k-1}}, \pi_{j_k}) \tag{27}$$

in G_τ such that the following invariants hold.

$$(\tau_i, \pi_{j_1}) \in M \tag{28}$$

$$\forall \ell \in [1, k-1] : (\tau_{h_\ell}, \pi_{j_\ell}) \in M' \tag{29}$$

$$\forall \ell \in [1, k-1] : (\tau_{h_\ell}, \pi_{j_{\ell+1}}) \in M \tag{30}$$

$$\text{The vertices of } p_k \text{ are distinct from each other and } \tau_i \tag{31}$$

(29) and (30) for path p_3 are illustrated in Fig. 9a and Fig. 9b, respectively. Note that $\{(\pi_{j_1}, \tau_{h_1}), (\pi_{j_2}, \tau_{h_2})\} \subseteq M'$ in Fig. 9a and $\{(\tau_{h_1}, \pi_{j_2}), (\tau_{h_2}, \pi_{j_3})\} \subseteq M$ in Fig. 9b, and hence path p_3 in Fig. 9e is made up of edges alternatingly in M' and M, thereby satisfying (29) and (30).

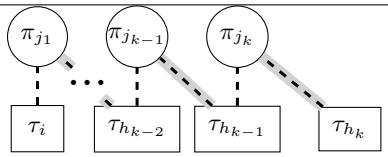
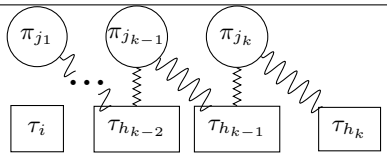
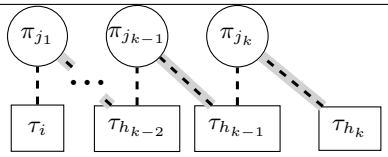
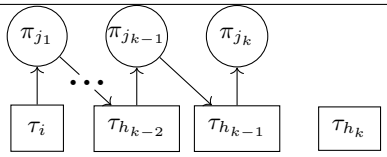
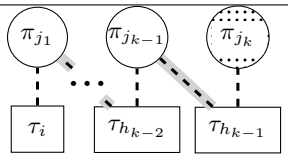
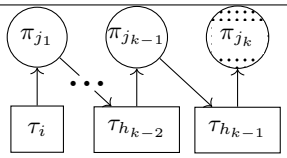
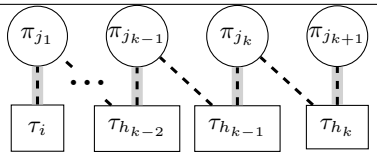
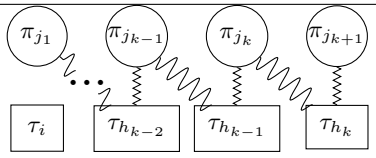
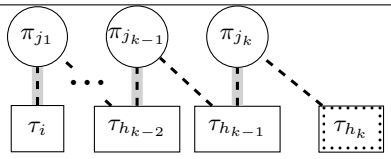
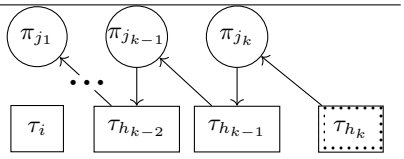
Base case ($k = 1$)**.** Our proof obligation is to show that some p_1 exists that maintains (28)–(31). Because τ_i is scheduled at t in $\mathcal{R}$, it must be scheduled on some processor in $\mathcal{R}$ (hence, it is matched with some processor in M). Let this processor be π_{j_1}. Thus, $p_1 = (\pi_{j_1})$, satisfying (28) (see Fig. 9c). Note (29) and (30) are vacuously true. (31) follows from the fact that p_1 contains a single vertex.

Induction step ($k > 1$). Our proof obligation is to show that if some path p_k maintains (28)–(31), then some path p_{k+1} exists that also maintains these invariants. We will let $p_{k+1} = p_k + (\pi_{j_k}, \tau_{h_k}, \pi_{j_{k+1}})$, with τ_{h_k} and $\pi_{j_{k+1}}$ to be defined later.

We start by showing (28) is maintained by p_{k+1}. Because we are building p_{k+1} inductively from p_k, p_k and p_{k+1} share the same beginning vertex π_{j_1}. Thus, (28) holds for p_{k+1} because it holds for p_k.

It remains to construct τ_{h_k} and $\pi_{j_{k+1}}$ such that (29) and (30) are maintained by p_{k+1}. During this construction, we will continuously reference Tbl. 1, which illustrates this construction. Rows 1–3 (resp., 4–5) represent distinct cases for the state of the matching M' (resp., M). For each case, either we construct a new vertex (τ_{h_k} or $\pi_{j_{k+1}}$) for p_{k+1} or we demonstrate that a scheduling cascade is possible in the respective matching. This is described in the "Description" column of Tbl. 1; the "Path" column illustrates either an extension of p_k or the alternating path by which the scheduling cascade is possible. Thus, for example, path p_2 in Fig. 9d is extended to obtain path p_3 in Fig. 9e by considering the cases in Tbl. 1 for $k = 2$.

We next show (29) is maintained by p_{k+1}. Because we are building p_{k+1} inductively from p_k, the cases $\ell \in [1, k-1]$ of (29) hold because they hold in p_k. Thus, we need only consider the case that $\ell = k$, which means we must construct task τ_{h_k} such that $(\tau_{h_k}, \pi_{j_k}) \in M'$.

		Matching	Description	Path
1	M'		$d_{h_k}(t) \prec d_i(t)$: add edge (τ_{h_k}, π_{j_k}) to p_k. $p_k + (\pi_{j_k}, \tau_{h_k})$ is shown on the right.	
2	M'		$d_i(t) \prec d_{h_k}(t)$: alternating path for M' in G_τ is shown on the right.	
3	M'		π_{j_k} is not matched: alternating path for M' in G_τ is shown on the right.	
4	M		Add edge $(\tau_{h_{k+1}}, \pi_{j_k})$ to $p_k + (\pi_{j_k}, \tau_{h_k})$. p_{k+1} is shown on the right.	
5	M		τ_{h_k} is not matched: alternating path for M in G_τ is shown on the right.	

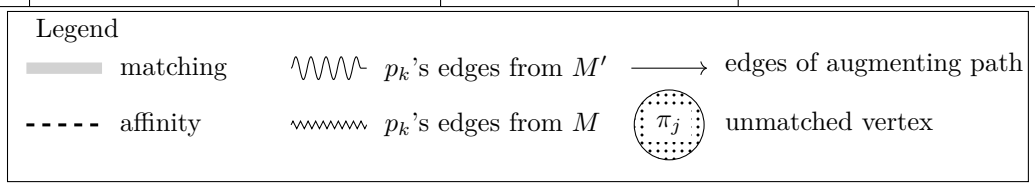


Table 1 The induction step for the proof of Lemma 61, which proves p_{k+1} maintains the listed invariants assuming p_k does.

▷ Claim 62. $\exists \tau_{h_k} \in \tau$ such that $(\tau_{h_k}, \pi_{j_k}) \in M'$.

Proof. We will prove the claim by contradiction by constructing an alternating path in M' by which a scheduling cascade is possible. Consider the state of processor π_{j_k} in schedule $\mathcal{R}'$. Processor π_{j_k} must schedule some

$$\tau_{h_k} \in \tau^{hp}(\mathcal{R}', \tau_i, t) \tag{32}$$

(the matching in row 1 of Tbl. 1); otherwise, either $\tau_{h_k} \notin \tau^{hp}(\mathcal{R}', \tau_i, t)$ (meaning $d_i(t) \prec d_{h_k}(t)$, the matching in row 2) or π_{j_k} is idle at t in $\mathcal{R}'$ (the matching in row 3). Furthermore, because τ_i is not scheduled in $\mathcal{R}'$ (hence, it is unmatched in M') and by (29), we then have that $(\tau_i, \pi_{j_1}) + p_k$ is an alternating path in M' for AG (see the identical alternating paths in rows 2 and 3). Thus, a scheduling cascade is possible via this alternating path. This contradicts that $\mathcal{R}'$ is produced under IA-GEDF. Thus, π_{j_k} schedules some $\tau_{h_k} \in \tau^{hp}(\mathcal{R}', \tau_i, t)$ at time t in schedule $\mathcal{R}'$ (in row 1, we extend p_2 by τ_{h_2}). This proves the claim. ◀

Thus, (29) is maintained by p_{k+1} (see the matching M' in Tbl. 1, row 1).

We next show (30) is maintained by p_{k+1}. Because we are building p_{k+1} inductively from p_k, the cases $\ell \in [1, k-1]$ of (30) hold because they hold in p_k. Thus, we need only consider the case that $\ell = k$, which means we must construct processor $\pi_{j_{k+1}}$ such that $(\tau_{h_k}, \pi_{j_{k+1}}) \in M'$.

▷ Claim 63. $\exists \pi_{j_{k+1}}$ such that $(\tau_{h_k}, \pi_{j_{k+1}}) \in M$.

Proof. Consider the state of task τ_{h_k} at time t in $\mathcal{R}$. Because $\tau_{h_k} \in \tau^{hp}(\mathcal{R}', \tau_i, t)$ (by (32)) and $\tau^{hp}(\mathcal{R}', \tau_i, t) \subseteq \tau^{hp}(\mathcal{R}, \tau_i, t)$ (by Lemma 59), we have that $\tau_{h_k} \in \tau^{hp}(\mathcal{R}, \tau_i, t)$. We claim that task τ_{h_k} is scheduled on some processor $\pi_{j_{k+1}}$ at t in $\mathcal{R}$ (the matching in row 4 of Tbl. 1). Assume, to the contrary, that it is not. Then, because τ_{h_k} is not scheduled in $\mathcal{R}$ (hence, it is unmatched in M) by (30), we have that $(\tau_{h_k}, \pi_{j_k}) + \bar{p}_k$ is an alternating path in M for G_τ (the matching in row 5). Furthermore, because $\tau_{h_k} \in \tau^{hp}(\mathcal{R}, \tau_i, t)$, we also have that $d_{h_k}(t) \prec d_i(t)$. Because the last vertex of this alternating path π_{j_1} schedules τ_i in $\mathcal{R}$, a scheduling cascade is possible via this alternating path (see the alternating path in row 5). This contradicts that $\mathcal{R}$ is produced under IA-GEDF. Because $\pi_{j_{k+1}}$ schedules τ_{h_k} in $\mathcal{R}$ (in row 4, we extend the path from row 3 by π_{j_3}), we have $(\tau_{h_k}, \pi_{j_{k+1}}) \in M$, and the claim is proven. ◀

Thus, (30) is maintained by p_{k+1} (see the matching in Tbl. 1, row 4).

To show the last invariant, (31), is maintained by p_{k+1}, our obligation is to show that τ_{h_k} and $\pi_{j_{k+1}}$ are distinct from the vertices in p_k and τ_i. Because (29) and (30) held for p_k, all such vertices are matched in M or M'. Thus, if τ_{h_k} or $\pi_{j_{k+1}}$ were not distinct, then they would be doubly matched in M or M', contradicting Def. 26. Hence, τ_{h_k} and $\pi_{j_{k+1}}$ are distinct from the vertices of p_k and τ_i.

Because all invariants are maintained by p_{k+1}, by induction we have proved that, for any $k \geq 1$, there exists a path p_k that maintains (28)–(31).

Note that there are $2k-1$ vertices in p_k. By (31), the number of distinct vertices in p_k goes to ∞ as $k \to \infty$. This contradicts that τ and π are finite, so the initial assumption that τ_i is scheduled in $\mathcal{R}$ and not in $\mathcal{R}'$ at t is false, proving Lemma 61.

◀

Removing assumption (P). We concluded in Sec. 7 that our tardiness bounds for IA-GEDF under Identical-Aff hold for sporadic tasks so long as the properties in Sec. 3 hold for VPP tasks. The proofs of the properties in Sec. 3 rely only on the model-related concepts we listed in the beginning of Sec. 3 and assumptions (P) and (W). We can produce an analogous assumption and list of concepts for the VPP task model:

Every job from τ_i^V has an execution requirement equal to $C_{i,j}^V$. (W-VPP)

- The task-system parameters C_i^V, T_i^V, u_i^V, $r_i^V(t)$, and $d_i^V(t)$;
- Lag and LAG, as defined in Def. 7, which are unchanged outside of exchanging τ_i with τ_i^V;
- The fact that $\mathcal{I}$ continuously executes task τ_i^V with speed u_i^V, which follows from (W-VPP)

under which the proofs still follow. The proofs are *exactly* the same outside of replacing C_i, T_i, u_i, $r_i(t)$, and $d_i(t)$ with C_i^V, T_i^V, u_i^V, $r_i^V(t)$, and $d_i^V(t)$, save one. The modified proof is as follows.

▶ **Lemma 64** (Analogous to Lemma 10 in Sec. 3)**.** *If task τ_i^V is pending at time t in $\mathcal{R}$, then*

$$t - \frac{\mathsf{Lag}_i(t)}{u_i^V} < d_i^V(t) \leq t - \frac{\mathsf{Lag}_i(t)}{u_i^V} + T_i^V. \tag{33}$$

Proof. Let $e_i(t)$ denote the remaining execution requirement for the ready job $J_{i,j}^V$ of τ_i^V at time t. Because this job is ready, it must not be complete, hence

$$0 < e_i(t) \leq C_{i,j}^V. \tag{34}$$

All jobs of τ_i^V prior to $J_{i,j}$ must have been completed by time t. Let W denote the total execution requirement of these jobs. Then,

$$A(\mathcal{R}, \tau_i, 0, t) = W + C_{i,j}^V - e_i(t). \tag{35}$$

In $\mathcal{I}$, all prior jobs of τ_i^V have completed by $r_i^V(t)$. Within $(r_i(t)^V, t)$, $\mathcal{I}$ continuously executes τ_i^V on a processor with speed u_i^V. Thus,

$$A(\mathcal{I}, \tau_i, 0, t) = W + (t - r_i^V(t))u_i^V. \tag{36}$$

Given these facts, an expression for $\mathsf{Lag}_i(t)$ can be derived as follows.

$$\begin{aligned}
\mathsf{Lag}_i(t) &= \{\text{by Def. 7}\} \\
&\quad A(\mathcal{I}, \tau_i^V, 0, t) - A(\mathcal{R}, \tau_i^V, 0, t) \\
&= \{\text{by (35) and (36)}\} \\
&\quad (t - r_i^V(t))u_i^V - (C_{i,j}^V - e_i(t)) \\
&= \{\text{because } d_i^V(t) = r_i^V(t) + T_{i,j}^V\} \\
&\quad (t - d_i^V(t) + T_{i,j}^V)u_i^V - (C_{i,j}^V - e_i(t)) \\
&= \{\text{because } u_i^V T_{i,j}^V = C_{i,j}^V\} \\
&\quad (t - d_i^V(t))u_i^V + e_i(t)
\end{aligned}$$

By (34) and the above expression, we have

$$(t - d_i^V(t))u_i^V < \mathsf{Lag}_i(t) \leq (t - d_i^V(t))u_i^V + C_{i,j}^V \leq (t - d_i^V(t))u_i^V + C_i^V, \tag{37}$$

and because $C_i^V \geq C_{i,j}^V$,

$$(t - d_i^V(t))u_i^V < \mathsf{Lag}_i(t) \leq (t - d_i^V(t))u_i^V + C_i^V, \tag{38}$$

which (using $T_i^V = C_i^V / u_i^V$) can be rearranged to obtain (33). ◀

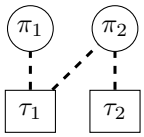


Figure 10 AG for task system of Lemma 65.

Hence, the general Lag properties, HP-LAG-compliant scheduler tardiness bounds, and IA-GEDF tardiness bounds all hold under VPP task systems with (W-VPP). Because we have proven in Lemma 57 that reducing execution requirements does not increase tardiness, our bounds apply to any VPP task system. Hence, Corollary 38 applies to sporadic tasks.

C SCHED_DEADLINE with Affinity Masks

In this appendix, we show that SCHED_DEADLINE is not HP-LAG-compliant and compare the time complexity of SCHED_DEADLINE to that of Alg. 2, our IA-GEDF algorithm.

C.1 HP-LAG-non-compliance for SCHED_DEADLINE

We begin with a brief idealized description of SCHED_DEADLINE. (Our description is idealized because we assume that scheduling decisions can be done in zero time.) Under SCHED_DEADLINE, all migrating pending tasks that are not scheduled are called *pushable*. The behavior of SCHED_DEADLINE is as follows.

SCHED_DEADLINE:	A processor π_j chooses its next task to run by selecting the task τ_i with the earliest deadline among the tasks fixed to π_j and the pushable tasks such that $\pi_j \in \tau_i$.

▶ **Lemma 65.** *SCHED_DEADLINE is not HP-LAG-compliant under Identical-Aff.*

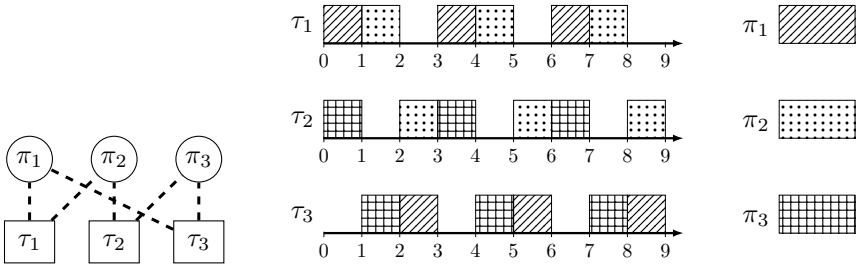
Proof. Our proof obligation is to provide a task system τ and time instant t such that the task-to-processor assignments made by SCHED_DEADLINE do not satisfy HP-LAG. The task system τ and the AG G_τ will be as in Fig. 10. Let $T_1 = 1$, $T_2 = 2$, and $u_1 = u_2 = 1.0$. This system is clearly feasible if task τ_1 executes exclusively on processor π_1 and task τ_2 executes exclusively on processor π_2.

Consider HP-LAG with $\tau'\{\tau_1, \tau_2\}$ and $t = 0$. For this τ' and t, by HP-LAG, we should have

$$\exists \delta > 0 \ \forall t' \in (0, \delta) : \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') \geq \mathsf{LAG}(\{\tau_1,\ \tau_2\}, t'). \tag{39}$$

We next construct the task-to-processor assignments that we will consider. At time $t = 0$, it is possible for task τ_1 to be scheduled on processor π_2 and for task τ_2 to be unscheduled under SCHED_DEADLINE. This will occur if processor π_2 selects a task to schedule before processor π_1. While π_2 makes its scheduling decision, task τ_1 is pushable and has an earlier deadline than the fixed task τ_2. Thus, π_2 selects τ_1 to schedule and removes it from the set of pushable tasks. Because τ_1 is no longer pushable, τ_1 is invisible to π_1 as it makes its scheduling decision.

We will contradict (39) using the task-to-processor assignments described above to show that SCHED_DEADLINE is not HP-LAG-compliant. Note that the time interval $(0, 1)$ is a continuous scheduling interval ($J_{1,1}$ completes at time 1). For any $\delta > 0$, there exists



(a) AG. **(b)** W-IA-GEDF schedule.

Figure 11 Counterexample for the proof of Lemma 67.

$t' = \min(0.5, \delta/2)$ such that $t' \in (0, \delta)$ and $(0, t') \subset (0, 1)$ (hence, the task-to-processor assignments do not change in $(0, t')$). Thus, the following holds.

$$
\begin{aligned}
\mathsf{LAG}(\{\tau_1,\ \tau_2\}, t') &= \{\text{by Def. 7}\} \\
&\quad \mathsf{LAG}(\{\tau_1,\ \tau_2\}, 0) + \sum_{i=1}^{2} A(\mathcal{I}, \tau_i, 0, t') - A(\mathcal{R}, \tau_i, 0, t') \\
&= \{\text{by Defs. 3 and 6}\} \\
&\quad \mathsf{LAG}(\{\tau_1,\ \tau_2\}, 0) + u_1 \cdot t' - A(\mathcal{R}, \tau_1, 0, t') + u_2 \cdot t' - A(\mathcal{R}, \tau_2, 0, t') \\
&= \{\text{by Def. 3 and the assumption that } \tau_1 \text{ is scheduled on } \pi_2 \\
&\quad \text{and } \tau_2 \text{ not scheduled in } (0, t')\} \\
&\quad \mathsf{LAG}(\{\tau_1,\ \tau_2\}, 0) + u_1 \cdot t' - 1.0 \cdot t' + u_2 \cdot t' - 0 \\
&= \{u_1 = 1.0 \text{ and } u_2 = 1.0\} \\
&\quad \mathsf{LAG}(\{\tau_1,\ \tau_2\}, 0) + 1.0 \cdot t' - 1.0 \cdot t' + 1.0 \cdot t' \\
&> \{\text{because } t' > 0\} \\
&\quad \mathsf{LAG}(\{\tau_1,\ \tau_2\}, 0)
\end{aligned}
$$

This contradicts (39), proving that `SCHED_DEADLINE` is not HP-LAG-compliant. ◀

The fact that `SCHED_DEADLINE` is not HP-LAG-compliant proves the following corollary.

▶ **Corollary 66.** *`SCHED_DEADLINE` does not implement IA-GEDF.*

Though we cannot prove at this time that `SCHED_DEADLINE` is or is not SRT-optimal under Identical-Aff (searching for counterexamples is tedious given the complexity of `SCHED_DEADLINE`'s implementation), we can show that generalizations of IG-GEDF for Identical-Aff that are not HP-LAG-compliant may have unbounded tardiness. Consider such a generalization below.

W-IA-GEDF:	At every scheduling event, the task-to-processor assignments are made such that afterwards no preemption is possible. These assignments do not change until the next scheduling event.

We denote this rule with the prefix "W-" because prioritization of tasks is weaker under this rule than under IA-GEDF (observe the rules are identical outside of the replacement of "scheduling cascade" with "preemption").

▶ **Lemma 67.** *W-IA-GEDF is not SRT-optimal under Identical-Aff.*

Proof. We prove the lemma by constructing a feasible task system with unbounded tardiness. Consider the task system in Fig. 11a where $u_1 = u_2 = u_3 = 1$ and $T_1 = T_2 = T_3 = 1$. Assume ties are broken in favor of the task with the lower index. This system is clearly feasible if τ_1 executes exclusively on π_1, τ_2 executes exclusively on π_2, and τ_3 executes exclusively on π_3.

We can show that the schedule for this task system in Fig. 11b is a W-IA-GEDF schedule that repeats every three time units with unbounded tardiness for every task. For any $t \in (0, 1)$, task τ_3 cannot preempt tasks τ_1 and τ_2 because all tasks have equal deadlines and deadline ties are not resolved in favor of τ_3. For any $t \in (1, 2)$, task τ_2 cannot preempt task τ_1 because the deadline tie is resolved in favor of τ_1, and cannot preempt task τ_3 because $d_3(t) = 1 < 2 = d_2(t)$. For any $t \in (2, 3)$, task τ_1 cannot preempt tasks τ_2 and τ_3 because $d_2(t) = d_3(t) = 2 < 3 = d_1(t)$. In the interval $(3, 4)$, all the tasks once again have equal deadlines, and so the schedule repeats.

Each task releases three jobs and completes two jobs every 3.0 time units. Hence, for any $k \in [1, \infty)$, by time $3k + 1$, each task has completed a job with tardiness equal to k. Thus, tardiness is unbounded for every task. ◀

This result cannot be applied to `SCHED_DEADLINE` because we relied on the fact that there are no restrictions in W-IA-GEDF on which processor each task is scheduled on so long as no preemptions can be made. `SCHED_DEADLINE` is more deterministic in how task-to-processor assignments are made. For example, `SCHED_DEADLINE` would not have migrated τ_1 at time instant 1 as in Fig. 11b.

C.2 Complexity of `SCHED_DEADLINE`

In the idealized description of `SCHED_DEADLINE`, we insinuated that all pushable tasks were contained in a single set. In reality, each processor maintains its own set of pushable tasks in a deadline-ordered balanced binary tree. If we assume for simplicity that all tasks are migrating, then a processor chooses to run the task with the earliest deadline within its local pushable set at any scheduling event under `SCHED_DEADLINE`.

Some pseudocode describing `SCHED_DEADLINE` which we will refer to in this subsection is provided in Alg. 3. Because each processor maintains its own local pushable set, a mechanism for sharing tasks is needed for `SCHED_DEADLINE` to be global. This mechanism is provided by *push* and *pull* operations. A pull operation (see the `pull_tasks` function in Alg. 3) by a processor π_j examines the pushable set of each other processor π_k for tasks with early deadlines that are eligible to run on π_j (line 12), and then pulls these tasks into its own pushable set (lines 14 and 15). Likewise, a processor π_j can push the lower-priority tasks in its pushable set to other processors running tasks with even later deadlines.

At a scheduling event, a processor π_j executes its scheduling function (`pick_next_task` in Alg. 3). In order to ensure that the unscheduled task with the earliest deadline whose affinity-mask contains π_j is scheduled, π_j first performs a pull operation (the task with the earliest deadline may have been in another processor's pushable set). The previously scheduled task on π_j is also enqueued in case it has high priority at the scheduling event. The task with the earliest deadline in π_j's pushable set is then dequeued and scheduled on π_j, after which π_j will attempt to push tasks to other processors (e.g., if the previously scheduled task was preempted, but has an early enough deadline to preempt the running task on some other processor).

Note that because of the `pull_tasks` function, the pushable set appears to be a "global" set. Furthermore, scheduled tasks are always dequeued from the pushable sets of their corresponding processors. Hence, the idealized description used above is valid.

```
1  Function pick_next_task(processor π_j, previous task τ_p): // a scheduling event
     at time t
2      pull_tasks(π_j);
3      pushable(π_j).enqueue(τ_p);
4      τ_h ← task τ_h with earliest deadline in pushable(π_j);
5      pushable(π_j).dequeue(τ_h);
6      π_j schedules τ_h;
7      push_tasks(π_j);
8      return;
9  Function pull_tasks(processor π_j, previous task τ_p):
10     d_min ← d_p(t);
11     for π_k ∈ π such that π_j ≠ π_k do
12         τ_h ← task τ_h with earliest deadline in pushable(π_k) such that π_j ∈ α_h;
13         if d_h(t) < d_min then
14             pushable(π_k).dequeue(τ_h);
15             pushable(π_j).enqueue(τ_h);
16             d_min ← d_h(t);
17     return;
```

Algorithm 3: Pseudocode of `SCHED_DEADLINE` as it relates to pushable tasks.

▶ **Lemma 68.** *Function* `pick_next_task` *of* `SCHED_DEADLINE` *has worst-case time complexity* $O(n + m \log n)$ *per scheduling event under* Identical-Aff.

Proof. The $O(n)$ term of the complexity results from line 12 of Alg. 3. At first blush, this step should take at most $O(\log n)$ operations because each pushable set is implemented as a deadline-ordered balanced binary tree. The problem arises from the requirement that the returned task needs to be eligible to run on processor π_j. Because affinity-mask information is irrelevant to the structure of the binary tree, line 12 may need to examine the affinity-mask of every task in π_k's pushable set. As this line is executed for almost every processor, the entirety of almost every pushable set may be examined. Because all unscheduled tasks are contained in some pushable set, in the worst case this requires examining $O(n)$ affinity masks.

The remaining $O(m \log n)$ term of the complexity arises from the fact that lines 14 and 15 may be run for each processor if the task returned by line 12 has an earlier deadline than its predecessor in each iteration of the for loop. Insertion and removal from balanced binary trees is $O(\log n)$, and there are $O(m)$ such trees, thereby leading to the $O(m \log n)$ term.

Thus, the total complexity is $O(n + m \log n)$ in the worst case. ◀

Somewhat surprisingly, the need to separate the state of the task system into per processor data structures and the treatment of affinity masks as a second-class concern result in `SCHED_DEADLINE` having a higher worst-case time complexity per scheduling event than Alg. 2 with preprocessing ($O(m + \log n)$ by Theorem 54). This suggests that even though Alg. 2 appears more complex than `SCHED_DEADLINE` on paper, in practice, this level of complexity is already present in real implementations of GEDF.

Note that Alg. 2 does not have lower time complexity if the same affinity reduction preprocessing is applied to `SCHED_DEADLINE`. Earlier in this section, we made the simplifying assumption that all tasks are migrating, while in reality, tasks may be fixed. These fixed tasks are not maintained in pushable sets because it is pointless to attempt pushing or pulling a fixed task that cannot migrate. As a result, they are not examined in line 12. Thus, the $O(n)$

term in Lemma 68 is actually O(the number of migrating tasks). If the same preprocessing step that we applied to Alg. 2 is used by `SCHED_DEADLINE`, the number of migrating tasks is $O(m)$ (Theorem 3 in [7]).

Furthermore, because the number of migrating tasks in each pushable set is reduced, the cost of the enqueue and dequeue operations in lines 14 and 15 is dominated by line 4. In reality, this line must also consider tasks that are fixed on π_j, and hence, not pushable. This is implemented in `SCHED_DEADLINE` by maintaining *two* binary trees for each processor π_j: one for maintaining π_j's pushable set and the other for maintaining all tasks in π_j's runqueue (pushable and fixed). The former is used in line 12 while the latter is used in line 4. Because line 4 still needs to consider π_j's pushable and fixed tasks, the surrounding enqueue and dequeue operations are actually $O(\log n)$ in time complexity. Hence, the total time complexity of `SCHED_DEADLINE` with affinity reduction preprocessing is $O(m + \log n)$ per scheduling event, which is equal to Alg. 2's time complexity with the same preprocessing.