Towards Practical Multiprocessor EDF with Affinities

Stephen Tang and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill
{sytang|anderson}@cs.unc.edu

***Abstract*—A gap exists between the theory of EDF scheduling on identical multiprocessors with arbitrary processor affinities (APA) and practical EDF scheduling as embodied by the `SCHED_DEADLINE` (SD) scheduler in Linux. This is because the EDF variant proposed in theory for APA, called *Strong APA EDF*, introduces affinity-related complexities that are not applicable under global EDF, the original target of SD. SD instead treats affinities as a secondary concern. It is shown herein that this treatment comes at the price of causing SD to be fundamentally broken with regard to soft real-time (SRT)-optimality with APA. This result resolves a longstanding open question regarding this matter. It also suggests that Strong APA EDF, which *has* been proven to be SRT-optimal, is necessary for practical EDF scheduling with APA. However, non-preemptive sections are typically required in practice, and prior work on Strong APA EDF is limited to fully preemptive systems. In this paper, this prior work is extended for the first time to deal with non-preemptivity, which introduces non-trivial nuances with APA. As a byproduct of considering non-preemptivity, it is shown that the SRT-optimality of EDF in this context carries over to a significantly expanded class of schedulers**

I. Introduction

Variants of earliest-deadline-first (EDF) scheduling have been proven to be *soft real-time (SRT)-optimal* on various processor models [6], [11], [13], meaning that EDF can guarantee bounded deadline tardiness for any (sporadic) task system that does not over-utilize the underlying multiprocessor platform. This property makes EDF an attractive scheduler for systems where some tardiness is permissible.

The SRT-optimality of EDF is significant enough to have warranted mentioning in the documentation of `SCHED_DEADLINE` (SD) [1], Linux's EDF implementation. This documentation specifically references work of Devi and Anderson [6], who were the first to establish the SRT-optimality of global EDF scheduling on identical multiprocessors. As this documentation suggests, the design of SD reflects an initial focus on global EDF (and by extension, clustered EDF, where global scheduling is used within each cluster of processors), with *arbitrary processor affinities (APA)* being treated as a secondary concern. This is unfortunate because setting processor affinities (`cpusets` in Linux) is useful for load balancing and cache locality. While SD may not crash under APA, its SRT-optimality under APA has never been established. The question of whether SD is SRT-optimal was listed among major open problems facing Linux by Peter Zijlstra in his ECRTS'17 keynote [14].

Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

This open problem was partially addressed by our prior work [11], in which we proved that an EDF variant (distinct from SD) introduced by Cerqueira et al. called *Strong APA EDF (SAPA-EDF)* [5] is SRT-optimal under APA.[1] While both SD and SAPA-EDF prioritize tasks on an EDF basis, SAPA-EDF migrates tasks more aggressively so that the maximum number of high-priority tasks is scheduled at any time. At runtime, computing which migrations to enact involves executing graph searches. The increased overheads of such searches and implementation complexities of SAPA-EDF are unnecessary under global scheduling, the main target of SD. Perhaps because of this, the maintainers of SD have not modified it to implement SAPA-EDF, even though it is SRT-optimal.

While SAPA-EDF is SRT-optimal in theory, other theoretical issues must be resolved before a practical implementation of it is possible. One key issue is that its proof of SRT-optimality requires tasks to always be preemptive [11]. This is unrealistic because actual workloads require some degree of non-preemptivity (for system calls, using locking protocols, etc.). Non-preemptivity has not been a problem under global scheduling because non-preemptive (NP) global EDF is also SRT-optimal [6]. In contrast, no work to date on tardiness under APA has considered non-preemptivity.

To summarize, SD and SAPA-EDF are the two natural potential starting points for designing an EDF scheduler that is practical under APA. However, given the aforementioned issues with both schedulers, it is not immediately clear which, if either, can realistically serve as a starting point.

Contributions. Towards clarifying which scheduler should be the starting point, this paper makes four major contributions.

First, we answer Zijlstra's open question (in the negative) by showing that the current SD implementation is fundamentally broken with respect to tardiness under APA. We do this by presenting counterexamples where feasible task systems have unbounded tardiness under an idealized version of SD without overheads. This suggests that an aggressive migration strategy similar to that of SAPA-EDF is necessary for SRT-optimality under APA, leading us to argue that SAPA-EDF is a preferable starting point for a practical EDF scheduler under APA.

Second, we demonstrate that the reliance on tasks being fully preemptive in the proof of SAPA-EDF's SRT-optimality [11] is not a limitation of this proof, but rather, is

[1] We did not realize that Strong APA EDF had already been discovered under a hard real-time context and unfortunately presented it as IA-GEDF in [11].

an intrinsic requirement of any work-conserving EDF variant under APA. We show this by presenting an example task system similar to that used to demonstrate Dhall's effect [7], where NP execution under APA causes tardiness to increase unboundedly even though total utilization is just above 1.0. Similarly to how Dhall's effect has not rendered EDF unusable with global scheduling, we argue that our counterexample does not render EDF unusable under APA. Specifically, we define a weaker notion of NP execution called *progression*, and argue that progression can replace NP sections in many use cases. We prove that SAPA-EDF fortunately can be modified to support progression while maintaining SRT-optimality.

Third, as a byproduct of establishing the above result about progression, we prove that variants of *window-constrained schedulers*, which have been proven to be SRT-optimal under global scheduling [8], are also SRT-optimal under APA. The class of window-constrained schedulers includes common schedulers such as first-in-first-out (FIFO), least-laxity-first (LLF), and EDF-until-zero-laxity (EDZL), so this result greatly expands the class of known SRT-optimal schedulers under APA. While our proof of this result leverages that pertaining to SAPA-EDF [11], a number of non-trivial extensions were required in our case, because certain properties of SAPA-EDF that were instrumental in [11] are not true generally under window-constrained schedulers.

Fourth, we present the first ever (to our knowledge) NP-blocking analysis under APA. Real workloads will need such analysis, as not all instances of NP execution can be replaced with progression. Unfortunately, NP-blocking analysis under APA introduces nuances that make computing blocking terms more pessimistic than under global scheduling.

Organization. The rest of this paper is organized as follows. After covering needed background in Sec. II, we demonstrate in Sec. III that, under APA, SD is not SRT-optimal, nor is any EDF variant with NP sections. Next, in Sec. IV, we establish the SRT-optimality of window-constrained schedulers under APA. We then use this result in weakening non-preemptivity to progression in EDF in Sec. V. We conclude in Sec. VI.

II. Background

In this section, we cover notation and necessary background. For ease of reading, tables of notation and abbreviations used in this work are presented in Tbls. I and II, respectively.

A. Task Model

We consider a system of n implicit-deadline sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ running on m unit-speed processors $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$. We assume basic familiarity with the sporadic task model and focus on introducing notation here. We denote the j^{th} job released by task τ_i as $\tau_{i,j}$ and index jobs starting at $j = 1$. Job $\tau_{i,j}$ must be completed before job $\tau_{i,j+1}$ is allowed to execute. We let $C_{i,j}$ denote the required execution time of $\tau_{i,j}$, and let C_i denote the worst-case execution time (WCET) of τ_i over all its jobs. We let T_i denote the period of task τ_i. We denote the release time, deadline, and completion time of job $\tau_{i,j}$ by $r_{i,j}$, $d_{i,j}$, and

TABLE I: Notation.

Symbol	Meaning
τ	Task system
τ_i	i^{th} task of τ
τ'	Task subset
n	Number of tasks
π	Multiprocessor
π_i	i^{th} processor of π
m	Number of processors
$\tau_{i,j}$	j^{th} job of τ_i
$C_{i,j}$	Execution cost of $\tau_{i,j}$
C_i	WCET of τ_i
T_i	Period of τ_i
$T_{\max}$	Largest period
$r_{i,j}$	Release time of $\tau_{i,j}$
$d_{i,j}$	Deadline of $\tau_{i,j}$; $r_{i,j} + T_i$
$f_{i,j}$	Completion time of $\tau_{i,j}$
$T_{i,j}$	$r_{i,j+1} - r_{i,j}$
u_i	Utilization of τ_i; C_i/T_i
$u_{\min}$	Smallest utilization
$u_{\max}$	Largest utilization
$U(\tau')$	Sum total utilization over τ'
$C_i(t)$	Execution cost of τ_i's ready job at t
$r_i(t)$	Release time of τ_i's ready job at t
$d_i(t)$	Deadline of τ_i's ready job at t
$c_i(t)$	Remaining execution of τ_i's ready job at t
$D_i(t)$	Deadline of most recently completed job of τ_i by t
$\chi_{i,j}(t)$	Priority of $\tau_{i,j}$ at t
ϕ	Window size of a window-constrained scheduler
$\alpha(\tau_i)$	Processor affinity mask of τ_i
τ^G	Set of migrating tasks of τ
$vt_i(t)$	Virtual time of τ_i at t; Def. 4
$\mathsf{Dev}(\tau_i, t)$	Deviation of τ_i at t; Def. 5
$\mathsf{Dev}(\tau', t)$	Sum total deviation over τ'; Def. 5
L	Constant, usually compared against Dev
δ	Quantifier variable
ϵ	Quantifier variable
M	Maximal number of processors for some τ'
K	$(T_{\max} + 2\phi)/(2u_{\min})$
B_i	Blocking term of τ_i
I_i	Largest interval over which τ_i may be NP blocked
$\text{npdbf}_i(t, \Delta)$	NP demand bound function of τ_i over $[t, t + \Delta]$
γ_i	Worst-case NP execution time of any job of τ_i
ω_i	NP utilization of τ_i; γ_i/T_i
Γ_i	Sum total worst-case NP execution time over $\tau/\{\tau_i\}$
Ω_i	Sum total NP utilization over $\tau/\{\tau_i\}$
R_i	Response time bound of τ_i
$R^i_{\max}$	Largest response time bound over $\tau/\{\tau_i\}$

TABLE II: Abbreviations.

Abbreviation	Full
SRT	Soft Real-Time
APA	Arbitrary Processor Affinities
NP	Non-Preemptive
SD	`SCHED_DEADLINE`
SAPA-EDF	Strong APA EDF
WAPA-EDF	Weak APA EDF
SAPA-EF	Strong APA EDF-FIFO
L-SAPA-EF	Link-based Strong APA EDF-FIFO

$f_{i,j}$, respectively, where $d_{i,j} = r_{i,j} + T_i$ (implicit deadlines). We let $T_{\max}$ denote the maximum period over all tasks. We denote the spacing between the releases of successive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ of τ_i by $T_{i,j} = r_{i,j+1} - r_{i,j}$, where $T_{i,j} \geq T_i$. The utilization u_i of task τ_i is given by C_i/T_i. We denote the smallest (resp., largest) utilization over all tasks as $u_{\min}$ (resp., $u_{\max}$). For a set of tasks $\tau' \subseteq \tau$, we let $U(\tau')$ denote the sum of the utilizations of tasks in τ'.

At time t, a job $\tau_{i,j}$ is either *unreleased* ($t < r_{i,j}$), *pending* ($r_{i,j} \leq t < f_{i,j}$), or *complete* ($t \geq f_{i,j}$). If a task τ_i has pending jobs at t, then its *ready* job at t is its earliest-released pending job at t. If task τ_i has a ready job $\tau_{i,j}$ at time t, then we define the functions $C_i(t)$, $r_i(t)$, and $d_i(t)$ to equate to $C_{i,j}$, $r_{i,j}$, and $d_{i,j}$, respectively. We define the function $c_i(t)$ to equate to the remaining execution required by the ready job $\tau_{i,j}$ at t. We define the function $D_i(t)$ to equate to the deadline of the latest completed job of τ_i by time t, or 0 if no jobs of τ_i have completed. If task τ_i has a ready job at time t, then its priority is defined by the priority of its ready job. We define the latter using prioritization functions later in Def. 1.

The *response time* of a job $\tau_{i,j}$ is defined as $f_{i,j} - r_{i,j}$. The *tardiness* of $\tau_{i,j}$ is defined as $\max\{0, f_{i,j} - d_{i,j}\}$. The *tardiness* of task τ_i is the supremum of the tardiness of its jobs. If the tardiness of all tasks is bounded under a given scheduler, then the task system is *SRT-schedulable* under the scheduler. A task system is *SRT-feasible* if it is SRT-schedulable under some scheduler. A scheduler is *SRT-optimal* if any SRT-feasible task system is SRT-schedulable under it.

We assume that τ executes on π without overheads (e.g., scheduling code and context switches execute in zero time). We also assume that time is continuous. This assumption does not invalidate our results on systems where time is integral so long as WCETs and periods are also integral. We limit attention to *non-fluid* schedulers, i.e., if a job $\tau_{i,j}$ is scheduled (resp., unscheduled) at time t, then there exists $\epsilon > 0$ such that $\tau_{i,j}$ is scheduled (resp., unscheduled) over $[t, t + \epsilon]$.

B. Window-Constrained Schedulers

Leontyev and Anderson considered a class of schedulers called "window-constrained schedulers," showed that many common schedulers such as EDF, FIFO, and LLF are in this class, and proved that all schedulers in this class are SRT-optimal under global scheduling [8]. This result is relevant to NP scheduling because NP variants of window-constrained schedulers are themselves window-constrained under global scheduling. In this work, we will prove a weaker version of this result concerning NP scheduling under APA.

The class of window-constrained schedulers is defined by introducing the notion of a prioritization function.

Def. 1. (Def. 1 of [8]) Associated with each job $\tau_{i,j}$ is a function of time $\chi_{i,j}(t)$, called its *prioritization function*. If, at time t, $\chi_{i,j}(t) < \chi_{h,k}(t)$ holds, then the priority of job $\tau_{i,j}$ is higher than that of job $\tau_{h,k}$ at t. We assume consistent tie-breaking if $\chi_{i,j}(t) = \chi_{h,k}(t)$.

Under global scheduling, prioritization functions can serve as abstractions of schedulers. For example, EDF is represented by $\chi_{i,j}(t) = d_{i,j}$, and FIFO by $\chi_{i,j}(t) = r_{i,j}$. Likewise, fixed-priority (FP) scheduling can be abstracted by setting $\chi_{i,j}(t)$ to be task τ_i's priority level.

Def. 2. (Def. 5 of [8][2]) A scheduling algorithm's prioritization functions are *window-constrained* if there exists finite ϕ such that for any job $\tau_{i,j}$ and time t, $|d_{i,j} - \chi_{i,j}(t)| \leq \phi$ holds.

For example, the EDF and FIFO prioritization functions above are window-constrained with $\phi = 0$ and $\phi = \max_{i,j}\{d_{i,j} - r_{i,j}\} = T_{\max}$, respectively. On the other hand, the FP prioritization function is not window-constrained because $d_{i,j}$ is unbounded for any given task τ_i as $j \to \infty$ (assuming jobs are released indefinitely), while $\chi_{i,j}(t)$ is constant.

C. APA Scheduling

We denote the *processor affinity mask* of task τ_i as $\alpha(\tau_i) \subseteq \pi$. Common migration schemes such as global, partitioned, and clustered scheduling can be represented using affinities. For example, under global scheduling, $\alpha(\tau_i) = \pi$ for every task τ_i, and under partitioned scheduling, $\alpha(\tau_i) = \{\pi_j\}$, where π_j denotes the processor assigned to task τ_i. A task τ_i is *migrating* if $|\alpha(\tau_i)| > 1$, and *fixed* otherwise.

Processor affinities are often illustrated using *affinity graphs*. These are undirected bipartite graphs where the nodes represent tasks and processors, and each edge is between a task τ_i and a processor π_j, denoting that π_j is in τ_i's affinity mask.

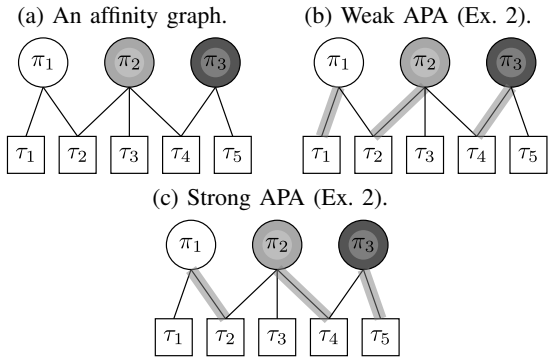
▷ **Ex. 1.** Consider a task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_5\}$ on $\pi = \{\pi_1, \pi_2, \pi_3\}$. Let $\alpha(\tau_1) = \{\pi_1\}$, $\alpha(\tau_2) = \{\pi_1, \pi_2\}$, $\alpha(\tau_3) = \{\pi_2\}$, $\alpha(\tau_4) = \{\pi_2, \pi_3\}$, and $\alpha(\tau_5) = \{\pi_3\}$. The corresponding affinity graph is shown in Fig. 1(a). τ_2 and τ_4 are migrating while all other tasks are fixed. ◁

Weak vs. Strong APA. A guarantee that should be made by any priority-based scheduler is that no preemption not taken by the scheduler can result in scheduling a higher-priority task. Ignoring the specifics of which task is assigned which processor, scheduling the m highest-priority tasks is the only way to meet this guarantee under global scheduling. Under APA, distinct scheduling decisions may meet this guarantee.

▷ **Ex. 2.** Suppose that the system in Ex. 1 is scheduled under EDF and that at time t, all tasks have ready jobs such that $d_2(t) < d_4(t) < d_5(t) < d_1(t) < d_3(t)$. Consider the scheduling decisions made in Fig. 1(b), where a highlighted edge denotes a task being scheduled on a processor. Because task τ_5 (resp., τ_3) cannot preempt τ_4 (resp., τ_2) on π_3 (resp., π_2) as $d_4(t) < d_5(t)$ (resp., $d_2(t) < d_3(t)$), this processor assignment meets our aforementioned guarantee. Likewise, because tasks τ_1 and τ_3 cannot preempt higher-priority tasks in Fig. 1(c), this assignment also meets the guarantee. ◁

[2]We have modified this definition to simplify our reasoning in Sec. IV. Any window-constrained prioritization functions as defined in [8] are also window-constrained as defined in this work and vice versa, though the size of windows may be larger under our definition.

Fig. 1: Affinity graph examples.



Even though no preemption can be taken in either of the assignments in Ex. 2, the one in Fig. 1(b) is inferior because it schedules τ_1 over τ_5. As a low-priority task is scheduled over a high-priority task, this is similar to a priority inversion.

Situations such as this motivate the question of how to schedule under APA while avoiding such priority inversions. Answering this led to the formalization of Weak and Strong APA [5]. *Weak APA* is the invariant that scheduling decisions are made such that no preemption that is not taken can result in the scheduling of a higher-priority task. For example, the assignments in Fig. 1(c) and Fig. 1(b) both maintain Weak APA. *Strong APA* maintains the invariant that any *shift* not taken cannot result in the scheduling of a higher-priority task.

Def. 3. An *alternating path* is a path in an affinity graph between a task and either an idle processor or a different task where every odd edge in the path connects a processor to a task that is not scheduled on it and and every even edge connects a processor to the task it is scheduling. *Shifting* [5] is the series of migrations that results from inverting each edge in an alternating path.

Shifting is better explained through example. Observe in Fig. 1(b) that a path $\{\tau_5, \pi_3, \tau_4, \pi_2, \tau_2, \pi_1, \tau_1\}$ can be traced from the higher-priority τ_5 to the lower-priority τ_1. This path alternates between edges where the corresponding task is not scheduled on the corresponding processor and edges where the reverse is true. By inverting the edges in this path, one arrives at Fig. 1(c). Because no shifts can result in the scheduling of a higher-priority task in Fig. 1(c), this assignment satisfies Strong APA. For convenience, any processor that is not executing a task is considered to be executing a logical task τ_0 with infinitely low priority. This way, actual tasks can be migrated to these processors via shifting.

A property satisfied by Strong-APA schedulers is that, for any subset of tasks with highest priority, the maximum number of tasks in this subset are scheduled. For example, in Fig. 1(c), no other processor assignment can schedule τ_1 without first unscheduling a higher-priority task. This property of Strong APA is key in the proof of the SRT-optimality of SAPA-EDF in [11], and in the proof in Sec. IV.

III. Sources of Non-SRT-Optimality in SD

In this section, we highlight three sources of non-SRT-optimality in SD under APA. The first two sources are due to specific design choices within SD, while the third pertains to any EDF variant. We consider an idealized version of SD in which scheduling decisions occur atomically and instantly. Note that sporadic releases must be exploited for tardiness to be unbounded as in these counterexamples. Extended versions of these counterexamples are available online [10].

The first source of non-optimality is that SD does not comply with Weak APA. This is due to an optimization made to speed up SD under global scheduling that prevents certain preemptions under APA.[3] Explaining how this occurs requires a high-level description of the SD implementation. For consistency, we refer to processes under Linux as tasks.

The design of SD is influenced by Linux's use of per-processor runqueues. It is required by Linux that tasks only ever execute on the processor that owns the runqueue that contains said task. Thus, to implement schedulers that require migration, processors must exchange tasks between their runqueues at runtime through operations called *push* (processor sends an unscheduled task to another processor) and *pull* (processor takes an unscheduled task from another processor). Generally, pushes (resp., pulls) are triggered by job releases (resp., completions). For example, consider a task τ_h that lacks the priority to execute on its runqueue's processor while having sufficient priority to preempt the running task τ_ℓ on another processor. For τ_h to preempt τ_ℓ, it must first be pushed from its current runqueue to the runqueue of the target processor.

The problem arises in how SD chooses which processor to send to in a push. When all processors in a pushed task's affinity mask are running other tasks, SD selects the processor whose running task has the latest deadline among *all*[4] processors as the target for the push, regardless of affinity masks. This is because the data structure (a heap called `cpu_dl`) that returns the target processor with latest deadline is oblivious to affinities. Note that this heap does not cause problems under global scheduling, where all tasks have affinity for all processors. Under global scheduling, usage of this heap is faster than iterating over all processors to determine which is running the task with the latest deadline.

If the processor chosen by a push is not in the pushed task's affinity mask, SD detects this and does not complete the push. Unfortunately, it does not retry the push, so the task remains unscheduled even if processors in its affinity mask are running tasks with later deadlines, thereby breaking Weak APA. The pushed task remains unscheduled until its runqueue's processor completes its running job or another processor in its affinity mask completes its running job and pulls the unscheduled task. In the worst-case, tasks may be starved as in the following example.

[3] We did not discover this. This issue was recognized by the SD maintainers in 2017 [9], but has not been patched in the Linux kernel. To our knowledge, no one has considered this issue from a tardiness perspective.

[4] In truth, only the processors in the same root domain are considered; however, all processors share the same root domain in our examples.

▷ **Ex. 3.** Consider the task system illustrated in Fig. 1(a). Let $(C_1, T_1) = (C_3, T_3) = (C_5, T_5) = (3, 6)$ and $(C_2, T_2) = (2, 2)$. For simplicity, assume that τ_4 does not release jobs in this example.[5] This task system has bounded tardiness under any Weak APA EDF (WAPA-EDF) scheduler,[6] but may have unbounded tardiness under SD, as shown in Fig. 2(a).

τ_2 is released periodically and initially executes on π_1. Even though fixed task τ_1 releases a job at $t = 1$, τ_2 does not migrate until τ_1 has higher priority at $t = 6$. However, prior to τ_2's attempt to migrate, τ_3 and τ_4 release jobs such that all processors execute jobs at $t = 6$. Thus, SD will mistakenly attempt to push τ_2 onto π_3 because it schedules the job with the latest deadline. Because π_3 is not in τ_2's affinity mask, this push will fail, and τ_2 is unscheduled until it is pulled by π_2 at $t = 7$. This occurs again at $t = 17$, except τ_3 forces τ_2 to attempt to migrate. This pattern can be repeated infinitely often, and with each occurrence, the maximum tardiness experienced by τ_2 increases by 1.0. ◁

We argue that even if the maintainers of SD are not interested in implementing SAPA-EDF, SD should at least be patched to comply with WAPA-EDF when not all tasks migrate globally. It is inconvenient from a theoretical perspective to consider SD when it does not satisfy basic invariants that one would expect from an EDF implementation.

Unfortunately, even if SD were modified such that pushes would only consider processors in the pushed task's affinity mask (thereby complying with Weak APA), unbounded tardiness would still be possible under feasible task systems. The second source of non-optimality of SD is that naïve implementations of WAPA-EDF are not optimal.[7]

▷ **Ex. 4.** Consider the task system illustrated in Fig. 1(a). Let $(C_1, T_1) = (C_5, T_5) = (2, 6)$, $(C_2, T_2) = (C_4, T_4) = (2, 2)$, and $(C_3, T_3) = (1, 6)$. This task system has bounded tardiness under SAPA-EDF [11], but may have unbounded tardiness under a WAPA-EDF scheduler, as shown in Fig. 2(b).

τ_2 and τ_4 release jobs periodically. Initially, τ_2 and τ_4 execute on π_1 and π_3, respectively. At $t = 6$, fixed tasks τ_1 and τ_5 preempt τ_2 and τ_4, respectively. The only other processor available to both τ_2 and τ_4 is π_2, which they cannot both use. We assume the tiebreak here favors τ_2 and it is scheduled, while τ_4 does not execute until $t = 8$ when it resumes execution on π_3. τ_2 is also forced to migrate off of π_2 by fixed task τ_3 at $t = 12$. This repeats at $t = 18$, except here τ_4 is scheduled over τ_2 because it is tardy by 2.0 time units due to not being scheduled over $[6, 8]$. As a result, τ_2 also

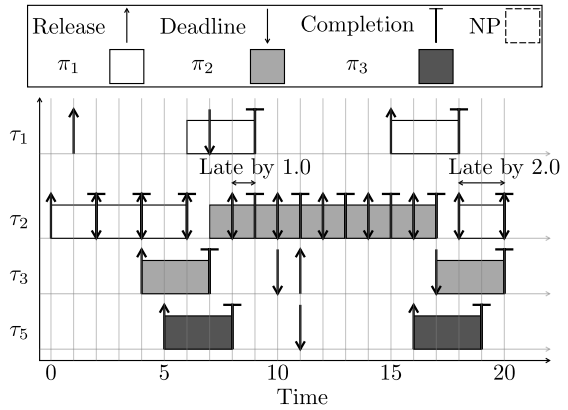
[5] τ_4 is necessary in this example for the assumption made in Footnote 4 that all processors share a root domain.

[6] Because τ_4 does not release jobs, the task system in Ex. 3 is made up of two disjoint subsystems: τ_5 running on π_3 and $\{\tau_1, \tau_2, \tau_3\}$ running on $\{\pi_1, \pi_2\}$. It can be shown with minor modifications to the proof in [6] that WAPA-EDF is SRT-optimal when $m \leq 2$. Thus, both subsystems have bounded tardiness under WAPA-EDF.

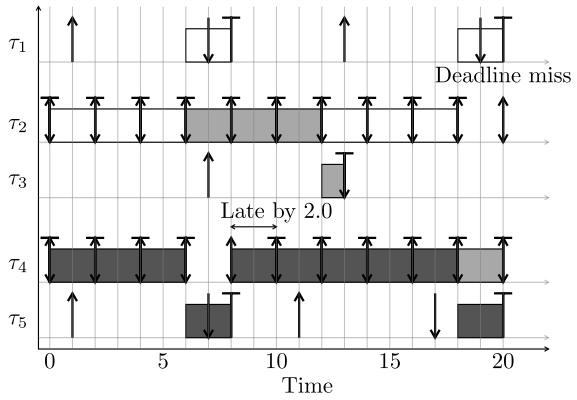
[7] Unbounded tardiness under a WAPA-EDF scheduler was demonstrated in App. C of [11], but the considered EDF variant can unrealistically migrate tasks for no reason. The modified SD that we consider in Ex. 4 is a more realistic WAPA-EDF scheduler because tasks only migrate when preempted.

Fig. 2: Bounded tardiness counterexamples. (Longer versions are available online [10].)

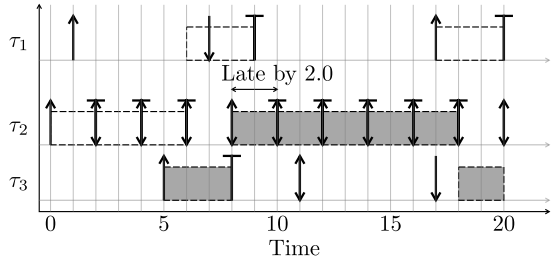
(a) SD violates Weak APA.



(b) SD with Weak APA has unbounded tardiness



(c) Non-preemptivity causes unbounded tardiness.



becomes tardy by 2.0 time units by $t = 20$. This pattern can repeat indefinitely, and with each occurrence, the maximum tardiness experienced by either τ_2 or τ_4 increases by 2.0. ◁

The final source of non-optimality we consider is the presence of NP sections. The next example shows that no work-conserving EDF variant is SRT-optimal under APA when jobs are fully NP. Note that there is no distinction between Strong and Weak APA when jobs do not migrate once scheduled.

▷ **Ex. 5.** For this example, we only consider $\tau_1, \ldots, \tau_3$ of Fig. 1(a). Let $(C_1, T_1) = (C_3, T_3) = (3, 6)$ and $(C_2, T_2) = (2, 2)$. Any EDF variant may have unbounded tardiness when jobs are fully NP, as seen in Fig. 2(c).

τ_2 releases jobs periodically. Initially, τ_2 executes on π_1. When τ_2 finishes its ready job at $t = 6$, it is preempted by τ_1. However, τ_3 has already begun its NP execution on π_2, and

so τ_2 is unscheduled over $[6, 8]$. This occurs again at $t = 18$, though this time τ_2 is forced to migrate by τ_3. This pattern can be repeated infinitely often, and with each occurrence, the maximum tardiness experienced by τ_2 increases by 2.0.

This example contains a single global task with utilization 1.0 that is eventually preempted by a task that is fixed on the processor the global task is executing on, and is then briefly unable to execute because all other processors are executing NP fixed tasks. While this example was limited to two processors and two fixed tasks for simplicity, it can be generalized for any $m \geq 2$ processors and fixed tasks. Also, increasing the fixed tasks' periods only increases the time in between intervals where the global task is unable to execute, so such intervals still can occur infinitely often. This means the global task still has unbounded tardiness even when the utilization of every fixed task approaches 0. Thus, when jobs are NP, for any arbitrarily small $\epsilon > 0$, there exists task system τ with $U(\tau) = 1 + \epsilon$ such that τ has unbounded tardiness. ◁

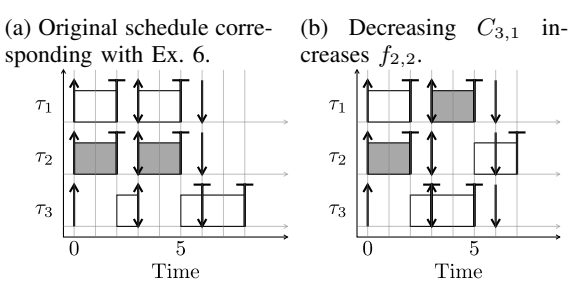
We believe that SAPA-EDF is a more reasonable starting point for a practical EDF scheduler under APA than WAPA-EDF because under *ideal* conditions (i.e., no overheads or NP sections), the former is SRT-optimal [11], while the latter has inherent capacity loss (Ex. 4). We believe that such inherent loss will outweigh any capacity loss caused by overheads or NP sections in SAPA-EDF. Unfortunately, justifying this via an overhead-aware schedulability study would require that several theoretical and implementation-related questions be answered. Firstly, though we have demonstrated in Ex. 4 that WAPA-EDF is not SRT-optimal, we lack a sufficient SRT schedulability condition for WAPA-EDF, which such a schedulability study would require. Secondly, we would also need well-maintained implementations of WAPA-EDF (as SD does not comply with it, as shown in Ex. 3) and SAPA-EDF (which, to our knowledge, does not exist[8]) in order to measure overheads under these respective schedulers. Lastly, we would also need analysis techniques for accounting for overheads and NP blocking (which some overheads can induce). Considering all of these issues will entail work that is beyond the scope of a single paper. Here, we begin this work by considering the important issue of NP execution under SAPA-EDF.

IV. SRT-Optimality of Window-Constrained Schedulers

Before discussing non-preemptivity, we prove an interesting sub-result in this section that we will make use of in our NP-blocking analysis. We prove that the analysis in [11] for EDF can be generalized to work for any window-constrained scheduler under fully preemptive Strong APA scheduling. Due to the density of notation presented in this and the following section, we recommend the reader refer to Tbls. I and II.

Our analysis here is similar to that of [11] in that almost every lemma, corollary, and theorem we present has an analogue in [11]. The major difference is that in [11] it is assumed that all tasks are periodic and that all jobs execute to their tasks' WCETs. Relaxing these assumptions to establish the SRT-optimality of SAPA-EDF under the sporadic task model (App. B of [11]) relies heavily on a sustainability property of SAPA-EDF, namely that diminishing the execution cost of any job does not increase the response time of any other job. This may not hold under window-constrained scheduling.

Fig. 3: Response time may increase when execution time decreases under window-constrained scheduling.



▷ **Ex. 6.** Consider $\tau = \{\tau_1, \tau_2, \tau_3\}$ scheduled globally on $\pi = \{\pi_1, \pi_2\}$ with $(C_1, T_1) = (C_2, T_2) = (C_3, T_3) = (2, 3)$. For the first job of each task, let $\chi_{1,1}(t) = 3$, $\chi_{2,1}(t) = 3$, and $\chi_{3,1}(t) = 8$. For the second job of each task, let $\chi_{1,2}(t) = 6$, $\chi_{2,2}(t) = 7$, and $\chi_{3,2}(t) = 3$. Fig. 3(a) shows a potential schedule for this system when jobs are period and execute to their tasks' WCET. Decreasing $C_{3,1}$ from 2 to 1 results in the schedule in Fig. 3(b). Because $C_{3,1}$ was decreased, $\tau_{3,1}$ completes earlier at time 3. As a result, the ready job of τ_3 at time 3 is $\tau_{3,2}$, which has higher priority than $\tau_{2,2}$, τ_2's ready job ($\chi_{3,2}(3) = 3 < 7 = \chi_{2,2}(3)$). Thus, $\tau_{2,2}$ does not begin execution until time 5, increasing its response time. ◁

As the needed sustainability property may not hold in our context, we cannot make the same assumptions as in [11] about execution costs and periodicity. This change percolates down into almost all proofs in this section. In eliminating these assumptions, we actually have streamlined the proof in [11].

A. Deviation

The foundation of our proof is the concept of *deviation* (Dev), which replaces the traditional notion of Lag used in [11]. The definition of deviation depends on *virtual time*.

Def. 4. The *virtual time* of task τ_i at time $t \geq 0$ is

$$vt_i(t) = \begin{cases} r_i(t) + T_i \dfrac{C_i(t) - c_i(t)}{C_i(t)} & \tau_i \text{ has ready job} \quad (1a) \\ \max(t, D_i(t)) & \text{else} \quad (1b) \end{cases}$$

.

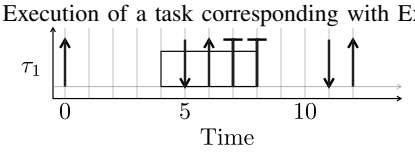
▷ **Ex. 7.** Consider a task τ_1 with $(C_1, T_1) = (3, 5)$ that executes as depicted in Fig. 4(a) over the interval $[0, 12]$. Note from Fig. 4(a) that $C_{1,1} = 3$, $C_{1,2} = 1$, and $T_{1,1} = T_{1,2} = 6$. $vt_1(t)$ over $[0, 12]$ is plotted in Fig. 4(b).

Over the interval $[0, 4)$, $\tau_{1,1}$ is ready and unscheduled. Here $r_1(t) = 0$, $C_1(t) = C_{1,1}$, and $c_1(t) = C_{1,1}$, so $vt_1(t)$ remains at 0. Over $[4, 7)$, $\tau_{1,1}$ is scheduled, causing $c_1(t)$ to decrease at a unit rate, and $vt_1(t)$ to increase at a rate of $T_1/C_1(t) = 5/3$.

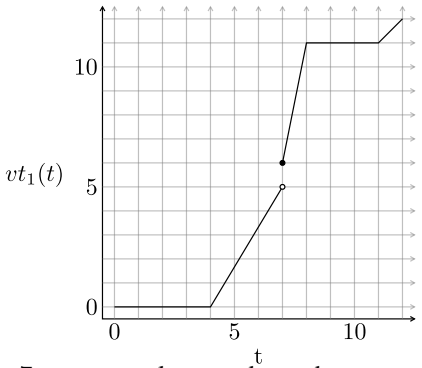
[8]The closest that we are aware of is an implementation of Strong APA FP under hierarchical affinities, presented in [2].

Fig. 4: Virtual time example.

(a) Execution of a task corresponding with Ex. 7.



(b) Corresponding $vt_1(t)$.



At time 7, $\tau_{1,1}$ completes and $\tau_{1,2}$ becomes the ready job of τ_1. This causes $r_1(t)$, $C_1(t)$, and $c_1(t)$ to change, causing a discontinuity at time 7. $\tau_{1,2}$ is completed over $[7, 8)$, causing $vt_1(t)$ to increase at a rate of $T_1/C_1(t) = 5/1$ over this interval. At time 8, τ_1 no longer has a ready job, and so $vt_1(t)$ remains constant at $D_1(t) = 11$ until time 11. Past time 11, the time t is greater than $D_1(t)$, so $vt_1(t)$ increases at a unit rate with time until $\tau_{1,3}$ is released at time 12. ◁

Dev compares a task's virtual time against actual time.

Def. 5. For task τ_i at time t, $\mathsf{Dev}(\tau_i, t) = u_i(t - vt_i(t))$. For the subset of tasks τ', $\mathsf{Dev}(\tau', t) = \sum_{\tau_i \in \tau'} \mathsf{Dev}(\tau_i, t)$.

Lemma 1. *If $\mathsf{Dev}(\tau_i, t) > 0$, then τ_i has a ready job at t.*

Proof. We prove the contrapositive: for every time instant t, if task τ_i does not have a ready job, then $\mathsf{Dev}(\tau_i, t) \leq 0$. By Def. 4, $vt_i(t) = \max(t, D_i(t))$.

If $vt_i(t) = D_i(t)$, we have $D_i(t) \geq t$. By Def. 5, we have $\mathsf{Dev}(\tau_i, t) = u_i(t - vt_i(t)) = u_i(t - D_i(t)) \leq u_i(t - t) = 0$.

If $vt_i(t) = t$, then by Def. 5, we have $\mathsf{Dev}(\tau_i, t) = u_i(t - vt_i(t)) = u_i(t - t) = 0$. □

Lemma 2. *If task τ_i has a ready job at time t, then*

$$t - \frac{\mathsf{Dev}(\tau_i, t)}{u_i} < d_i(t) \leq t - \frac{\mathsf{Dev}(\tau_i, t)}{u_i} + T_i. \tag{2}$$

Proof. By Def. 4 and because τ_i has a ready job at t, we have $vt_i(t) = r_i(t) + T_i \frac{C_i(t) - c_i(t)}{C_i(t)}$. By the definition of $c_i(t)$, we have $0 < c_i(t) \leq C_i(t)$. Thus, $vt_i(t) + T_i \geq r_i(t) + T_i \frac{0}{C_i(t)} + T_i = r_i(t) + T_i = d_i(t)$ and $vt_i(t) < r_i(t) + T_i \frac{C_i(t)}{C_i(t)} = r_i(t) + T_i = d_i(t)$. Thus, we have $vt_i(t) < d_i(t) \leq vt_i(t) + T_i$, which by Def. 5 is equivalent to (2). □

Corollary 1. *If for some $L > 0$ we have $\forall t \geq 0 : \mathsf{Dev}(\tau_i, t) \leq L$, then the tardiness of any job of τ_i is at most L/u_i.*

Proof. We prove the contrapositive: if some job of τ_i has tardiness exceeding L/u_i, then $\exists t \geq 0 : \mathsf{Dev}(\tau_i, t) > L$ holds.

Let $\tau_{i,j}$ be a job with tardiness exceeding L/u_i. Consider time $t = d_{i,j} + L/u_i$. Because the tardiness of $\tau_{i,j}$ exceeds L/u_i, job $\tau_{i,j}$ is pending at t. Also, because τ_i's ready job at t cannot have been released later than $\tau_{i,j}$, we have $d_i(t) \leq d_{i,j}$. Thus, by Lemma 2, $t - \mathsf{Dev}(\tau_i, t)/u_i < d_{i,j}$. By the definition of t above, this implies that $\mathsf{Dev}(\tau_i, t) > L$ holds. □

Lemma 3. *If at time t, tasks τ_e and τ_ℓ have ready jobs $\tau_{e,h}$ and $\tau_{\ell,k}$ and we have*

$$\frac{\mathsf{Dev}(\tau_e, t)}{u_e} \geq \frac{\mathsf{Dev}(\tau_\ell, t)}{u_\ell} + T_{max} + 2\phi, \tag{3}$$

then $\chi_{e,h}(t) < \chi_{\ell,k}(t)$. ($e$ signifies "earlier" and ℓ "later".)

Proof.
$$\begin{aligned}
&\chi_{e,h}(t) \\
&\leq d_{e,h} + \phi && \{\text{By Def. 2}\} \\
&= d_e(t) + \phi && \{\text{By the definition of ready}\} \\
&\leq t - \frac{\mathsf{Dev}(\tau_e, t)}{u_e} + T_e + \phi && \{\text{By Lemma 2}\} \\
&\leq t - \frac{\mathsf{Dev}(\tau_e, t)}{u_e} + T_{\max} + \phi && \{T_e \leq T_{\max}\} \\
&\leq t - \frac{\mathsf{Dev}(\tau_\ell, t)}{u_\ell} - \phi && \{\text{By (3)}\} \\
&< d_\ell(t) - \phi && \{\text{By Lemma 2}\} \\
&= d_{\ell,k} - \phi && \{\text{By the definition of ready}\} \\
&\leq \chi_{\ell,k}(t) && \{\text{By Def. 2}\}
\end{aligned}$$ □

Lemma 4. *For τ_i, $\epsilon > 0$, and $t \geq 0$, $vt_i(t + \epsilon) \geq vt_i(t)$.*

Proof. Assume without loss of generality that ϵ is arbitrarily small. There are four cases to consider.

Case 4.1. *τ_i has a ready job at both t and $t+\epsilon$.* For arbitrarily small ϵ, τ_i has a ready job throughout $[t, t + \epsilon]$ and the ready job of τ_i changes at most once.

Suppose the ready job of τ_i does not change over $[t, t + \epsilon]$. Because the remaining execution required by a job does not increase with time, $c_i(t) \geq c_i(t + \epsilon)$. By (1a) of Def. 4, $vt_i(t + \epsilon) = r_i(t + \epsilon) + T_i \frac{C_i(t+\epsilon) - c_i(t+\epsilon)}{C_i(t+\epsilon)}$. Since the ready job of τ_i does not change, the latter expression equals $r_i(t) + T_i \frac{C_i(t) - c_i(t+\epsilon)}{C_i(t)} \geq r_i(t) + T_i \frac{C_i(t) - c_i(t)}{C_i(t)} = vt_i(t)$, where the last step follows from (1a) of Def. 4.

Otherwise, suppose $\tau_{i,j}$ is ready at t and $\tau_{i,j+1}$ is ready at $t + \epsilon$. By (1a) of Def. 4, $vt_i(t + \epsilon) = r_i(t + \epsilon) + T_i \frac{C_i(t+\epsilon) - c_i(t+\epsilon)}{C_i(t+\epsilon)} = r_{i,j+1} + T_i \frac{C_{i,j+1} - c_i(t+\epsilon)}{C_{i,j+1}}$. Because $c_i(t + \epsilon) \leq C_{i,j+1}$, the latter expression is at least $r_{i,j+1} + T_i \frac{C_{i,j+1} - C_{i,j+1}}{C_{i,j+1}} = r_{i,j+1} \geq r_{i,j} + T_i = r_{i,j} + T_i \frac{C_{i,j}}{C_{i,j}} \geq r_i(t) + T_i \frac{C_i(t) - c_i(t)}{C_i(t)} = vt_i(t)$, where the last step follows by (1a) of Def. 4.

Case 4.2. *τ_i has a ready job at t but not at $t + \epsilon$.* As τ_i transitioned from having a ready job to not having one, the ready job $\tau_{i,j}$ of τ_i at t completed by $t + \epsilon$. This implies $d_{i,j} \leq D_i(t+\epsilon)$. Also, because $\tau_{i,j}$ is ready at t, $d_i(t) = d_{i,j}$. By (1b) of Def. 4, $vt_i(t + \epsilon) = \max(t + \epsilon, D_i(t + \epsilon)) \geq D_i(t + \epsilon) \geq d_{i,j} = d_i(t) = r_i(t) + T_i$. Because $c_i(t) > 0$,

$r_i(t)+T_i > r_i(t)+T_i\frac{C_i(t)-c_i(t)}{C_i(t)} = vt_i(t)$, where the last step follows by (1a) of Def. 4.

Case 4.3. *No job of τ_i is ready at both t and $t+\epsilon$.* For arbitrarily small ϵ, no job of τ_i is released or completed within $[t, t+\epsilon]$. Thus, τ_i never has a ready job over this interval and $D_i(t+\epsilon) = D_i(t)$. By (1b) of Def. 4, $vt_i(t+\epsilon) = \max(t+\epsilon, D_i(t+\epsilon)) = \max(t+\epsilon, D_i(t)) \geq \max(t, D_i(t)) = vt_i(t)$.

Case 4.4. *τ_i has no ready job at t but does at $t+\epsilon$.* Because τ_i transitioned from not having a ready job to having one, the ready job $\tau_{i,j}$ of τ_i at $t+\epsilon$ is such that $t < r_{i,j} \leq t+\epsilon$. Also, because jobs of τ_i are released at least T_i time units apart, $D_i(t) \leq r_{i,j}$. By (1a) of Def. 4, $vt_i(t+\epsilon) = r_i(t+\epsilon)+T_i\frac{C_i(t+\epsilon)-c_i(t+\epsilon)}{C_i(t+\epsilon)} \geq r_i(t+\epsilon)$, where the last step follows from $c_i(t+\epsilon) \leq C_i(t+\epsilon)$. Because $\tau_{i,j}$ is ready at $t+\epsilon$, $r_i(t+\epsilon) = r_{i,j} \geq \max(t, D_i(t)) = vt_i(t)$, where the last step follows by (1b) of Def. 4.

For all cases, $vt_i(t+\epsilon) \geq vt_i(t)$. □

Lemma 5. *For any task set $\tau' \subseteq \tau$ and time t, if $\forall \delta > 0 : \exists \epsilon \in (0, \delta] : \mathsf{Dev}(\tau', t+\epsilon) > L$ holds, then $\mathsf{Dev}(\tau', t) \geq L$.*

Proof. Assume, to the contrary, that $\forall \delta > 0 : \exists \epsilon \in (0, \delta] : \mathsf{Dev}(\tau', t+\epsilon) > L$ and $\mathsf{Dev}(\tau', t) < L$ both hold. Let $\delta' = (L - \mathsf{Dev}(\tau', t))/(|\tau'|u_{\max}) > 0$. Then, $\forall \delta > 0 : \exists \epsilon \in (0, \delta] : \mathsf{Dev}(\tau', t+\epsilon) > L = |\tau'|u_{\max}\delta' + \mathsf{Dev}(\tau', t)$.

This implies that $\forall \delta > 0 : \exists \epsilon \in (0, \delta] : \mathsf{Dev}(\tau', t+\epsilon) - \mathsf{Dev}(\tau', t) > |\tau'|u_{\max}\delta'$ holds. By Def. 5, $\mathsf{Dev}(\tau', t+\epsilon) - \mathsf{Dev}(\tau', t) = \sum_{\tau_i \in \tau'} (\mathsf{Dev}(\tau_i, t+\epsilon) - \mathsf{Dev}(\tau_i, t))$. Because the maximum of a finite set must be at least the arithmetic mean of the set, we have $\forall \delta > 0 : \exists \epsilon \in (0, \delta] : \exists \tau_i \in \tau' : \mathsf{Dev}(\tau_i, t+\epsilon) - \mathsf{Dev}(\tau_i, t) > |\tau'|u_{\max}\delta'/|\tau'| = u_{\max}\delta'$.

Thus, $\exists \epsilon \in (0, \delta'] : \exists \tau_i \in \tau' : \mathsf{Dev}(\tau_i, t+\epsilon) - \mathsf{Dev}(\tau_i, t) > u_{\max}\delta'$. By Def. 5, $\mathsf{Dev}(\tau_i, t+\epsilon) - \mathsf{Dev}(\tau_i, t) = u_i(t+\epsilon - vt_i(t+\epsilon)) - u_i(t - vt_i(t)) = u_i(\epsilon + vt_i(t) - vt_i(t+\epsilon)) > u_{\max}\delta'$. Thus, $\epsilon + vt_i(t) - vt_i(t+\epsilon) > (u_{\max}/u_i)\delta' \geq \delta'$. Rearranging yields $vt_i(t+\epsilon) - vt_i(t) < \epsilon - \delta' \leq 0$, which contradicts Lemma 4. □

Lemma 6. *For arbitrarily small $\epsilon > 0$, if task τ_i is scheduled over $[t, t+\epsilon]$, then $\mathsf{Dev}(\tau_i, t+\epsilon) \leq \mathsf{Dev}(\tau_i, t) + \epsilon(u_i - 1)$.*

Proof. For small enough ϵ, the same job of τ_i is scheduled over $[t, t+\epsilon]$ and this job does not complete in this interval.

By Def. 5, $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - vt_i(t+\epsilon))$. Because τ_i is scheduled, it must have a ready job, therefore only case (1a) of Def. 4 can apply. Thus, $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i\left(t+\epsilon - r_i(t+\epsilon) - T_i\frac{C_i(t+\epsilon)-c_i(t+\epsilon)}{C_i(t+\epsilon)}\right)$.

As the ready job of τ_i at t does not change by $t+\epsilon$, $r_i(t+\epsilon) = r_i(t)$ and $C_i(t+\epsilon) = C_i(t)$. Since ϵ units of execution of this ready job are completed, $c_i(t+\epsilon) = c_i(t) - \epsilon$. Thus,

$$\begin{aligned}
&\mathsf{Dev}(\tau_i, t+\epsilon) \\
&= u_i\left(t+\epsilon - r_i(t) - T_i\frac{C_i(t) - c_i(t) + \epsilon}{C_i(t)}\right) \\
&= u_i(t - vt_i(t)) + u_i\epsilon - u_i\frac{T_i}{C_i(t)}\epsilon \qquad \{\text{By Def. 4}\} \\
&\leq \mathsf{Dev}(\tau_i, t) + \epsilon(u_i - 1) \ \{\text{By Def. 5 and } T_i/C_i(t) \geq 1/u_i\}.
\end{aligned}$$ □

Lemma 7. *For arbitrarily small $\epsilon > 0$, if task τ_i is not scheduled over $[t, t+\epsilon]$, then $\mathsf{Dev}(\tau_i, t+\epsilon) \leq \mathsf{Dev}(\tau_i, t) + \epsilon u_i$.*

Proof. By Def. 5, $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - vt_i(t+\epsilon))$. For small enough ϵ, τ_i either has a ready job or does not over all of $[t, t+\epsilon]$.

Case 7.1. *τ_i has a ready job over $[t, t+\epsilon]$.* By (1a), we have $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i\left(t+\epsilon - r_i(t+\epsilon) - T_i\frac{C_i(t+\epsilon)-c_i(t+\epsilon)}{C_i(t+\epsilon)}\right)$. Because the ready job of τ_i does not change while τ_i is unscheduled over $[t, t+\epsilon]$, we have $r_i(t+\epsilon) = r_i(t)$, $C_i(t+\epsilon) = C_i(t)$, and $c_i(t+\epsilon) = c_i(t)$. Thus, by Defs. 4 and 5, $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i\left(t+\epsilon - r_i(t) - T_i\frac{C_i(t)-c_i(t)}{C_i(t)}\right) = u_i(t - vt_i(t)) + \epsilon u_i = \mathsf{Dev}(\tau_i, t) + \epsilon u_i$.

Case 7.2. *τ_i has no ready job over $[t, t+\epsilon]$.* $D_i(t+\epsilon) = D_i(t)$ because τ_i is not scheduled over $[t, t+\epsilon]$. Thus, $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - \max(t+\epsilon, D_i(t+\epsilon))) = u_i(t+\epsilon - \max(t+\epsilon, D_i(t)))$. Either $D_i(t) \leq t$, $t < D_i(t) \leq t+\epsilon$, or $D_i(t) > t+\epsilon$.

If $D_i(t) \leq t$, then $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - t - \epsilon) \leq u_i(t-t) + \epsilon u_i = u_i(t - \max(t, D_i(t))) + \epsilon u_i = \mathsf{Dev}(\tau_i, t) + \epsilon u_i$.

If $t < D_i(t) \leq t+\epsilon$, then $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - t - \epsilon) = u_i(t - t - \epsilon) + \epsilon u_i \leq u_i(t - D_i(t)) + \epsilon u_i = \mathsf{Dev}(\tau_i, t) + \epsilon u_i$.

If $D_i(t) > t+\epsilon$, then $\mathsf{Dev}(\tau_i, t+\epsilon) = u_i(t+\epsilon - D_i(t)) = u_i(t - D_i(t)) + \epsilon u_i = \mathsf{Dev}(\tau_i, t) + \epsilon u_i$.

For all cases, $\mathsf{Dev}(\tau_i, t+\epsilon) \leq \mathsf{Dev}(\tau_i, t) + \epsilon u_i$. □

Lemma 8 is unchanged from [11] (except being slightly reworded), so we omit its proof. (It concerns feasibility, so it is not impacted by focusing on window-constrained schedulers.)

Lemma 8. *(Lemma 35 of [11]) If a task system τ is feasible, then for any set of tasks $\tau' \subseteq \tau$, the maximum number of processors M that tasks of τ' may be simultaneously scheduled upon is such that $M \geq U(\tau')$.*

Corollary 2. *Let τ be a feasible task system. Under a Strong APA scheduler, at any time instant t, for any set of tasks $\tau' \subseteq \tau$ with ready jobs such that the tasks of τ' have highest priority out of all tasks with ready jobs, $\exists \delta > 0 : \forall \epsilon \in (0, \delta] : \mathsf{Dev}(\tau', t) \geq \mathsf{Dev}(\tau', t+\epsilon)$ holds.*

Proof. Let M be the number of processors scheduling tasks of τ' at t. M is maximal due to Strong APA. Let δ be arbitrarily small such that any task scheduled (resp., unscheduled) at t is scheduled (resp., unscheduled) over $[t, t+\delta]$. Let $\tau^s \subseteq \tau'$ be the subset of tasks in τ' that are scheduled over $[t, t+\delta]$.

Because $\epsilon \in (0, \delta]$, these tasks are also the tasks scheduled over $[t, t+\epsilon]$. Thus,

$$\begin{aligned}
&\mathsf{Dev}(\tau', t+\epsilon) \\
&= \mathsf{Dev}(\tau^s, t+\epsilon) + \mathsf{Dev}(\tau' \setminus \tau^s, t+\epsilon) \\
&\le \mathsf{Dev}(\tau^s, t) + \epsilon(U(\tau^s) - |\tau^s|) + \mathsf{Dev}(\tau' \setminus \tau^s, t) \\
&\quad + \epsilon U(\tau' \setminus \tau^s) && \{\text{By Lemmas 6 and 7}\} \\
&= \epsilon(U(\tau') - M) + \mathsf{Dev}(\tau', t) && \{\text{Def. of } M \text{ and } U(\tau')\} \\
&\le \mathsf{Dev}(\tau', t) && \{\text{By Lemma 8}\}. \square
\end{aligned}$$

B. Modified Invariant

The rest of the proof entails proving that the vector-valued function $\langle \mathsf{Dev}(\tau_1, t), \mathsf{Dev}(\tau_2, t), \ldots, \mathsf{Dev}(\tau_n, t)\rangle$ stays in a bounded region of $\mathbb{R}^n$ for all $t \ge 0$. The same approach is used in [11], though we have used Dev instead of Lag. The proof in [11] relied on the fact that Lag is continuous, which is not true of Dev. We show that continuity is not necessary; that Dev satisfies Lemma 5 is sufficient for our proof. Our bounded region is also larger than in [11] since we consider window-constrained prioritizations instead of only EDF.

Lemma 9. *For any feasible task system and Strong APA window-constrained scheduler, $\forall t \ge 0$,*

$$\forall \tau' \subseteq \tau : \mathsf{Dev}(\tau', t) \le \beta(\tau') \tag{4}$$

where $\beta(\tau')$ is defined as

$$\forall \tau' \subseteq \tau : \beta(\tau') = \frac{T_{max} + 2\phi}{2u_{min}} U(\tau')(2U(\tau) - U(\tau')). \tag{5}$$

Proof. As a shorthand, let $K = (T_{\max} + 2\phi)/(2u_{\min}) > 0$. By Def. 5 and (5), (4) is true at $t = 0$. Assuming (4) is falsified, let t_b (b for "boundary") be the last time instant such that (4) is true over $[0, t_b)$. We prove the lemma by contradicting the definition of t_b.

► **Claim 9.1.** (4) is true over $[0, t_b]$.

Proof. We will prove by contradiction that (4) holds at t_b. Assume, to the contrary, that (4) is true over $[0, t_b)$ but not at t_b. Let $\epsilon = (\mathsf{Dev}(\tau', t_b) - \beta(\tau'))/(|\tau'|u_{\max}) > 0$ and $t' \in (\max(t_b - \epsilon, 0), t_b)$. Then $t' \in [0, t_b)$. Thus, $\mathsf{Dev}(\tau', t') \le \beta(\tau')$, and so $\mathsf{Dev}(\tau', t_b) - \mathsf{Dev}(\tau', t') \ge \mathsf{Dev}(\tau', t_b) - \beta(\tau')$. By the definition of ϵ, $\mathsf{Dev}(\tau', t_b) - \mathsf{Dev}(\tau', t') \ge \epsilon|\tau'|u_{\max}$.

By Def. 5, $\sum_{\tau_i \in \tau'} u_i(t_b - vt_i(t_b) - t' + vt_i(t')) \ge \epsilon|\tau'|u_{\max}$. Because the maximum of a finite set must be at least the arithmetic mean of the set, we have $\exists \tau_i \in \tau' : u_i(t_b - vt_i(t_b) - t' + vt_i(t')) \ge \epsilon|\tau'|u_{\max}/|\tau'| = \epsilon u_{\max}$. Because $u_{\max} \ge u_i$, $t_b - vt_i(t_b) - t' + vt_i(t') \ge \epsilon$. Rearranging yields $vt_i(t') - vt_i(t_b) \ge \epsilon - t_b + t'$. Because $t' > \max(t_b - \epsilon, 0) \ge t_b - \epsilon$, $vt_i(t') - vt_i(t_b) > \epsilon - t_b + t_b - \epsilon = 0$. Thus, $vt_i(t') > vt_i(t_b)$. As $t' < t_b$, this contradicts Lemma 4. ◄

► **Claim 9.2.** At time t_b,

$$\forall \tau' \subseteq \tau : \mathsf{Dev}(\tau', t_b) \le \beta(\tau'), \tag{6}$$

$$\exists \tau^b \subseteq \tau : \forall \delta > 0 : \exists \epsilon \in (0, \delta] : \mathsf{Dev}(\tau^b, t_b + \epsilon) > \beta(\tau^b) \tag{7}$$

$$\wedge\, \mathsf{Dev}(\tau^b, t_b) = \beta(\tau^b). \tag{8}$$

Proof. (6) follows from Claim 9.1.

If (7) is false, then $\forall \tau^b \subseteq \tau : \exists \delta > 0 : \forall \epsilon \in (0, \delta] : \mathsf{Dev}(\tau^b, t_b + \epsilon) \le \beta(\tau^b)$. This means that (4) is true over $(t_b, t_b + \delta]$, which contradicts the definition of t_b.

(8) follows from (6), (7), and Lemma 5. ◄

Let τ^b be any of the task subsets known to exist by (7).

► **Claim 9.3.** For any task $\tau_e \in \tau^b$,

$$\mathsf{Dev}(\tau_e, t_b) \ge K(2U(\tau) - 2U(\tau^b) + u_e)u_e > 0.$$

Proof.

$$\begin{aligned}
&\mathsf{Dev}(\tau_e, t_b) \\
&= \mathsf{Dev}(\tau^b, t_b) - \mathsf{Dev}(\tau^b \setminus \{\tau_e\}, t_b) \\
&\ge \beta(\tau^b) - \beta(\tau^b \setminus \{\tau_e\}) && \{\text{By (6) and (8)}\} \\
&= KU(\tau^b)(2U(\tau) - U(\tau^b)) - \\
&\quad KU(\tau^b \setminus \{\tau_e\})(2U(\tau) - U(\tau^b \setminus \{\tau_e\})) && \{\text{By (5)}\} \\
&= K(2U(\tau) - 2U(\tau^b) + u_e)u_e \\
&> 0 && \{U(\tau) \ge U(\tau^b)\}
\end{aligned}$$ ◄

► **Claim 9.4.** Any task τ_e in τ^b has a ready job at t_b.

Proof. Follows from Lemma 1 and Claim 9.3. ◄

► **Claim 9.5.** For any task $\tau_\ell \notin \tau^b$,

$$\mathsf{Dev}(\tau_\ell, t_b) \le K(2U(\tau) - 2U(\tau^b) - u_\ell)u_\ell$$

Proof.

$$\begin{aligned}
&\mathsf{Dev}(\tau_\ell, t_b) \\
&= \mathsf{Dev}(\tau^b \cup \{\tau_\ell\}, t_b) - \mathsf{Dev}(\tau^b, t_b) \\
&\le \beta(\tau^b \cup \{\tau_\ell\}) - \beta(\tau^b) && \{\text{By (6) and (8)}\} \\
&= KU(\tau^b \cup \{\tau_\ell\})(2U(\tau) - U(\tau^b \cup \{\tau_\ell\})) - \\
&\quad KU(\tau^b)(2U(\tau) - U(\tau^b)) && \{\text{By (5)}\} \\
&= K(2U(\tau) - 2U(\tau^b) - u_\ell)u_\ell
\end{aligned}$$ ◄

► **Claim 9.6.** For any tasks $\tau_e \in \tau^b$ and $\tau_\ell \notin \tau^b$, if τ_ℓ has a ready job at t_b, then τ_e has higher priority than τ_ℓ at t_b.

Proof.

$$\begin{aligned}
&\frac{\mathsf{Dev}(\tau_e, t_b)}{u_e} - \frac{\mathsf{Dev}(\tau_\ell, t_b)}{u_\ell} \\
&\ge K(u_e + u_\ell) && \{\text{By Claims 9.3 and 9.5}\} \\
&\ge T_{\max} + 2\phi && \{u_e + u_\ell \ge 2u_{\min}\}
\end{aligned}$$

The claim follows from Claim 9.4 and Lemma 3. ◄

By Claim 9.6, Corollary 2, and (8), there exists $\delta > 0$ such that for all $\epsilon \in (0, \delta]$, $\mathsf{Dev}(\tau^b, t_b + \epsilon) \le \mathsf{Dev}(\tau_i^b, t_b) = \beta(\tau^b)$. This contradicts (7). Thus, t_b does not exist, implying that (4) is true over $[0, \infty)$. □

Theorem 1. *For any feasible task system τ, the tardiness of any task τ_i under any Strong APA window-constrained scheduler is at most*

$$\frac{T_{max} + 2\phi}{2u_{min}}(2U(\tau) - u_i). \tag{9}$$

Proof. By Lemma 9, $\mathsf{Dev}(\tau_i, t) \le \frac{T_{\max} + 2\phi}{2u_{\min}} u_i(2U(\tau) - u_i)$ for all $t \ge 0$. By Corollary 1, the bound in (9) follows. □

Corollary 3. *For any system as in Thm. 1, the tardiness of any task is at most*

$$m(T_{max} + 2\phi)/u_{min}. \tag{10}$$

Proof. Because $m \geq U(\tau)$ (by Lemma 8), $m > (2U(\tau) - u_i)/2$ for any task τ_i. Thus, the expression in (10) is greater than that in (9) from Thm. 1. □

Using (10) over (9) will simplify the NP-blocking analysis presented later.

As a brief aside, much like how [11] considered identical multiprocessors under APA and uniform multiprocessors, our proof of Thm. 1 also applies to uniform multiprocessors with minor modifications. These modifications are detailed online [10].

V. Non-preemptivity

In this section, we define progression, a notion that can replace NP execution in some instances. We use Thm. 1 to show that, unlike NP execution, progression does not lead to capacity loss. For instances where NP execution is required in its strictest sense, we present modifications to SAPA-EDF under which we can compute blocking terms.

A. Progression

We begin by formally defining progression.

Def. 6. A portion of a job is guaranteed *progression* if, once the portion is first scheduled, it is always scheduled on some processor until the portion ends.

We do not consider time spent migrating as unscheduled time. Thus, progression is weaker than true NP execution because the considered job may migrate.

Though progression is weaker than NP execution, it is sufficient in many use cases. Locking protocols are a good example. An NP FIFO spin-lock ensures $O(m)$ blocking on m processors because NP execution guarantees that the lock's spin queue may contain at most m tasks. Progression provides this same guarantee. Of course, other use cases, such as NP execution that occurs within certain OS system calls, may be intolerant of the migrations permitted by progression.

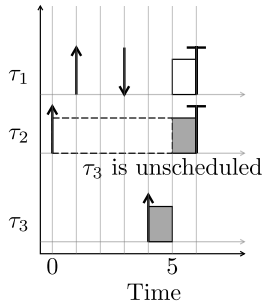
Code sections that require progression can be realized under SAPA-EDF by prioritizing a job by its release time instead of its deadline while within such a section. We call this new scheduler *Strong APA EDF-FIFO (SAPA-EF)* because it uses a combination of EDF and FIFO priorities. These priorities are window-constrained, with $\phi = T_{\max}$, so by Thm 1, SAPA-EF is SRT-optimal with tardiness at most $3mT_{\max}/u_{\min}$.

SAPA-EF guarantees progression for the same reason that global FIFO ensures NP execution. It is impossible for any newly released job to preempt an executing job under global FIFO because the newly released job must have a later release time than any executing job. Recall that under Strong APA, a job is unscheduled when it is at the end of a shifting path that begins with a higher-priority job. Similarly to global FIFO, it is impossible for any newly released job to have an earlier deadline than the release time of any job already executing. Thus, no job release can cause an executing job whose priority is its release time to be unscheduled via shifting. Such an executing job can, however, be caused to migrate, a behavior that true NP execution would disallow.

B. Link-Based Scheduling

As noted above, true NP execution may be required in some use cases. Unfortunately, SAPA-EF does not guarantee progression if NP sections exist.

Fig. 5: NP execution (denoted with dashes) breaks progression.



▷ **Ex. 8.** Consider $\tau_1, \ldots, \tau_3$ in Fig. 1(a) executing on π_1 and π_2. Let $(C_1, T_1) = (1, 2)$, $(C_2, T_2) = (6, 20)$, and $(C_3, T_3) = (2, 10)$. Consider the schedule in Fig. 5. Initially, τ_2 releases a job at $t = 0$ and is scheduled upon π_1. τ_2 immediately begins executing an NP section. τ_2 also immediately requests progression for the entire duration of its job, changing its priority from 20 to 0. Fixed task τ_1 releases a job at $t = 1$, but is unable to execute because τ_2 is in an NP section on π_1. Meanwhile, τ_3 releases a job at $t = 4$, and immediately begins executing on π_2, which was idle. τ_3 requests progression at $t = 4$, changing its priority from 14 to 4. However, at $t = 5$, τ_2 completes its NP section. Because it requested progression for its entire job, τ_1 and τ_2 have the highest-priority jobs (with priorities of 0 and 3, compared to 4 for τ_3). Moreover, τ_1 can now execute on π_1. Thus, at $t = 5$, τ_1 shifts via path $\{\tau_1, \pi_1, \tau_2, \pi_2, \tau_3\}$ and τ_3 is unscheduled, violating its progression guarantee. ◁

There is little value in trying to replace some instances of non-preemptivity with progression if the remaining NP sections break progression. To support NP sections and progression simultaneously, we adapt link-based scheduling [3], [4] for SAPA-EF. Under link-based scheduling, a distinction is made between what task is *actually* scheduled on a processor (the scheduled task) and what task *would* have been scheduled on this processor (the linked task) if NP sections were ignored. The linked task of every processor is updated whenever a task changes its priority (e.g., job releases or completions or via a prioritization function) and is based purely on these priorities, while actual scheduling decisions will only match linked tasks if doing so will not migrate a job in an NP section.

We assume that every job requires progression throughout its execution. This will be necessary to guarantee that any job is NP-blocked at most once, which is desirable because NP-blocking is fundamentally more pessimistic under APA than under global scheduling, as will be demonstrated in Ex. 10. Formally, our link-based SAPA-EF (L-SAPA-EF) scheduler is defined by the following rules.

R1 The priority of job $\tau_{i,j}$ is initially $d_{i,j}$, and permanently changes to $r_{i,j}$ the instant it is first linked.

R2 Tasks are linked onto processors such that if every processor scheduled its linked task, the scheduler would maintain Strong APA.

R3 A processor executing a job continues executing said job until it either completes or is scheduled by another processor, at which point the processor becomes idle.

R4 An idle processor with a linked task schedules the ready job of its linked task and ceases to be idle, unless said task is executing an NP section on another processor.

Our assumption that every job requires progression is reflected in R1. This assumption is guaranteed by R3.

▷ **Ex. 9.** We find it easier to demonstrate linking by examining affinity graphs rather than schedules. Consider Fig. 6. Initially, τ_3 is the first to release a job. τ_3 becomes linked to π_2. By R4, π_2 begins executing τ_3's job. τ_3 then enters a NP section, which reflects the first affinity graph in Fig. 6.

After this, τ_2 releases a job. Because τ_3 is already linked on π_2, τ_2 is linked to π_1. By R4, π_1 begins executing τ_2's job. This job also enters an NP section.

Next, τ_1 releases a job. As τ_2 is linked on π_1 and τ_3 is linked on π_2, there is a path $\{\tau_1, \pi_1, \tau_2, \pi_2, \tau_3, \pi_3, \tau_0\}$ over which a shift in links may occur. By R2, this shift is taken such that τ_1 is linked on π_1, τ_2 is linked on π_2, and τ_3 is linked on π_3. However, as π_1 and π_2 are still executing non-preemptively, no migrations occur. This reflects the third graph.

Next, τ_2 finishes its NP section. By R3, π_1 continues executing τ_2 even though its linked task is now τ_1. Note that since τ_3 is executing an NP section on π_2, having π_1 schedule τ_1 at this time would have broken progression for τ_2. This reflects the fourth graph.

Lastly, τ_3 completes its NP section. By R4, π_3 schedules its linked task τ_3, thereby making π_2 idle. Thus, π_2 schedules its linked task τ_2, thereby making π_1 idle. This makes π_1 schedule its linked task τ_1. This reflects the fifth graph. ◁

Besides R1 and R2, which are specific to EDF-FIFO and APA scheduling, respectively, link-based scheduling under R3 and R4 differs slightly from how it is defined in [3]. Under R3, a processor may continue executing a task besides the one it is linked to, even if neither it nor its linked task are executing NP sections. Additionally, under R4, processors only attempt to schedule their linked tasks when they become idle. This is distinct from the original definition, where a processor will reschedule as soon as its current task is no longer its linked task and neither its current nor its linked task is executing an NP section. This change to the original definition is necessary to avoid breaking progression, which was not a concern in [3].

Under global scheduling, linking ensures that a job is blocked for at most one NP section [3]. In our context, each job is similarly blocked at most once. However, in our case, the length of blocking may be due to a sequence of NP sections.

▷ **Ex. 10.** Consider the assignments in Fig. 7. Assume that tasks are prioritized on an EDF basis and for any time t considered in this example, $d_2(t) < d_3(t) < d_4(t) < d_5(t) < d_1(t)$. Initially, τ_5 is unable to shift over the path $\{\tau_5, \pi_3, \tau_4, \pi_2, \tau_2, \pi_1, \tau_1\}$ because τ_4 is executing an NP section on π_3. Thus, τ_5 is NP blocked for the length of τ_4's NP section, but before this NP section completes, τ_2 begins an NP section on π_2. Thus, even after τ_4 completes its NP section, τ_5 cannot execute on π_3 without unscheduling τ_4, which has higher priority. This pattern may continue to the end of the shifting path, forcing τ_5 to wait for a sequence of NP sections. ◁

C. Blocking Analysis

We now present blocking analysis for L-SAPA-EF. Due to space constraints and our usage of fairly standard blocking-analysis techniques, we cover this analysis at a high level and defer formal details to the online version of this work [10].

It is common practice to account for NP blocking by adding a blocking term B_i to the WCET of every task τ_i, where B_i upper bounds the duration that task τ_i can be NP blocked. Because links reflect tasks' priorities, NP blocking occurs when a task is linked while unscheduled.

Observe in the rightmost affinity graph in Fig. 6 that whenever no jobs are executing NP sections, every linked task is scheduled. This is true generally due to R1–R4. By R1, R2, and the fact that SAPA-EF guarantees progression, any linked task continues to be linked on some processor until its ready job completes. Additionally, by R4, only linked tasks are ever scheduled for execution, and, by R3, any scheduled task executes until its ready jobs complete. Thus, all scheduled task are also linked tasks. Now suppose some linked task is unscheduled while no job is executing an NP section. By R4, the processor the task is linked on must be executing some other task that, because all scheduled tasks are linked, must be linked on some other processor that, by R4, must also be executing some other linked task, and so forth. As the task system is not infinite, it must be that all linked tasks are scheduled when no job is executing an NP section.

Let I_i denote the largest interval in which, at any time, some job not from τ_i (a task does not NP block itself) is executing an NP section. Because all linked tasks are scheduled when no jobs are executing an NP section, the duration a task τ_i can be continuously linked and unscheduled is upper bounded by $|I_i|$. Furthermore, because L-SAPA-EF guarantees progression for all jobs, any job can be linked and unscheduled (and hence, NP blocked) at most once. Thus, it is sufficient if $B_i \geq |I_i|$.

For τ_i, let the *NP demand bound function* $\text{npdbf}_i(t, \Delta)$ be the maximum amount of NP execution required by jobs not from τ_i that are pending at some point in $[t, t+\Delta)$. Note that $\max_{t \geq 0} (\text{npdbf}_i(t, |I_i|)) \geq |I_i|$, else it is impossible for some job to be executing an NP section at every time instant in I_i.

If a function $g(\Delta)$ and $\Delta' \geq 0$ exist such that $g(\Delta)$ upper bounds $\text{npdbf}_i(t, \Delta)$ and $\forall \Delta > \Delta' : g(\Delta) < \Delta$, then $|I_i| \leq \Delta'$. The set of jobs of task $\tau_k \neq \tau_i$ considered by $\text{npdbf}_i(t, \Delta)$ includes jobs released in $[t, t+\Delta)$ and jobs released prior to t that are incomplete at t. The former is upper bounded by $\lceil \Delta / T_k \rceil$. Assuming the response time of any job is at most

Fig. 6: Linking example.

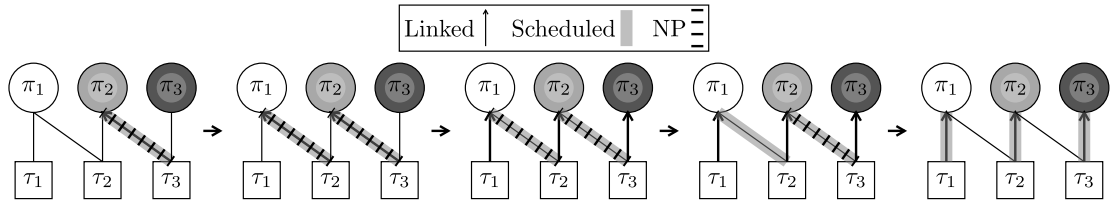
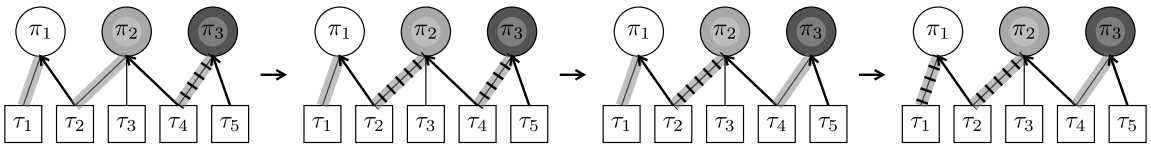


Fig. 7: Example of simultaneous NP blocking.



$R_k = T_k + 3mT_{\max}/u_{\min}$,[9] the latter is bounded by $\lceil R_k/T_k \rceil$. Let γ_k denote the maximum amount of time any job of τ_k will execute non-preemptively and let $\omega_k = \gamma_k/T_k$. Let

$$R^i_{\max} = \max_{\tau_k \in \tau \setminus \{\tau_i\}} (R_k), \qquad \Gamma_i = \sum_{\tau_k \in \tau \setminus \{\tau_i\}} \gamma_k, \qquad (11)$$

$$\Omega_i = \sum_{\tau_k \in \tau \setminus \{\tau_i\}} \omega_k.$$

$\text{npdbf}_i(t, \Delta)$ can be upper bounded by a linear function $g(\Delta) = \Omega_i \Delta + \Omega_i R^i_{\max} + 2\Gamma_i$. If $\Omega_i < 1$, then $\Delta' = (\Omega_i R^i_{\max} + 2\Gamma_i)/(1 - \Omega_i)$.

Because $\Delta' \geq |I_i|$ if $\Omega_i < 1$ and it is sufficient that $B_i \geq |I_i|$, it is sufficient if $B_i = (\Omega_i R^i_{\max} + 2\Gamma_i)/(1 - \Omega_i)$. Thus, for the task system τ, if $\max_{\tau_i \in \tau}(\Omega_i) < 1$ and τ is feasible when C_i is inflated by B_i, then τ is SRT-schedulable under L-SAPA-EF with tardiness at most $3mT_{\max}/u_{\min}$.

D. Refining the Blocking Analysis

For simplicity, several over-approximations have been made in this section. For example, the tardiness bound of $3mT_{\max}/u_{\min}$ shown for preemptive SAPA-EF can be reduced to $2mT_{\max}/u_{\min}$ by specifying unique upper and lower bounds on $\chi_{i,j}(t)$ for each τ_i (as in [8]) and modifying the proof of Thm. 1 accordingly. This has the twofold effect of reducing R_k for each τ_k in (11), which reduces B_i, and reducing the tardiness bound under L-SAPA-EF. In the same vein, we made use of (10) as a tardiness bound instead of the tighter (9). This is because (9) depends on $U(\tau)$ while (10) does not. If tardiness depends on $U(\tau)$, then it also depends on the WCET inflation caused by B_i, while B_i in turn also depends on tardiness due to R_k in (11). This circular dependence would require that B_i be deduced via some iterative procedure. In this case, the condition required would no longer be $\Omega_i < 1$, but rather that the procedure converges. Such an iterative procedure would yield less pessimistic B_i terms. It is also not necessary to consider $R^i_{\max}$ instead of each individual R_k.

[9]Assuming that tardiness is bounded may seem like circular reasoning. In the formal version of this analysis, the value of B_i is proven through induction on the jobs of τ. Part of the induction hypothesis is that prior jobs have bounded tardiness, which allows us to make this assumption about R_k.

A more substantial over-approximation is our assumption that all tasks in τ besides τ_i contribute to $\text{npdbf}_i(t, \Delta)$, causing the sums in (11) to be over all such tasks. Fixed tasks do not need to be considered in these sums, as they do not cause NP blocking. This is intuitive, as for them, progression, which does not cause capacity loss, is equivalent to NP execution. In practice, many tasks would likely be fixed as this maximizes cache locality for most tasks. Even if this is not the case, it was proven in [12] that for any feasible task system, it is possible to remove processors from tasks' affinity masks until at most $m - 1$ tasks are not fixed without affecting feasibility.

VI. Conclusion

We have demonstrated fundamental faults in SD regarding bounded tardiness under APA scheduling. The natural alternative is SAPA-EDF, which has attractive theoretical properties, but is hindered by complexities in its implementation and analysis. We have addressed complexities introduced by non-preemptivity and proposed a weaker form of it, progression. In practice, true NP sections (which cause capacity loss) would likely be used only when such sections are short (as in an OS kernel), with progression (which causes no loss) sufficing in other cases. As a side effect of obtaining these results, we greatly expanded the space of SRT-optimal APA schedulers.

Several other issues with SAPA-EDF must be resolved before it can be made into a practical EDF scheduler under APA. From an implementation point of view, we must consider how to balance scheduler efficiency with correctness, else we risk introducing scheduling glitches such as those possible under SD. From an analytical point of view, it remains an open problem how to account for suspensions, as well as how to implement suspension-based locking protocols under APA.

While we have demonstrated that SD may be unsuitable for APA, we believe that small modifications to SD are sufficient to yield SRT-optimality under special cases of processor affinities beyond just global or clustered scheduling. Given that SD is not SRT-optimal, identifying these special cases and the corresponding required modifications is likely the most complete answer we can give for the problem of how to provide real-time guarantees for SD with affinity masks.

REFERENCES

[1] Deadline task scheduling. https://github.com/torvalds/linux/blob/master/Documentation/scheduler/sched-deadline.rst. Online; accessed 03 June 2020.

[2] V. Bonifaci, B. B. Brandenburg, G. DAngelo, and A. Marchetti-Spaccamela. Multiprocessor real-time scheduling with hierarchical processor affinities. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 237–247, July 2016.

[3] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, Chapel Hill, NC, USA, 2011. AAI3502550.

[4] B. B. Brandenburg and J. H. Anderson. A clarification of link-based global scheduling. Technical Report MPI-SWS-2014-007, November 2014.

[5] F. Cerqueira, A. Gujarati, and B. B. Brandenburg. Linux's processor affinity api, refined: Shifting real-time tasks towards higher schedulability. In *2014 IEEE Real-Time Systems Symposium*, pages 249–259, Dec 2014.

[6] U. M. C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 12 pp.–341, 2005.

[7] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[8] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 413–422, 2007.

[9] B. Park. Return the best satisfying affinity and dl in cpudl_find. https://lkml.org/lkml/2017/3/23/171, March 2017. Online; accessed 03 June 2020.

[10] S. Tang and J. H. Anderson. Towards practical multiprocessor EDF with affinities (full version). In *41st IEEE Real-Time Systems Symposium (RTSS 2020)*, 2020. Available at http://jamesanderson.web.unc.edu/papers/.

[11] S. Tang, S. Voronov, and J. H. Anderson. GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] S. Voronov and J. H. Anderson. An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 408–420, 2018.

[13] K. Yang and J. H. Anderson. On the soft real-time optimality of global edf on uniform multiprocessors. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 319–330, Dec 2017.

[14] P. Zijlstra. An update on real-time scheduling on linux. https://www.ecrts.org/archives/index7520.html?id=284, June 2017. Online; accessed 09 June 2020.