Generalized Tardiness Bounds for Global Multiprocessor Scheduling*

Hennadiy Leontyev and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

We consider the issue of deadline tardiness under global multiprocessor scheduling algorithms. We present a general tardiness-bound derivation that is applicable to a wide variety of such algorithms (including some whose tardiness behavior has not been analyzed before). Our derivation is very general: job priorities may change rather arbitrarily at runtime, arbitrary non-preemptive regions are allowed, and capacity restrictions may exist on certain processors. Our results show that, with the exception of static-priority algorithms, most global algorithms considered previously have bounded tardiness. In addition, our results provide a simple means for checking whether tardiness is bounded under newly-developed algorithms.

1 Introduction

There is growing interest in using multicore platforms, which are becoming increasingly ubiquitous, in settings where soft real-time constraints are required. For example, one envisioned use of such platforms is as *multi-purpose home appliances*, with one machine providing many functions, including soft real-time applications like HDTV streaming and interactive video games. In *soft* real-time applications, deadline constraints exist, but deadline misses are sometimes tolerable.

To support such applications on a multicore platform, an appropriate multiprocessor scheduling algorithm must be used. This paper is directed at issues concerning such algorithms. Our specific focus is multiprocessor algorithms for scheduling soft real-time workloads specified as sporadic tasks (see Sec. 2). In devising such algorithms, two basic approaches exist: partitioning and global scheduling. Under partitioning, tasks are statically assigned to processors, and each processor schedules its assigned tasks using a uniprocessor scheduling algorithm. Under global scheduling, tasks are scheduled from a single run queue and may migrate among processors. For soft real-time systems, global algorithms have the advantage of being able to ensure bounded deadline tardiness, as long as the available processing capacity is not exceeded (something we assume throughout this paper). Bounded tardiness is a sufficient property in many soft real-time applications (provided the bounds are not too large). In particular, such bounds ensure that the long-term processor share of each task is in accordance with its specified utilization. Due to bin-packing limitations, such share guarantees are not possible under partitioning approaches, unless the overall system utilization is restricted. The main focus of this paper is algorithms that are capable of ensuring bounded tardiness (without restrictions on overall utilization).

Motivation and prior work. The first tardiness bounds to be established for a global scheduling algorithm pertained to the earliest-pseudo-deadline-first (EPDF) Pfair algorithm [5, 13]. This analysis was later extended to establish tardiness bounds for several variants of the global earliest-deadline-first (EDF) algorithm, wherein jobs with earlier deadlines have higher priority. These include preemptive and non-preemptive EDF [6] and two variants that slightly alter EDF prioritizations and allow a small number of special tasks to be guaranteed lower tardiness [7] or cause temporary overloads [9]. (The latter variant arises in an approach for scheduling multi-speed multiprocessor systems.) Tardiness bounds have also been established for the global first-in first-out (FIFO) algorithm [10], wherein jobs with earlier release times have higher priority. Given that tardiness is bounded under such disparate algorithms, several questions come to mind. Do other widely-studied global algorithms have bounded tardiness? Is there a singular characteristic of such algorithms that results in bounded tardiness? Can the class of algorithms for which tardiness is bounded be generally characterized?

Contributions. In this paper, we present a generalized tardiness result that answers these questions. This result implies that the singular characteristic needed for tardiness to be bounded is that a pending job's priority eventually (in bounded time) is higher than that of any future job. Global algorithms that do *not* have this characteristic (and for which tardiness can be unbounded) include static-priority algorithms such as the rate-monotonic (RM) algorithm, and "non-fair" dynamic-priority algorithms such as the earliest-deadline-*last* (EDL) algorithm, wherein jobs with earlier deadlines have *lower* priority. Global algorithms that *do* have this property include the EDF, FIFO, EDF-until-zero-laxity (EDZL), and least-laxity-first (LLF) algorithms. (EDZL and LLF are described later.)

We establish a generalized tardiness result by considering

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425.

a generic scheduling algorithm where job priorities are defined by points in time that may vary as time progresses. All of the algorithms mentioned above can be seen as special cases of this generic algorithm in which priorities are further constrained. Even the PD^2 Pfair algorithm [1], which uses a rather complex notion of priority, is a special case. The main result of this paper is a derivation of a tardiness bound that applies if priorities are *window-constrained*: a job's priority at any time must correspond to a point in time lying within a certain time window that contains its release and deadline. We also show that if this window constraint is violated, then tardiness can be unbounded. It is possible to define window-constrained prioritizations for EDF, FIFO, EDZL, LLF, EPDF, and PD^2, so these algorithms have bounded tardiness. (For EDF, FIFO, EPDF, and PD^2, this was previously known.) For any other algorithm that may be devised in the future, our results enable tardiness bounds to be established by simply showing that prioritizations can be expressed in a window-constrained way (instead of laboriously devising a new proof).

The notion of priority used in our generic algorithm is very general. For example, it allows arbitrary non-preemptive regions within a job to be specified, as well as combinations of different prioritizations, *e.g.*, using a combination of EDF and FIFO in the same system. Priority rules can even change dynamically (subject to the window constraint). For example, if a task has missed too many deadlines, then its job priorities can be boosted for some time so that it receives special treatment. Or, if a single job is in danger of being tardy, then its prioritization may be changed so that it completes execution non-preemptively. Tardiness also remains bounded if early-release behavior is allowed (see Sec. 4.3) or if the capacity of each processor that is available to the (soft) real-time workload is restricted. In simplest terms, the main message of this paper is that, *for global scheduling algorithms, bounded tardiness is the common case, rather than the exception* (at least, ignoring clearly impractical algorithms such as EDL). For the widely-studied EDZL and LLF algorithms, and for several of the variants of existing algorithms discussed above, this paper is the first to show that tardiness is bounded.

The rest of this paper is organized as follows. In Secs. 2–3, we present our task model and scheduling framework. Then, in Sec. 4, we present the tardiness proof that is the main result of this paper. As discussed later, tardiness may be different under different scheduling algorithms. In Sec. 5, we present results from experiments conducted to assess such differences. We conclude the paper in Sec. 6.

2. System Model

We consider the problem of scheduling on m processors a set τ of n sporadic tasks, $T_1, \ldots, T_n$. Each task is invoked or *released* repeatedly, with each such invocation called a *job*. Associated with each task T_i are two parameters, e_i and p_i: e_i gives the maximum *execution time* of one job of T_i, while, p_i, called the *period* of T_i, gives the minimum time between consecutive job releases of T_i. For simplicity, we assume that each job of T_i has an execution time of exactly e_i; this assumption can be removed and e_i treated as an upper bound at the expense of additional notation. For brevity, we often use the notation $T_i = (e_i, p_i)$ to specify task parameters. The *utilization of task* T_i is defined as $u_i = e_i/p_i$, and the *utilization of the task system* τ as $U_{sum} = \sum_{T_i \in \tau} u_i$.

The j^{th} job of T_i, where $j \geq 1$, is denoted $T_{i,j}$. A task's first job may be released at any time $t \geq 0$. The release time of job $T_{i,j}$ is denoted $r_{i,j}$ and its (absolute) deadline $d_{i,j}$ is defined as $r_{i,j} + p_i$. If $T_{i,j}$ completes at time t, then its *tardiness* is $max(0, t - d_{i,j})$. A *task*'s tardiness is the maximum of the tardiness of any of its jobs. We assume

$$U_{sum} \leq m. \tag{1}$$

Otherwise, tardiness may grow unboundedly. When a job of a task misses its deadline, the release time of the next job of that task is not altered. However, at most one job of a task may execute at any time, even if deadlines are missed.

We assume that released jobs are placed into a single global ready queue. When choosing a new job to schedule, the scheduler selects (and dequeues) the ready job of highest priority. As reiterated in Def. 4 in Sec. 4, a job is *ready* if it has been released and its predecessor (if any) has completed execution. Priorities are determined as follows.

Definition 1. (prioritization functions) Associated with each released job $T_{i,j}$ is a function of time $\chi_{i,j}(t)$, called its *prioritization function*. If $\chi_{i,j}(t) < \chi_{k,h}(t)$, then the priority of $T_{i,j}$ is higher than the priority of $T_{k,h}$ at time t.

We assume that, when comparing priorities, any ties are broken arbitrarily but consistently.

3. Example Mappings

We now show how to describe several well-known scheduling policies in our framework, using the two-processor task set $\tau = \{T_1 = (1,3), T_2 = (2,3), T_3 = (1,4), T_4 = (3,4)\}$ as an example. In depicting example schedules, we use up (down) arrows to depict job releases (deadlines).

Example 1. Fig. 1 shows a schedule for τ under the preemptive EDF algorithm. In this case, since jobs are prioritized by deadline, it suffices to define $\chi_{i,j}(t) = d_{i,j}$ for each $T_{i,j}$. In Fig. 1, the value of $\chi_{i,j}(t)$ is shown for each job $T_{i,j}$ using a black circle.

Example 2. Fig. 2 shows a schedule for τ under the preemptive global RM algorithm. In this case, $T_{i,j}$ should have priority over $T_{k,h}$ if $i < k$ (since the tasks in τ are

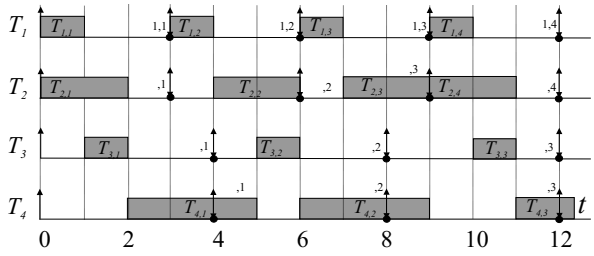

Figure 1. Example 1 (preemptive global EDF).

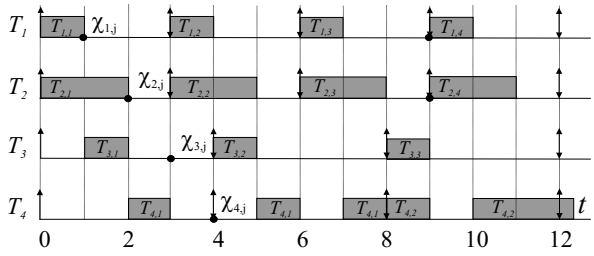

Figure 2. Example 2 (preemptive global RM).

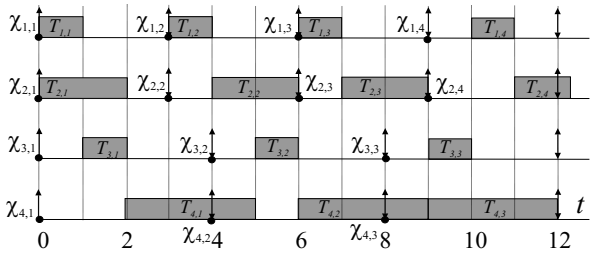

Figure 3. Example 3 (global FIFO).

ordered by increasing periods). Thus, we can simply define $\chi_{i,j}(t) = i$ for each job $T_{i,j}$, as shown.

Example 3. Fig. 3 shows a schedule for τ under the global FIFO algorithm (which, by definition, schedules jobs non-preemptively). In this case, since jobs are prioritized by release times, it suffices to define $\chi_{i,j}(t) = r_{i,j}$ for each job $T_{i,j}$, as shown.

Example 4. We now consider a slightly more complicated example, namely the preemptive LLF global scheduling algorithm [11]. The *laxity* or *slack* of a job $T_{i,j}$ at time t is defined as

$$slack_{i,j}(t) = d_{i,j} - t - (e_i - \delta_{i,j}(t)), \qquad (2)$$

where $\delta_{i,j}(t)$ is the amount of time for which $T_{i,j}$ has executed before t. If a job does not miss its deadline, then its slack is always non-negative; if it does miss its deadline, then its slack becomes negative at some time prior to its deadline. According to LLF, $T_{i,j}$ has higher priority than $T_{k,h}$ at time t if $slack_{i,j}(t) < slack_{k,h}(t)$. To capture this,

Table 1. χ-values in Example 4.

Time t	$\chi_{1,j}(t)$	$\chi_{2,j}(t)$	$\chi_{3,j}(t)$	$\chi_{4,j}(t)$
0	2	1	3	1
1	2	2	3	2
2	2	–	3	3
3	5	4	3	–
4	5	5	7	5
5	5	–	7	6
6	8	7	7	7
7	8	8	7	–
8	8	–	11	9
9	11	10	11	10
10	11	11	11	11
11	11	–	11	–

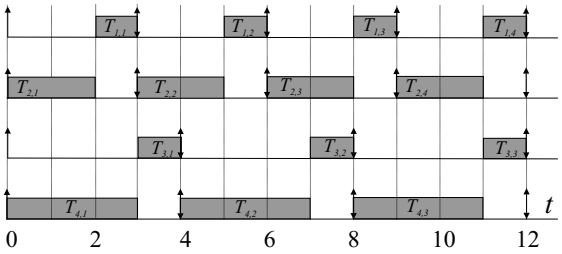

Figure 4. Example 4 (global preemptive LLF).

we can simply define $\chi_{i,j}(t) = d_{i,j} - (e_i - \delta_{i,j}(t))$ for each job $T_{i,j}$. Because this definition depends on $\delta_{i,j}(t)$, $\chi_{i,j}(t)$ is not constant, as in the prior examples, but is time-dependent. Assuming that it is updated only at integral points in time, $\chi_{i,j}(t+1) := \chi_{i,j}(t)+1$, if $T_{i,j}$ executes during the interval $[t, t+1)$, and $\chi_{i,j}(t+1) := \chi_{i,j}(t)$, otherwise.

Fig. 4 shows an LLF schedule for τ where ties are broken in favor of jobs currently executing. Because χ-values change with time, they are not shown in the schedule, as earlier, but are depicted separately in Table 1. The table shows the value of $\chi_{i,j}(t)$ for the earliest pending job $T_{i,j}$ of each task T_i where $0 \leq t \leq 11$.

Example 5. Interestingly, the definition of $\chi_{i,j}(t)$ is flexible enough to allow *combinations* of scheduling policies to be specified. For example, we can prioritize the jobs of $T_1, \ldots, T_3$ on an EDF basis and those of T_4 on a FIFO basis by defining $\chi_{i,j}(t) = d_{i,j}$ for $1 \leq i \leq 3$, and $\chi_{4,j}(t) = r_{4,j}$. A schedule for this hybrid policy is shown in Fig. 5.

Example 6. The EDZL algorithm [12], which is a hybrid of EDF and LLF, can be specified as well. In this case, $\chi_{i,j}(t)$ is set to $d_{i,j}$ (as in EDF) when $T_{i,j}$ is released, and is reset to $d_{i,j} - (e_i - \delta_{i,j}(t)) \leq d_{i,j}$ (as in LLF) when $T_{i,j}$'s slack becomes zero, where $\delta_{i,j}(t)$ is as defined earlier. To our knowledge, EDZL has not been considered previously in systems where deadlines can be missed.

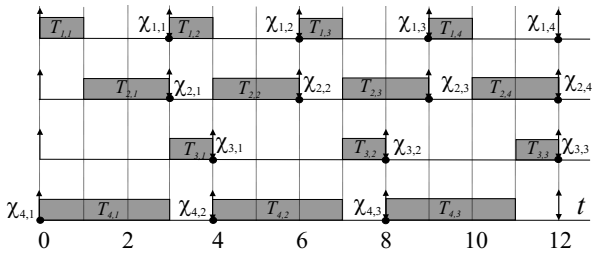

Figure 5. Example 5 (hybrid global scheduler).

However, if no deadlines are missed, then our definition yields priority comparisons that match exactly how EDZL has been specified in prior work. It is possible that other variants could be defined that prioritize jobs differently when deadlines are missed.

As noted earlier, the PD^2 Pfair algorithm [1] can also be modeled using our framework. PD^2 uses a rather complicated notion of priority, and correspondingly, requires more complex definitions than those used above. Due to space constraints, these definitions are omitted here.

4. Tardiness Bound

In this section, we show that any scheduling algorithm (specified according to Def. 1) has bounded tardiness if its prioritization functions are "window-constrained," as defined below in Def. 5. This definition imposes two separate constraints on χ-values. We show that if either is violated, then tardiness may become unbounded.

4.1. Definitions

The system start time is assumed to be zero. For any time $t > 0$, t^- denotes the time $t - \epsilon$ in the limit $\epsilon \to 0+$.

Definition 2. (active jobs) A task T_i is *active* at time t if there exists a job $T_{i,j}$ (called T_i's *active job* at t) such that $r_{i,j} \leq t < d_{i,j}$. By our task model, every task has at most one active job at any time.

Definition 3. (pending jobs) $T_{i,j}$ is *pending* at t in a schedule $\mathcal{S}$ if $r_{i,j} \leq t$ and $T_{i,j}$ has not completed execution by t in $\mathcal{S}$.

Definition 4. (ready jobs) A pending job $T_{i,j}$ is *ready* at t in a schedule $\mathcal{S}$ if $t \geq r_{i,j}$ and all prior jobs of T_i have completed execution by t in $\mathcal{S}$.

Definition 5. (window-constrained priorities) A scheduling algorithm's prioritization functions are *window-constrained* iff, for each task T_i, there exist constants $\phi_i \geq 0$ and $\psi_i \geq 0$ such that, for each job $T_{i,j}$ of T_i,

$$r_{i,j} - \phi_i \leq \chi_{i,j}(t) \leq d_{i,j} + \psi_i \qquad (3)$$

holds at each time t where $T_{i,j}$ is pending.

Note that (3) requires a job's χ-values to lie within a window that contains its release and deadline. This window may extend beyond the release and deadline by constant amounts. It is easy to see that, other than RM, all of the algorithms considered in Sec. 3 have prioritization functions that satisfy this requirement. In fact, for these algorithms, even non-preemptive execution can be allowed. (FIFO is non-preemptive, by definition.) Specifically, non-preemptivity can be modeled by setting $\chi_{i,j}(t)$ to be $r_{i,j} - C$ whenever the job $T_{i,j}$ enters a non-preemptive region, where C is a constant large enough to ensure that any unscheduled or newly-released job has lower priority.

In contrast, the prioritization functions specified for RM fail to be window-constrained because they violate the required lower bound: as new jobs of each task T_i are released, $\chi_{i,j}(t) < r_{i,j} - \phi_i$ will eventually hold for some job $T_{i,j}$ for any choice of the constant ϕ_i. It can be shown that the task system in Example 2 has unbounded tardiness. In particular, if each job is released as soon as possible, then the processing capacity available to T_4 every 12 time units is the same as is depicted in Fig. 2. This capacity is less than the amount of work generated by T_4 during the same interval. As a result, more and more work shifts to future intervals, causing tardiness for T_4 to grow unboundedly. (The fact that tardiness can be unbounded under RM was also established by Devi [4].)

It is possible to "fix" the prioritization functions for RM so that the required lower bounds are adhered to, but then the upper bounds will be violated. For example, we could simply define $\chi_{i,j}(t) = i + t'$, where t' is the time where the most recent job release occurred at or before t. This definition simply shifts the χ-values defined earlier to future points in time as new jobs are released. However, we know that tardiness for T_4 is unbounded, so eventually $\chi_{4,j}(t) > d_{4,j} + \psi_4$ will hold for some pending job $T_{4,j}$ of T_4 for any choice of the constant ψ_4. We summarize this discussion as follows. (Recall that any task set considered in this paper is assumed to satisfy (1).)

Theorem 1. *If either the lower or upper bound given in* (3) *is eliminated, then there exists a scheduling algorithm that satisfies the remaining condition for which tardiness is unbounded for some task set.*

Most of the rest of this paper is devoted to showing that any scheduling algorithm $\mathcal{A}$ with window-constrained prioritization functions has bounded tardiness. Before embarking on the proof, a few more definitions are in order.

A task system is *concrete* if the release times of all jobs are specified, and *non-concrete*, otherwise. The tardiness bound established for $\mathcal{A}$ is derived by comparing the allocations to a concrete task system τ in an ideal processor-sharing (PS) schedule to those in a schedule produced by

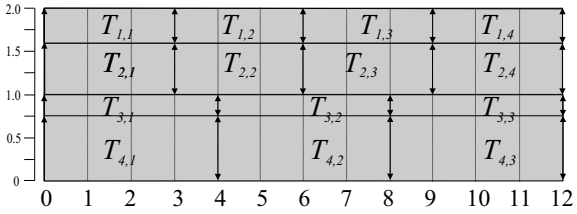

Figure 6. PS **schedule for τ in Example 1.**

$\mathcal{A}$. In a *PS schedule*, each job of a task T_i is executed at a constant rate of $u_i = \frac{e_i}{p_i}$ between its release and deadline. Fig. 6 depicts an example. Note that, in a PS schedule, each job completes exactly at its deadline. Thus, if a job misses its deadline, then it is "lagging behind" the PS schedule — this concept of "lag" is instrumental in the analysis and is formalized below.

Let $\mathsf{A}(T_{i,j}, t_1, t_2, \mathcal{S})$ denote the total allocation to the job $T_{i,j}$ in an arbitrary schedule $\mathcal{S}$ in $[t_1, t_2)$. Then, the difference between the allocations to a job $T_{i,j}$ up to time t in a PS schedule and an arbitrary schedule $\mathcal{S}$, termed the *lag of job $T_{i,j}$ at time t in schedule $\mathcal{S}$*, is given by

$$\mathsf{lag}(T_{i,j}, t, \mathcal{S}) = \mathsf{A}(T_{i,j}, 0, t, \mathsf{PS}) - \mathsf{A}(T_{i,j}, 0, t, \mathcal{S}). \quad (4)$$

Task lags can be similarly defined:

$$\mathsf{lag}(T_i, t, \mathcal{S}) = \sum_{j \geq 1} \mathsf{A}(T_{i,j}, 0, t, \mathsf{PS}) - \mathsf{A}(T_{i,j}, 0, t, \mathcal{S}). \quad (5)$$

Finally, the *lag for a finite job set Ψ at time t in the schedule $\mathcal{S}$* is defined by

$$\begin{aligned} &\mathsf{LAG}(\Psi, t, \mathcal{S}) \\ &= \sum_{T_{i,j} \in \Psi} \mathsf{lag}(T_{i,j}, t, \mathcal{S}) \\ &= \sum_{T_{i,j} \in \Psi} (\mathsf{A}(T_{i,j}, 0, t, \mathsf{PS}) - \mathsf{A}(T_{i,j}, 0, t, \mathcal{S})). \end{aligned} \quad (6)$$

Since $\mathsf{LAG}(\Psi, 0, \mathcal{S}) = 0$, the following holds for $t' \leq t$.

$$\begin{aligned} &\mathsf{LAG}(\Psi, t, \mathcal{S}) \\ &= \mathsf{LAG}(\Psi, t', \mathcal{S}) \\ &\quad + \mathsf{A}(\Psi, t', t, \mathsf{PS}) - \mathsf{A}(\Psi, t', t, \mathcal{S}) \end{aligned} \quad (7)$$

The concept of lag is important because, if lags remain bounded, then tardiness is bounded as well.

Definition 6. (busy/non-busy intervals) A time interval $[t_1, t_2)$ is *busy* for a job set Ψ if, at each time $t \in [t_1, t_2)$, all processors execute jobs from Ψ, and is *non-busy* for Ψ otherwise. An interval $[t_1, t_2)$ is *maximally non-busy* for Ψ if it is non-busy for Ψ at every instant within it and either $t_1 = 0$ or t_1^- is a busy instant for Ψ.

When using the above terminology, we will omit "for Ψ" if the job set under consideration is clear. According to the lemma below, the lag for a job set Ψ cannot increase across a busy interval for Ψ. This fact was proved in [6]. Its proof relies only on the fact that the interval in question is non-busy, and not on how jobs are scheduled, so it applies in our context as well.

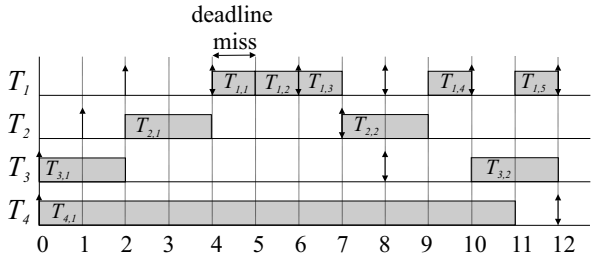

Figure 7. A schedule for τ in Example 7.

Lemma 1. *For any interval $[t_1, t_2)$ that is busy for Ψ,* $\mathsf{LAG}(\Psi, t_2, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t_1, \mathcal{S})$.

We are interested in non-busy intervals (for a job set) because total lag (for that job set) can increase unboundedly only across such (non-busy) intervals. Such increases can lead to deadline misses. The following example illustrates how lag can change across busy and non-busy intervals.

Example 7. Consider a two-processor system upon which a task set $\tau = \{T_1 = (1, 2), T_2 = (2, 6), T_3 = (2, 8), T_4 = (11, 12)\}$ is to be scheduled, where the first jobs of T_1, T_2, T_3, and T_4 are released at times 2, 1, 0, and 0 respectively. Assume that $\mathcal{A}$ is the FIFO algorithm, *i.e.*, jobs are prioritized using $\chi_{i,j}(t) = r_{i,j}$. Consider the schedule for τ in Fig. 7. Under $\mathcal{A}$, $T_{1,1}$ misses its deadline at time 4 by one time unit because it cannot preempt $T_{2,1}$ and $T_{4,1}$, which have earlier release times and later deadlines.

Let $\Psi = \{T_{1,1}, \ldots, T_{1,5}, T_{2,1}, T_{3,1}, T_{4,1}\}$ be the set of jobs with deadlines at most 12. The interval $[4, 7)$ in Fig. 7 is a busy interval for Ψ. By (7), $\mathsf{LAG}(\Psi, 7, \mathcal{S}) = \mathsf{LAG}(\Psi, 4, \mathcal{S}) + \mathsf{A}(\Psi, 4, 7, \mathsf{PS}) - \mathsf{A}(\Psi, 4, 7, \mathcal{S})$, where $\mathcal{S}$ is the schedule under $\mathcal{A}$. The allocation of Ψ in the PS schedule during the interval $[4, 7)$ is $\mathsf{A}(\Psi, 4, 7, \mathsf{PS}) = 3/2 + 6/6 + 6/8 + 33/12 = 6$. The allocation of Ψ in $\mathcal{S}$ throughout $[4, 7)$ is also 6. Thus, $\mathsf{LAG}(\Psi, 7, \mathcal{S}) = \mathsf{LAG}(\Psi, 4, \mathcal{S})$.

Now let $\Psi = \{T_{1,1}\}$ be the set of jobs with deadlines at most 4. Because the jobs $T_{2,1}$ and $T_{4,1}$, which have deadlines after time 4, execute within the interval $[2, 4)$ in Fig. 7, this interval is non-busy for Ψ in $\mathcal{S}$. By (6), $\mathsf{LAG}(\Psi, 4, \mathcal{S}) = \mathsf{A}(\Psi, 0, 4, \mathsf{PS}) - \mathsf{A}(\Psi, 0, 4, \mathcal{S})$. The allocation of Ψ in the PS schedule throughout the interval $[0, 4)$ is $\mathsf{A}(\Psi, 0, 4, \mathsf{PS}) = 2 \cdot 1/2 = 1$. The allocation of Ψ in $\mathcal{S}$ is $\mathsf{A}(\Psi, 0, 4, \mathcal{S}) = 0$. Thus, $\mathsf{LAG}(\Psi, 4, \mathcal{S}) = 1 - 0 = 1$. Fig. 7 shows that at time 4, $T_{1,1}$ from Ψ is pending. This job has unit execution cost, which is equal to the amount of pending work given by $\mathsf{LAG}(\Psi, 4, \mathcal{S})$.

4.2 Tardiness Bound for $\mathcal{A}$

Given an arbitrary non-concrete task system τ^N, we want to determine the maximum tardiness of any job of any task in any concrete instantiation of τ^N scheduled on m processors.

The approach for doing this is based on techniques from [6, 7]. Let τ be a concrete instantiation of τ^N. Let

$$\rho = max_{T_h \in \tau}(\phi_h) + max_{T_h \in \tau}(\psi_h). \tag{8}$$

Let $T_{\ell,j}$ be a job of a task T_ℓ in τ, let $t_d = d_{\ell,j}$, and let $\mathcal{S}$ be a schedule, produced for τ by the scheduling algorithm $\mathcal{A}$. We assume that the schedule $\mathcal{S}$ has the following property.

(P) The tardiness of every job of every task T_k in τ with deadline less than t_d is at most $x + e_k$, where $x \geq \rho$.

Our goal is to determine the smallest $x \geq \rho$ such that the tardiness of $T_{\ell,j}$ remains at most $x + e_\ell$. Such a result would by induction imply a tardiness of at most $x + e_k$ for all jobs of every task $T_k \in \tau$. Because τ is arbitrary, the tardiness bound will hold for every concrete instantiation of τ^N.

The objective is easily met if $T_{\ell,j}$ completes by its deadline, t_d, so assume otherwise. The completion time of $T_{\ell,j}$ then depends on the amount of work that can compete with $T_{\ell,j}$ after t_d. Hence, a value for x can be determined via the following steps.

1. Compute an upper bound on pending work for tasks in τ (including $T_{\ell,j}$) that can compete with $T_{\ell,j}$ after t_d.
2. Determine the amount of such work necessary for the tardiness of $T_{\ell,j}$ to exceed $x + e_\ell$.
3. Determine the smallest $x \geq \rho$ such that the tardiness of $T_{\ell,j}$ is at most $x + e_\ell$ using the upper bound in Step 1 and the necessary condition in Step 2.

To reason about the tardiness of $T_{\ell,j}$ we need to determine how other jobs delay its execution. We classify such jobs based on the relation between their prioritization functions and deadlines and those of $T_{\ell,j}$, as follows.

$$\begin{aligned}
\textbf{dH} &= \{T_{i,k} :: \exists a : \chi_{i,k}(a) \leq \chi_{\ell,j}(a) \wedge (d_{i,k} \leq t_d)\} \\
\textbf{dL} &= \{T_{i,k} :: \forall a : \chi_{i,k}(a) > \chi_{\ell,j}(a) \wedge (d_{i,k} \leq t_d)\} \\
\textbf{DH} &= \{T_{i,k} :: \exists a : \chi_{i,k}(a) \leq \chi_{\ell,j}(a) \wedge i \neq \ell \\
&\qquad\qquad \wedge (d_{i,k} > t_d)\} \\
\textbf{DL} &= \{T_{i,k} :: \forall a : \chi_{i,k}(a) > \chi_{\ell,j}(a) \wedge (d_{i,k} > t_d)\}
\end{aligned}$$

In this notation, **d** and **D** denote, respectively, deadlines at most and greater than t_d. Also, **H** denotes that $T_{i,k}$'s priority is higher or equal to that of $T_{\ell,j}$ and **L** denotes that $T_{i,k}$'s priority is lower than that of $T_{\ell,j}$. Note that $T_{\ell,j} \in$ **dH**.

Example 8. Consider the task set $\tau = \{T_1 = (1, 2), T_2 = (4, 7), T_3 = (8, 9)\}$ and the PS schedule for it in Fig. 8.

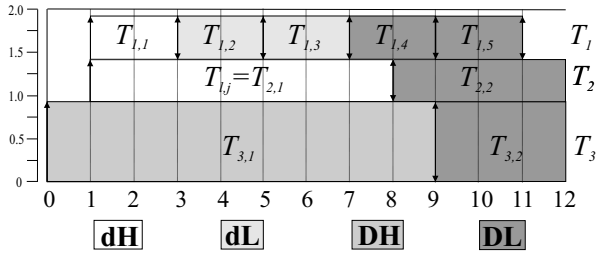

Figure 8. Job set partitioning.

Jobs $T_{1,1}$ and $T_{2,1}$ are released at time 1, and job $T_{3,1}$ is released at time 0. Consider the job $T_{\ell,j} = T_{2,1}$, which has a deadline at time 8. As in Example 7, assume that $\mathcal{A}$ is the FIFO algorithm, *i.e.*, jobs are prioritized using $\chi_{i,j}(t) = r_{i,j}$. With respect to $T_{2,1}$, the four sets mentioned above are **dH** $= \{T_{1,1}, T_{2,1}\}$, **dL** $= \{T_{1,2}, T_{1,3}\}$, **DH** $= \{T_{3,1}\}$, and **DL** $= \{T_{1,4}, T_{1,5}, T_{2,2}, T_{3,2}\}$. (**DL** would also include any jobs released after time 11.)

The set of jobs with deadlines at most t_d is further referred to as $\Psi =$ **dH** $\cup$ **dL**. We are interested in this set of jobs because these jobs do not execute beyond t_d in the PS schedule. Because the jobs in **dH** $\cup$ **DH** might have the priority at least that of $T_{\ell,j}$ (at some time instant), the execution of $T_{\ell,j}$ might be postponed (in the worst case) until there are at most m ready jobs in Ψ including $T_{\ell,j}$.

Determining an upper bound on competing work. Because jobs in **dH** $\cup$ **DH** can have priority at least that of $T_{\ell,j}$, the competing work for $T_{\ell,j}$ beyond t_d, $W($**dH** $\cup$ **DH**$, t_d, \mathcal{S})$, is bounded from above by the sum of **(i)** the amount of work pending at t_d for jobs in **dH**, and **(ii)** the amount of work $\mathsf{D}($**DH**$, t_d, \mathcal{S})$ demanded by jobs in **DH** that can compete with $T_{\ell,j}$ after t_d.

For the pending work mentioned in (i), because jobs from Ψ have deadlines at most t_d, they do not execute in the PS schedule beyond t_d. Thus, the work pending for jobs in **dH** is given by $\mathsf{LAG}($**dH**$, t_d, \mathcal{S})$, which must be positive in order for $T_{\ell,j}$ to miss its deadline at t_d. We find it more convenient to reason about $\mathsf{LAG}(\Psi, t_d, \mathcal{S})$ instead. Note that $\mathsf{LAG}($**dL**$, t_d, \mathcal{S})$ is non-negative because the jobs in **dL** cannot perform more work by time t_d in $\mathcal{S}$ than they have performed in the PS schedule. Hence, $\mathsf{LAG}($**dH**$, t_d, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t_d, \mathcal{S})$, which implies that $W($**dH** $\cup$ **DH**$, t_d, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{D}($**DH**$, t_d, \mathcal{S})$. Thus, an upper bound on $W($**dH** $\cup$ **DH**$, t_d, \mathcal{S})$ can be obtained by determining bounds for $\mathsf{LAG}(\Psi, t_d, \mathcal{S})$ and $\mathsf{D}($**DH**$, t_d, \mathcal{S})$ individually.

Upper bound on $\mathsf{LAG}(\Psi, t_d, \mathcal{S})$**.** In deriving this bound, we assume that all busy and non-busy intervals considered are with respect to Ψ and the schedule $\mathcal{S}$ produced by the scheduling algorithm $\mathcal{A}$ unless stated otherwise.

To begin, note that, by Lemma 1, if no non-busy interval exists in $[0, t_d)$, then $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq \mathsf{LAG}(\Psi, 0, \mathcal{S}) = 0$. In that which follows, we consider the more interesting case wherein some non-busy interval exists in $[0, t_d)$. An interval could be non-busy for two reasons:

1. There are not enough ready jobs in Ψ to occupy all available processors, so it is immaterial whether jobs from **DH** or **DL** execute during the interval. We call such an interval *non-busy non-displacing*.

2. There are ready jobs in Ψ that cannot execute because, within certain sub-intervals, jobs in **DH** occupy one or more processors because they have higher priority. We call such an interval *non-busy displacing*.

Let the *carry-in* job $T_{k,j}$ of a task T_k be defined as the job, if any, for which $r_{k,j} \leq t_d < d_{k,j}$ holds. At most one such job could exist for each task T_k. Only such jobs may prevent the execution of jobs in Ψ before time t_d and hence increase the LAG for Ψ.

Definition 7. Let τ_H be the set of tasks that have carry-in jobs in **DH**.

Definition 8. Let δ_k be the amount of work performed by a carry-in job $T_{k,j}$ in the schedule $\mathcal{S}$ by time t_d.

In much of the rest of the analysis, we focus on a time t_n defined as follows: if there exists a non-busy non-displacing interval before t_d, then t_n is the end of the latest such interval; otherwise, $t_n = 0$.

Lemmas 2 and 3, given next, were proved in [7, 10] and [8], respectively. These proofs rely only on Property (P), and for Lemma 2, the definition of t_n. In particular, the exact way in which jobs are scheduled does not arise. Thus, both apply in our context. Intuitively, Lemma 2 holds because when carry-in jobs execute prior to t_d, they deprive the jobs in Ψ of processor time. If a carry-in job executes prior to t_d in the actual schedule $\mathcal{S}$ for δ_i time, then while it is executing, it receives an allocation of $u_i \cdot \delta_i$ in the PS schedule. This means its lag changes by $u_i \cdot \delta_i - \delta_i$, which is a decrease. Such decreases translate into a corresponding increased lag for the jobs in Ψ. Lemma 3 holds because, by Property (P), each job with a deadline prior to t_d is tardy by at most $x + e_k$ time units. From this, it can be shown that the allocation difference for any task in comparison to the PS schedule at any point in $[0, t_d]$ is at most $u_k \cdot x + e_k$.

Lemma 2. $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t_n, \mathcal{S}) + \sum_{T_k \in \tau_H} \delta_k(1 - u_k)$.

Lemma 3. $\mathsf{lag}(T_k, t, \mathcal{S}) \leq x \cdot u_k + e_k$ *for any task* T_k *and* $t \in [0, t_d]$.

Lemma 4. *Let* $U(\tau, y)$ $(E(\tau, y))$ *be the set of at most* y *tasks from* τ *of highest utilization* (*execution cost*)*, and let*

$$E_L = \sum_{T_i \in E(\tau, m-1)} e_i \text{ and } U_L = \sum_{T_i \in U(\tau, m-1)} u_i. \quad (9)$$

Then, $\mathsf{LAG}(\Psi, t_n, \mathcal{S}) \leq E_L + x \cdot U_L$.

Proof. If $t_n = 0$, then $\mathsf{LAG}(\Psi, t_n, \mathcal{S}) = 0$ and the lemma holds trivially, so assume that $t_n > 0$. Consider the set of tasks $\alpha = \{T_i : \exists T_{i,j} \in \Psi \text{ such that } T_{i,j} \text{ is pending at } t_n^-\}$. Because the instant t_n^- is non-busy non-displacing, $|\alpha| \leq m - 1$. If a task has no pending jobs at t_n^-, then $\mathsf{lag}(T_i, t_n, \mathcal{S}) \leq 0$. Thus, by (6) and Lemma 3, we have

$$\begin{aligned} &\mathsf{LAG}(\Psi, t_n, \mathcal{S}) \\ &= \sum_{T_i:\, T_{i,j} \in \Psi} \mathsf{lag}(T_i, t_n, \mathcal{S}) \leq \sum_{T_i \in \alpha} \mathsf{lag}(T_i, t_n, \mathcal{S}) \\ &\leq \sum_{T_i \in \alpha} x \cdot u_i + e_i \leq E_L + x \cdot U_L. \end{aligned}$$ □

From Lemmas 2 and 4, we have the desired upper bound, $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq E_L + x \cdot U_L + \sum_{T_k \in \tau_H} \delta_k(1 - u_k)$.

Upper bound on D**.** To compute a bound on the demand of jobs that can compete with $T_{\ell,j}$ after t_d, $\mathsf{D}(\mathbf{DH}, t_d, \mathcal{S})$, we first find the latest release time of such a job.

Lemma 5. *If* $T_{k,h} \in \mathbf{DH} \cup \mathbf{dH}$*, then* $r_{k,h} \leq d_{\ell,j} + \psi_\ell + \phi_k$.

Proof. Given that $r_{i,j} - \phi_i \leq \chi_{i,j}(t) \leq d_{i,j} + \psi_i$ is assumed to hold for any job $T_{i,j}$, if $T_{k,h} \in \mathbf{DH} \cup \mathbf{dH}$, then for some time t, $r_{k,h} - \phi_k \leq \chi_{k,h}(t) \leq \chi_{i,j}(t) \leq d_{\ell,j} + \psi_\ell$ holds. This implies $r_{k,h} \leq d_{\ell,j} + \psi_\ell + \phi_k$. □

Corollary 1. *All jobs in* $\mathbf{DH} \cup \mathbf{dH}$ *are released at or before* $t_d + \rho$*, where* $\rho = max_{T_h \in \tau}(\psi_h) + max_{T_h \in \tau}(\phi_h)$.

The following lemma bounds the amount of work due to jobs in **DH** that may delay $T_{\ell,j}$ after t_d.

Lemma 6. $\mathsf{D}(\mathbf{DH}, t_d, \mathcal{S}) \leq \sum_{T_k \in \tau_H}(e_k - \delta_k) + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil \right) \cdot e_k$.

Proof. Each job $T_{k,h}$ in **DH** is either a carry-in job or is released after t_d. In the latter case, by Lemma 5, $T_{k,h}$ is released in the interval $(t_d, t_d + \psi_\ell + \phi_k]$. Each task T_k may have one carry-in job in **DH** and up to $\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil$ jobs in **DH** released after t_d. If T_k has a carry-in job, then T_k is in τ_H and the work due to its carry-in job after t_d is at most $e_k - \delta_k$. The work generated by any job of T_k in **DH** released after t_d is at most e_k. From these facts, the lemma follows. (Note that T_ℓ is excluded from the second summation because it is does not have jobs in **DH**.) □

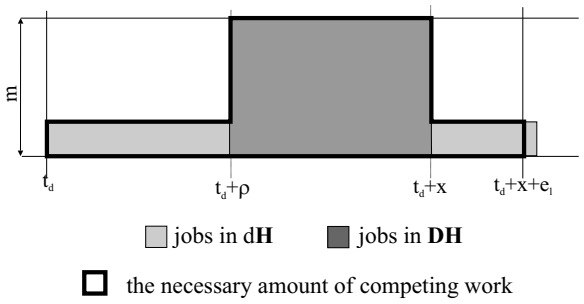

Figure 9. Illustration of Lemma 7.

Upper bound on $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S})$. Because $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S}) \leq \mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{D}(\mathbf{DH}, t_d, \mathcal{S})$, from Lemmas 2, 4, and 6 we have

$$\begin{aligned} & W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S}) \\ & \leq E_L + x \cdot U_L + \sum_{T_k \in \tau_H} (\delta_k \cdot (1 - u_k) + (e_k - \delta_k)) \\ & \quad + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil \right) \cdot e_k \\ & \leq E_L + x \cdot U_L + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k. \quad (10) \end{aligned}$$

Necessary condition for tardiness to exceed $x + e_\ell$. We now find a lower bound on the amount of competing work that is necessary for $T_{\ell,j}$ to miss its deadline by more than $x + e_\ell$. The following lemma, which is proved in an appendix, establishes the desired bound. Here, we give an informal explanation. The tardiness of $T_{\ell,j}$ can exceed $x + e_\ell$ only if competing jobs execute during the interval $[t_d, t_d + x + e_\ell)$ and the remaining processing capacity is not sufficient to accommodate the work pending for $T_{\ell,j}$ and any preceding jobs of T_ℓ. Fig. 9 illustrates the situation wherein the tardiness of $T_{\ell,j}$ barely exceeds $x + e_\ell$. This happens because jobs in **DH**, released at $t_d + \rho$ (refer to Corollary 1), occupy all the processors, thereby postponing the execution of $T_{\ell,j}$. The time instants where processors are allocated to jobs in $\mathbf{dH} \cup \mathbf{DH}$ are shaded. As seen, $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S})$ must exceed $\rho + m \cdot (x - \rho) + e_\ell$, which is outlined in bold.

Lemma 7. *If the tardiness of $T_{\ell,j}$ exceeds $x + e_\ell$, where $x \geq \rho$, then $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S}) > \rho + m \cdot (x - \rho) + e_\ell$.*

Deriving a tardiness bound. By Lemma 7, setting the upper bound on $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S})$ as implied by (10) to be at most $\rho + e_\ell + m \cdot (x - \rho)$ will ensure that the tardiness of $T_{\ell,j}$ is at most $x + e_\ell$. The resulting inequality is as follows.

$$\begin{aligned} E_L & + x \cdot U_L + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k \\ & \leq m \cdot (x - \rho) + e_\ell + \rho \end{aligned}$$

Solving (11) for x, we have

$$x \geq \frac{E_L + A(\ell)}{m - U_L}, \quad (11)$$

where

$$A(\ell) = (m-1) \cdot \rho - e_\ell + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k.$$

If x equals the greater of ρ and the right-hand side of (11) (recall that $x \geq \rho$ is required), then the tardiness of $T_{\ell,j}$ will not exceed $x + e_\ell$. A value for x that is independent of the parameters of T_ℓ can be obtained by replacing $A(\ell)$ by $max_{T_\ell \in \tau}(A(\ell))$ in the numerator of (11).

Theorem 2. *With x as defined above, the tardiness for a task T_k scheduled under the window-constrained algorithm $\mathcal{A}$ is at most $x + e_k$.*

Note that, for tardiness to be bounded under $\mathcal{A}$, the denominator in the right-hand side of (11) must be positive. This condition is satisfied because $U_L < m$ holds by (9).

4.3. Extensions to the Analysis

In this section, we briefly discuss several possible extensions to the analysis above.

Tightening the bound for specific algorithms. The bound in Theorem 2 can be improved for particular algorithms, by exploiting the structure of the set of tasks with carry-in jobs τ_H and the way jobs are prioritized. For example, for EDF, jobs with deadlines after t_d have lower priority than $T_{\ell,j}$. Thus, $\mathbf{DH} = \emptyset$, $\tau_H = \emptyset$, and $\mathsf{D}(\mathbf{DH}, t_d, \mathcal{S}) = 0$, which makes $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S})$ in (10) at most $E_L + x \cdot U_L$. As a result, tardiness under EDF is at most $\frac{E_L - min_{T_h \in \tau}(e_h)}{m - U_L} + e_k$.

Allowing early releases. Our analysis applies if jobs are allowed to be "early-released" [1], *i.e.*, to become available for execution before their actual release times. The idea of early releasing is illustrated in Fig. 10, which shows an EDF schedule for the task set in Example 1 in which early releases are allowed. Note that job $T_{1,2}$ begins its execution one time unit earlier than its actual release time. Note also that jobs $T_{4,1}$ and $T_{4,3}$ miss their deadlines by one time unit.

Restricting the available processing capacity. Several authors have established tardiness bounds in scenarios in which the full capacity of one or more processors is not available for the soft real-time workload [2, 7, 9]. Such capacity restrictions can be more generally dealt with in analyzing tardiness through the use of *service functions* [3]. Specifically, the capacity that Processor k can provide to the tasks in τ in any time interval of length $\Delta > 0$ can be characterized by a service function $\beta_k(\Delta) = max(0, \widehat{u_k} \cdot$

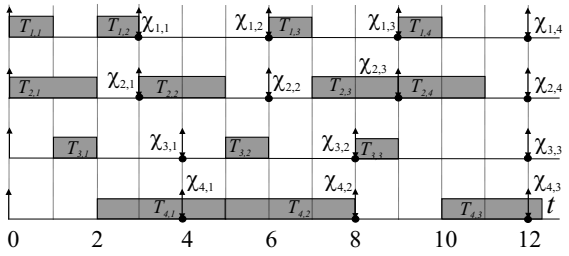

Figure 10. Early releasing under global EDF**.**

$(\Delta - \sigma_k))$, where $\widehat{u_k} \in (0, 1]$ and $\sigma_k \geq 0$. We require $\beta_k(\Delta)$ and σ_k to be specified for each k and further require $U_{sum} \leq \sum_{k=1}^{m} \widehat{u_k}$. Note that, if (unit-speed) Processor k is fully available to the tasks in τ, then $\beta_k(\Delta) = \Delta$.

Example 9. Consider a system with two processors that are not fully available for soft real-time tasks. The availability pattern, which repeats every eight time units, is shown in Fig. 11(a); intervals of unavailability are shown as shaded regions. For Processor 1, the minimum amount of service that is guaranteed to soft real-time tasks over any interval of length Δ is zero if $\Delta \leq 2$, $\Delta - 2$ if $2 \leq \Delta \leq 4$, and so on. Fig. 11(b) shows the minimum amount of time $\beta^*(\Delta)$ that is available on Processor 1 for soft real-time tasks over any interval $[t, t + \Delta]$. It also shows a service curve $\beta_1(\Delta) = max(0, \widehat{u_1}(\Delta - \sigma))$, where $\widehat{u_1} = \frac{5}{8}$ and $\sigma = 2$, which bounds $\beta^*(\Delta)$ from below. $\beta_1(\Delta)$ can be used to reflect the minimum service guarantee for soft real-time tasks on Processor 1.

Theorem 2 can be generalized as follows when service functions are used in this way.

Theorem 3. *If service functions are used as described above, then the tardiness of any task T_k under $\mathcal{A}$ is at most $x + e_k$, where*

$$x = \frac{E_L + max(A(\ell))}{\sum_{k=1}^{m} \widehat{u_k} - max(H - 1, 0) \cdot max(u_\ell) - U_L},$$

$$\begin{aligned} A(\ell) \quad = \quad & e_\ell \cdot (\sum_{k=1}^{m} (1 - \widehat{u_k}) - 1) + \sum_{k=1}^{m} \widehat{u_k} \cdot (\sigma + \sigma_k) \\ & + \sum_{T_k \in \tau \setminus T_\ell} \left(\left\lceil \frac{\psi_\ell + \phi_k}{p_k} \right\rceil + 1 \right) \cdot e_k \\ & + (m - 1 - max(H - 1, 0) \cdot u_\ell) \cdot \rho, \end{aligned}$$

$\sigma = max_{k \in [1..m]}(\sigma_k)$, and H is the number of processors with $\beta_k(\Delta) \neq \Delta$, provided the denominator of x is positive.

The new bound differs from that stated in Theorem 2 in several ways. First, the total processing capacity of the system in the denominator of (11) is changed from m to $\sum_{k=1}^{m} \widehat{u_k}$, which is the total guaranteed processing capacity of the system. Second, the term $\sum_{k=1}^{m} \widehat{u_k} \cdot (\sigma + \sigma_k)$ in $A(\ell)$ appears because one or more processors might not be available during an interval of length σ. This results in postponing pending work to the future and increasing tardiness. The other terms, specifically $e_\ell \cdot (\sum_{k=1}^{m} (1 - \widehat{u_k}) - 1)$ in $A(\ell)$ and the two expressions involving H, arise because of the fact that capacity is not provided to the tasks in τ at a steady rate. (Due to space constraints, it is not possible to give a more precise explanation of these terms.) For example, in Fig. 11(b), no service is guaranteed for intervals of length at most two. Note that, if all processors are fully available to the tasks in τ, then for each k, $\beta_k(\Delta) = \Delta$, $\widehat{u_k} = 1$, and $\sigma_k = \sigma = H = 0$. Thus, Theorem 2 becomes a special case of Theorem 3.

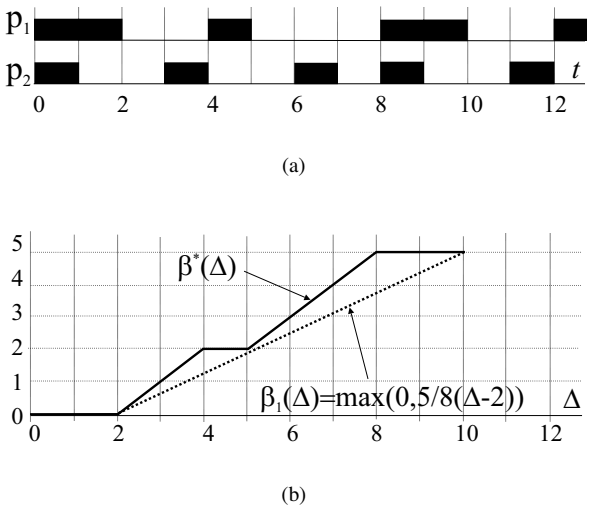

Figure 11. (a) Unavailable time instants and (b) service functions for Processor 1 (denoted P_1) in Example 9.

5. Experiments

As noted in Sec. 4.3, different algorithms to which Theorem 2 applies may exhibit very different behavior in terms of tardiness. To provide a sense of how significant such differences can be, we present here the results of some experiments that we conducted to compare observed tardiness under different scheduling algorithms. (The setup in these experiments is rather simple. We do not have sufficient space to present more exhaustive experiments.) In these experiments, task sets were generated at random. Task periods and utilizations were taken from {5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 28, 30, 32, 36, 40} and $(0, u_{max}]$, respectively, where u_{max} varied over

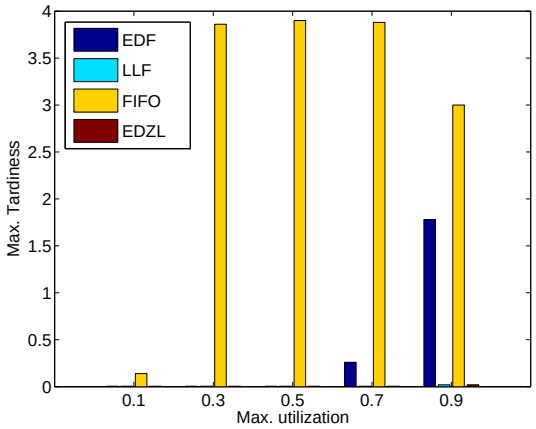

Figure 12. Maximum observed tardiness for different scheduling algorithms.

$\{0.1, 0.3, 0.5, 0.7, 0.9\}$. For each value of u_{max}, 50 task sets were generated for a four-processor system. Schedules were then produced for these task sets (with job releases occurring in a synchronous, periodic manner) for each of EDF, FIFO, LLF, and EDZL for 20,000 time units. In producing these schedules, system and scheduling overheads were taken to be negligible. For each schedule, the maximum observed tardiness was recorded. Fig. 12 shows the average of these values as a function of u_{max}. Note that tardiness under LLF and EDZL is smaller than that under FIFO and EDF (*much* smaller than FIFO). While LLF may be impractical in reality because it preempts jobs frequently, EDZL could be a viable approach for scheduling soft real-time workloads when tardiness is allowed.

6 Conclusion

We have presented a general tardiness-bound derivation that applies to a wide variety of global scheduling algorithms. This result shows that, with the exception of static-priority algorithms, most global algorithms of interest in the real-time-systems community have bounded tardiness. When considering new algorithms, the question of whether tardiness is bounded can be answered in the affirmative by simply showing that the required prioritization can be specified. Of course, a tardiness bound that is tighter than that given by our results might be possible through the use of reasoning specific to a particular algorithm. Indeed, it is difficult to obtain a very tight bound when assuming so little concerning the nature of the scheduling algorithm. Our goal in this paper was not to produce the tightest bound possible, but rather to produce a bound that could be widely applied.

Several interesting avenues for further work exist. For example, it would be interesting to determine if service functions could be used to deal with heterogeneous systems with processors of different speeds. It would also be interesting to investigate reactive techniques that can lessen tardiness for certain jobs, as circumstances warrant. Such techniques might exploit the fact that our framework allows priority definitions to be changed rather arbitrarily at runtime. Finally, our experimental results suggest that actual tardiness under EDZL is likely to be very low. It would be interesting to improve our analysis as it applies to EDZL in order to obtain a tight tardiness bound.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[2] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proc. of the 19th Euromicro Conf. on Real-Time Systems*, 2007. To appear.

[3] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[4] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[5] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF fair multiprocessor scheduling. In *Proc. of the 12th Int'l Workshop on Parallel and Distributed Real-Time Systems*, 2004.

[6] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symp.*, pages 330–341, 2005.

[7] U. Devi and J. Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *Proc. of the 20th IEEE Int'l Parallel and Distributed Processing Symp.*, 2006.

[8] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 75–84, 2006.

[9] H. Leontyev and J. Anderson. Tardiness bounds for EDF scheduling on multi-speed multicore platforms. 2007. In *Proc. of the 13th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2007. To appear.

[10] H. Leontyev and J. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *Proc. of the 19th Euromicro Conf. on Real-Time Systems*, 2007. To appear.

[11] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[12] X. Piao, S. Han, H. Kim, M. Park, Y. Cho, and S. Cho. Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. In *Proc. of the 9th IEEE Int'l Symp. on Object and Component-Oriented Real-Time Distributed Computing*, pages 359–364, 2006.

[13] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. *Journal of Embedded Computing*, 1(2):285–302.

Appendix: Proof of Lemma 7

The proof of Lemma 7 is based on similar proofs found in [4, 7, 10]. We prove it by proving the contrapositive: we assume that $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S}) \leq \rho + m(x-\rho) + e_\ell$ holds and show that the tardiness of $T_{\ell,j}$ cannot exceed $x + e_\ell$. In that which follows, the terms "busy" and "non-busy" are assumed to be interpreted with respect to the job set **dH** $\cup$ **DH**.

To begin, note that, by Property (P), if T_ℓ releases jobs before $T_{\ell,j}$, then the job $T_{\ell,j-1}$ completes by time t', where

$$t' \leq t_d - p_\ell + e_\ell + x \leq t_d + x. \tag{12}$$

Thus, if the latest busy instant after t_d is at or before $t_d + x$, then $T_{\ell,j}$ executes uninterruptibly after $t_d + x$ (if it has not completed by then) and completes by $t_d + x + e_\ell$, *i.e.*, its tardiness is at most $x + e_\ell$.

In the rest of the proof, we assume that the latest busy instant is after $t_d + x$. Let B_1 denote the total length of all busy intervals in $[t_d+x, t_d+x+e_\ell)$, as illustrated in Fig. 13. According to Corollary 1, all jobs in **dH** $\cup$ **DH** are released at or before $t_d + \rho$. Thus, the number of tasks with pending jobs in **dH**$\cup$**DH** can only decrease after $t_d+\rho$. If fewer than m processors are busy at any instant $t_n \in [t_d + \rho, t_d + x)$, then at most $m-1$ tasks with jobs in **dH**$\cup$**DH** have pending work at or after t_n. These tasks can be accommodated by at most $m - 1$ processors and execute without interruption at or after t_n. Because $T_{\ell,j}$ is in **dH** (see Sec. 4.2), by (12), this implies that it completes by time $t_d + x + e_\ell$, *i.e.*, its tardiness is at most $x + e_\ell$.

In the rest of the proof, we assume the following.

(B) m processors are busy throughout $[t_d + \rho, t_d + x)$.

Let B_2 denote the total length of all busy intervals during $[t_d, t_d + \rho)$, and let B denote the total length of all busy intervals in $[t_d, t_d + x + e_\ell)$, as illustrated in Fig. 13. Then, from Property (B), it follows that

$$B = B_1 + B_2 + (x - \rho). \tag{13}$$

Let W_1 denote the total length of all non-busy intervals in $[t_d + x, t_d + x + e_\ell)$, and let W_2 denote the total length of all non-busy intervals in $[t_d, t_d + \rho)$. Then, we have the following.

$$W_1 = e_\ell - B_1 \tag{14}$$

$$W_2 = \rho - B_2 \tag{15}$$

As noted above, $T_{\ell,j} \in$ **dH**. Also, any earlier jobs of T_ℓ that are pending at t_d are also in **dH**. Given this, we now show that the assumption that the tardiness of $T_{\ell,j}$ exceeds $x + e_\ell$ leads to a contradiction. Assuming this, the system performs work on a least one job in **dH** (namely, a job of T_ℓ) at every non-busy instant in $[t_d, t_d + x + e_\ell)$. Thus,

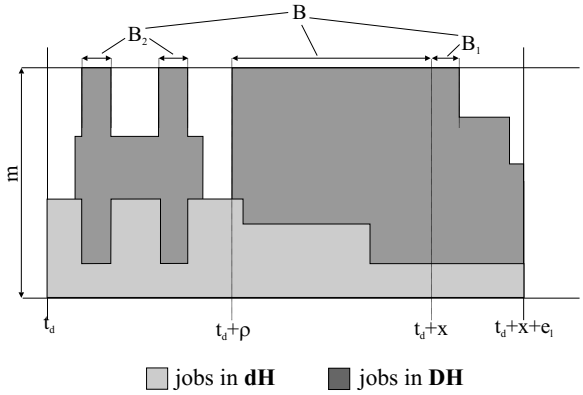

Figure 13. The structure of busy intervals in Lemma 7.

by the definition of B, the total work performed on jobs in **dH** $\cup$ **DH** in this interval is at least $Z = W_1 + W_2 + m \cdot B$. By (13)–(15),

$$\begin{aligned} Z = &\ W_1 + W_2 + m \cdot B \\ = &\ e_\ell - B_1 + \rho - B_2 + m \cdot (x - \rho + B_1 + B_2) \\ = &\ e_\ell + \rho + m \cdot (x - \rho) + (m-1) \cdot B_3, \end{aligned} \tag{16}$$

where $B_3 = B_1 + B_2 \geq 0$.

At the beginning of the proof, we assumed that $W(\mathbf{dH} \cup \mathbf{DH}, t_d, \mathcal{S}) \leq \rho + m \cdot (x - \rho) + e_\ell$. From this and (16), it follows that the amount of work pending at $t_d + e_\ell + x$ for jobs in **dH**$\cup$**DH** is at most $\rho + m \cdot (x-\rho) + e_\ell - e_\ell - \rho - m \cdot (x-\rho) - (m-1) \cdot B_3 = -(m-1) \cdot B_3 \leq 0$. From this, we conclude that the tardiness of $T_{\ell,j}$ does not exceed $x + e_\ell$, which contradicts our assumption to the contrary above. □