Scheduling Suspendable, Pipelined Tasks with Non-Preemptive Sections in Soft Real-Time Multiprocessor Systems[*]

Cong Liu and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

While most prior work on multiprocessor real-time scheduling focuses on independent tasks, dependencies due to non-preemptive sections, suspensions, and pipeline-based precedence constraints are common in practice. In this paper, such complexities are considered in the context of the global earliest-deadline-first scheduling algorithm. It is shown that any periodic task system with such dependencies can be transformed into one with only suspensions in a way that preserves maximum per-task response times. This result enables analysis directed at systems with suspensions to be applied if non-preemptive sections and/or pipelines are present as well.

1 Introduction

The growing prevalence of multicore platforms has led to much recent work on multiprocessor real-time scheduling. Most of this work has been directed at scheduling problems that arise when systems of independent tasks are to be supported. In practice, however, programming methodologies are often used that result in dependencies among tasks. As multiprocessor platforms become more ubiquitous, and real-time applications of greater complexity are supported on them, it is crucial that scheduling-related research be extended so that such dependencies can be properly addressed. In this paper, we consider this issue in the context of periodic task systems in which task dependencies may exist due to non-preemptive sections, suspensions, and pipeline-based precedence constraints.

Any one of these kinds of dependencies can cause a task system to be difficult to analyze from a schedulability perspective. For instance, non-preemptive sections may cause scheduling anomalies (e.g., shortening a job's execution time may actually increase some job's response time) and suspensions may cause unbounded job response times even in lightly-loaded systems [9]. Still, situations may exist in which all three kinds of dependencies are present. Consider, for example, a pipelined real-time computation where some tasks may require disk accesses and non-preemptivity arises due to system calls or critical sections. The timing correctness of such a system may be quite difficult to analyze, particularly if deadline misses cannot be tolerated. However, we show in this paper that the situation is not nearly so bleak, if bounded deadline tardiness is acceptable.

Bounded tardiness is a notion that has been studied extensively in the context of *global* scheduling algorithms, and such algorithms are our focus as well. In global algorithms, tasks are scheduled from a single run queue and may migrate across processors. Such algorithms stand in contrast to *partitioning* algorithms, which statically assign tasks to processors and use per-processor run queues. Under partitioning schemes, constraints on overall utilization are required to ensure timeliness even if bounded deadline tardiness can be tolerated. On the other hand, a variety of global-scheduling approaches are capable of ensuring bounded tardiness in ordinary periodic and sporadic systems (without suspending tasks or pipeline tasks) with no utilization loss, even if non-preemptive sections exist [2, 6]. In recent research, this work has been extended to show that, in fully-preemptive systems, bounded tardiness can be ensured if tasks either suspend [9] or have pipeline constraints [7, 8] (but not both), provided certain utilization restrictions hold.

In this paper, we consider whether these research results can be combined. That is, we address the problem of deriving conditions under which bounded tardiness can be ensured when *all* of the above-mentioned behaviors—non-preemptive sections, pipelines, and suspensions—are allowed. In considering this problem, we focus specifically on the *global earliest-deadline-first* (GEDF) algorithm, but our analysis could potentially be extended to apply to other global algorithms as well. Our main result is a transformation process that converts any implicit-deadline periodic task system with suspensions, pipelines, and non-preemptive sections into a simpler system with only suspensions. In the simpler system, each task's maximum job response time is at least that of the original system. This result allows tardiness bounds to be established by focusing only on the impacts of suspensions. While this result was motivated by our interest in tardiness bounds, the transformation we present is also relevant to hard real-time systems.

Related Work. To our knowledge, the problem addressed in this paper has not been considered before, in the context of either global or partitioned scheduling. However, some work has been done in which the effects of non-

[*]Work supported by AT&T, IBM, and Sun Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA9550-09-1-0549.

preemptive sections, pipelines, or suspensions have been considered independently, as noted above [2, 6–9]. Prior work on pipelining has also been done in the context of distributed systems (which must be scheduled by partitioning approaches) [3, 4, 13].

In work pertaining to uniprocessors (and by extension multiprocessors scheduled via partitioning), several sufficient schedulability tests have been presented for analyzing tasks with suspensions [1, 5, 10–12, 14]. Intractability/impossibility results pertaining to be suspension-oriented analysis have also been obtained [15, 16].

Contributions. In this paper, we consider periodic task systems with suspensions, pipelines, and non-preemptive sections; for conciseness, we henceforth refer to these systems as *NPS systems*. We show how to transform such a task system into a simpler periodic task system with only suspensions. In the transformed system, per-task maximum job response times are at least those of the original system. We show that this enables prior results on systems with suspensions [9] to be applied to derive tardiness bounds for more complex systems, as scheduled by GEDF. Such bounds are applicable to NPS task systems provided certain utilization constraints are met. We analyze the loss of system capacity inherent in these constraints via an extensive experimental study involving randomly-generated task systems. This study shows that the capacity loss is moderate to non-existent in scenarios where the executions costs of pipeline stages vary moderately and suspension and non-preemptive-section lengths are moderate.

Organization. The rest of this paper is organized as follows. Sec. 2 describes our system model. The transformation discussed above is obtained via a sequence of sub-transformations, which are described in Secs. 3 and 4. In Sec. 5, a tardiness bound for periodic NPS task systems is derived. The above-mentioned experimental evaluation is then presented in Sec. 6. Sec. 7 concludes.

2 NPS Task Systems

We consider the problem of scheduling a set $\tau^{NPS} = \{T_1, ..., T_n\}$ of n periodic pipeline tasks on $m \geq 2$ identical processors, where any such task may suspend and contain non-preemptive sections. Below, we present definitions pertaining to pipeline tasks. These definitions are illustrated with an example given later.

An h-stage pipeline task T_l, where $1 \leq h \leq m$, consists of h subtasks, $T_l^1, ...T_l^h$. Each subtask is released repeatedly, with each such invocation called a *job*. Jobs alternate between computation and suspension phases. We assume that each job of T_l^h executes for at most e_l^h time units (across all of its computation phases), suspends for at most s_l^h time units (across all of its suspension phases), and has at most c_l^h computation phases. We place no restrictions on how these phases interleave (a job can even begin or end with a suspension phase). We also do not restrict their lengths, other than upper-bounding them. Note, in particular, that 0-length computation phases are allowed in our model. The circumstances under which a 0-length computation phase can commence execution are just like with any other computation phase (as described later). However, a 0-length computation phase that commences execution at time t finishes execution at time t. Computation phases are also allowed to contain non-preemptive code segments. The maximum duration of any non-preemptive segment of T_l^h is denoted b_l^h, and the maximum such value taken over all subtasks in τ^{NPS} is denoted b_{max}.

The j^{th} job of T_l^h, denoted $T_{l,j}^h$, is released at time $r_{l,j}^h$ and has a deadline at time $d_{l,j}^h$. Associated with each pipeline task T_l is a period p_l, which specifies both the exact time between two consecutive job releases of T_l and the relative deadline of each such job, i.e., $d_{l,j}^h = r_{l,j}^h + p_l$. The release time of job $T_{l,j}^h$ is required to satisfy $r_{l,j}^h = (j - 1 + h - 1) \cdot p_l$. The *utilization of subtask* T_l^h is defined as $u_l^h = e_l^h / p_l$, and the utilization of T_l is defined as $u_l = \sum_{i=1}^{h} u_l^i$. The utilization of the task system τ^{NPS} is defined as $U_{sum} = \sum_{T_l \in \tau^{NPS}} u_l$. Notice that, if $h = 1$, and for the subtask T_l^h, $s_l^h = 0$ holds, then T_l^h is an ordinary (non-suspending) periodic task.

Successive jobs of the same subtask are required to execute in sequence. Also, for $h > 1$, job $T_{l,j}^h$ cannot commence execution (no matter whether its first phase is a computation phase or a suspension phase) until the last phase of the prior-stage job $T_{l,j}^{h-1}$ (be it a computation phase or suspension phase) completes. (That is, the j^{th} per-stage jobs must execute in sequence.) To avoid confusion when discussing these precedence constraints, we will refer to $T_{l,j-1}^h$ as the *L-predecessor* of $T_{l,j}^h$ (assuming $j > 1$), and $T_{l,j}^{h-1}$ as the *U-predecessor* of $T_{l,j}^h$ (assuming $h > 1$). ("L" and "U" stand for "left" and "upper", respectively).

The timeliness constraint considered in this paper is that deadline tardiness be bounded. If a job $T_{i,j}^k$ completes at time t, then its *response time* is defined as $t - r_{i,j}^k$ and its *tardiness* is defined as $max(0, t - d_{i,j}^k)$. A pipeline task's tardiness is the maximum of the tardiness of any job of any of its subtasks. Note that, when a job of a subtask misses its deadline, the release time of the next job of that subtask is not altered. Despite this, it is still required that a job cannot execute in parallel with either of its predecessors.

The results of this paper pertain to the GEDF scheduling algorithm. Under GEDF, released jobs are prioritized by their deadlines. Jobs in τ^{NPS} are ordered based on their priorities: $T_{i,v}^w \prec T_{a,b}^c$ if and only if $d_{i,v}^w < d_{a,b}^c$ or $(d_{i,v}^w = d_{a,b}^c) \wedge (i = a) \wedge (w < c)$ or $(d_{i,v}^w = d_{a,b}^c) \wedge (i < a)$. Thus, when comparing equal deadlines, the tie is broken in favor of earlier stages of the same pipeline task, and any

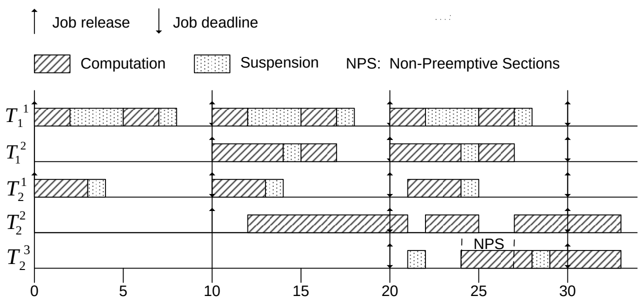


Figure 1: Example NPS task system.

remaining tie is broken by task ID. $T_{i,v}^w$ has higher priority than $T_{a,b}^c$ if and only if $T_{i,v}^w \prec T_{a,b}^c$. At any time, GEDF schedules the highest-priority enabled jobs (the term "enabled" is formally defined in Sec. 3) on the available processors subject to the constraint that a job in a non-preemptive section cannot be preempted. GEDF becomes fully preemptive when $b_{max} = 0$ and fully non-preemptive when $b_l^h = e_l^h$, for all l, h.

Example 1. Fig. 1 depicts an example GEDF schedule of a suspendable pipelined task system with non-preemptive sections, scheduled on three processors. This task system contains a two-stage pipeline task T_1 with a period of 10 time units and a three-stage pipeline task T_2 with a period of 10 time units. T_1^1 executes for 2 time units, then suspends for 3 time units, executes for another 2 time units, and finally suspends for 1 time unit. T_1^2 executes for 4 time units, then suspends for 1 time unit, and executes for another 2 time units. T_2^1 executes for 3 time units, and then suspends for 1 time unit. T_2^2 executes for 9 time units. T_2^3 first suspends for 1 time unit, then executes non-preemptively for 3 time units and then preemptively for another time unit, suspends for 1 time unit, and finally executes for 4 time units. As seen in the GEDF schedule, $T_{2,1}^2$ misses its deadline at time 20 by 1 time unit, which causes $T_{2,1}^3$ to start its first suspension phase at time 21. At time 24, $T_{2,1}^3$ starts executing its first computation phase since $T_{1,3}^1$, $T_{1,2}^2$, $T_{2,3}^1$ are suspended at that time. Since this execution is non-preemptive, at time 25, $T_{1,2}^2$ preempts $T_{2,1}^2$ instead of $T_{2,1}^3$.

Roadmap. In this paper, we show how to transform τ^{NPS}, a periodic NPS task system, into a periodic task system with only suspensions. This transformation requires two steps:

1. Transform τ^{NPS} into τ^{PS}, where τ^{PS} denotes a fully preemptive suspendable pipelined task system, by treating blocking times due to non-preemptive sections as suspensions. This is dealt with in Sec. 3.
2. Transform τ^{PS} into τ^{S}, where τ^{S} is a periodic task system with only suspensions (i.e., it contains no pipeline tasks and it is fully preemptive), by treating pipeline blocking times (see Sec. 4) as suspensions. This is dealt with in Sec. 4.

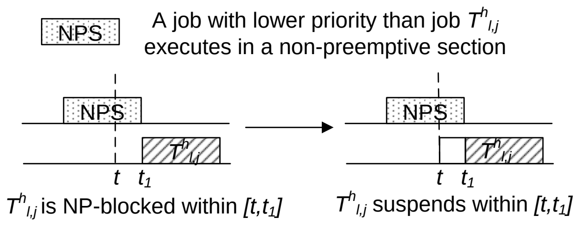

NPS
A job with lower priority than job $T^h_{l,j}$ executes in a non-preemptive section
NPS
$T^h_{l,j}$
t
t_1
$T^h_{l,j}$ is NP-blocked within [t,t₁]
NPS
$T^h_{l,j}$
t
t_1
$T^h_{l,j}$ suspends within [t,t₁]

Figure 2: Modeling NP-blocking as suspension.

3 Transforming τ^{NPS} to τ^{PS}

We transform τ^{NPS} into τ^{PS} by treating blocking times due to non-preemptive sections as suspensions.

Definition 1. We say that a task system τ is *concrete* if the actual execution cost and suspension time of every job of each task is fixed. For any τ (τ may be any of the task systems mentioned in the roadmap at the end of the prior section), we let $\overline{\tau}$ denote any arbitrary concrete instantiation of it.

Definition 2. A job $T_{l,j}^h$ is *enabled* if it has been released and both its U-predecessor (if any) and L-predecessor (if any) have completed. A job is considered to be *completed* if it has finished its last phase (be it suspension or computation).

A job $T_{l,j}^h$ is non-preemptively blocked, or *NP-blocked*, at time t if it is among m highest-priority enabled jobs according to GEDF, but it cannot execute because lower-priority jobs are executing non-preemptively at t. This can happen only when $T_{l,j}^h$ commences executing one of its computation phases. Given that any job $T_{l,j}^h$ has at most c_l^h such phases and the maximum length of any job's non-preemptive section is at most b_{max}, we have the following lemma.

Lemma 1. *Any job $T_{l,j}^h$ in τ^{NPS} can be NP-blocked for at most $c_l^h \cdot b_{max}$ time units.*

Given Lemma 1, we can define τ^{PS} by simply treating NP-blocking times as suspensions. That is, we view all non-preemptive sections as preemptive, and for any subtask T_l^h, we increase s_l^h by $c_l^h \cdot b_{max}$, which by Lemma 1 upper-bounds the NP-blocking time of T_l^h. This is illustrated in Fig. 2. The theorem below immediately follows.

Theorem 1. *For any concrete instantiation $\overline{\tau}^{NPS}$ of τ^{NPS}, there exists a concrete instantiation $\overline{\tau}^{PS}$ of τ^{PS} such that $\overline{\tau}^{NPS}$ and $\overline{\tau}^{PS}$ have equivalent GEDF schedules.*[1]

[1]That is, if S^{NPS} (S^{PS}) is the GEDF schedule for $\overline{\tau}^{NPS}$ ($\overline{\tau}^{PS}$), then job

Note that this transformation strongly exploits the fact that, in our task model, suspension phases are upper-bounded, and hence, can be reduced to reflect actual NP-blocking times. Note also that the "reverse" of this theorem may not hold: for a concrete instantiation $\overline{\tau}^{PS}$ of τ^{PS}, there may not exist a concrete instantiation $\overline{\tau}^{NPS}$ of τ^{NPS} such that $\overline{\tau}^{PS}$ and $\overline{\tau}^{NPS}$ have equivalent GEDF schedules.

Corollary 1. *For any subtask T_l^h, if the maximum response time of any job of T_l^h in any GEDF schedule for τ^{PS} is z time units, then the maximum response time of any such job in any GEDF schedule for τ^{NPS} is at most z time units.*

4 Transforming τ^{PS} to τ^S

In this section, we transform τ^{PS} into τ^S by treating pipeline blockings as suspensions. Notice that τ^S is a periodic task system where the first job of any task is released at time 0. However, according to our pipeline task model as described in Sec. 2, the first job of any non-first-stage subtask (T_l^h) (where $h \geq 2$) is released at time $(h-1) \cdot p_l$. However, we can easily add some "extra" initial jobs to each non-first-stage subtask so that each subtask starts at time 0. Specifically, for any subtask (T_l^h) (where $h \geq 2$), we add $h-1$ such "extra" jobs, $T_{l,2-h}^h, T_{l,2-h+1}^h, ..., T_{l,-1}^h, T_{l,0}^h$, with execution and suspension times of zero. Adding these jobs will not affect the schedule.

As shown in [7], the fundamental problem that makes pipeline scheduling difficult is *PL-blocking*, as defined below.

Definition 3. If a released job's L-predecessor has completed, but its U-predecessor has not, then it is said to be *PL-blocked*. Note that a first-stage job cannot be PL-blocked because it has no U-predecessor. Note also that the first job of any subtask cannot be PL-blocked because it has no L-predecessor.

PL-blocking is illustrated in Fig. 3(a). Note that, in the figure, we use J to denote job $T_{i,v}^k$, J_U to denote J's U-predecessor $T_{i,v}^{k-1}$ (if $k \geq 2$), and J_L to denote J's L-predecessor $T_{i,v-1}^k$ (if $v \geq 2$). This shorthand notation is sometimes used in the discussion that follows.

If no job of any subtask of any pipeline task is PL-blocked, then every job's U-predecessor completes no later than its L-predecessor. In this case, the precedence constraint enforced by the periodic task model, which requires consecutive jobs of the same subtask to execute in sequence, is sufficient to ensure that the pipeline task is scheduled correctly. Therefore, it is the PL-blocking effect that makes pipeline task scheduling different from ordinary periodic task scheduling. (Note that, when jobs are never tardy, PL-blocking cannot happen.)

$T_{l,j}^h$ is scheduled at time t in S^{NPS} if and only if it is scheduled at time t in S^{PS}. A similar interpretation of "equivalent" applies to Theorem 2.

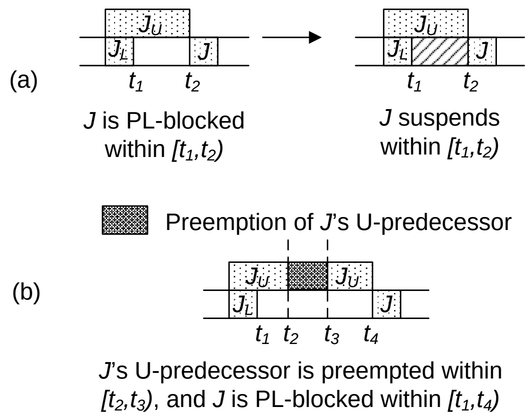

(a)
J_U
J_L
J
t_1
t_2
J is PL-blocked within $[t_1,t_2)$
J suspends within $[t_1,t_2)$
Preemption of J's U-predecessor
(b)
t_1 t_2 t_3 t_4
J's U-predecessor is preempted within $[t_2,t_3)$, and J is PL-blocked within $[t_1,t_4)$

Figure 3: **(a)** Job J is PL-blocked at time t. **(b)** Treating PL-blockings as suspensions.

Based upon the above observation, we intend to transform pipeline tasks into suspending tasks by treating PL-blocking times as suspensions, and eliminating precedence constraints among pipeline stages. The transformation converts an h-stage pipeline task into h independent suspending tasks. If a job is PL-blocked within the time interval $[t_1, t_2)$, then it could be deemed to suspend within $[t_1, t_2)$, as illustrated in Fig. 3(a). The new system obtained by performing such a transformation may have schedules in which the eliminated precedence constraint is violated. However, each schedule of the old system is also a valid schedule of the new system, so any per-(sub)task response time bound established for the new system is applicable to the old system as well.

In order to treat PL-blockings as suspensions, we need to upper-bound a job's PL-blocking time. However, preemptions (as defined below) may cause difficulties in upper-bounding PL-blocking times.

Definition 4. If job $T_{i,v}^w$ is enabled at time t and does not suspend at t, but does not execute at t, then it is *preempted* at t. The total time for which $T_{i,v}^w$ is preempted is called its *preemption time*.

Fig. 3(b) illustrates why preemption is a problem. Here, J is PL-blocked within $[t_1, t_4)$ and its U-predecessor is preempted within $[t_2, t_3)$. Since it is difficult to upper-bound preemption times, it is also difficult to upper-bound PL-blocking times. The situation is made even more complicated by the fact that "upstream" jobs in the pipeline must also be considered. These jobs are characterized as follows.

Definition 5. We define the set of *U-jobs* of any job $T_{i,v}^w$ to be $T_{i,v}^{w-1}, T_{i,v}^{w-2}, ..., T_{i,v}^1$.

A U-job of $T_{i,v}^w$ is a job of the same pipeline task that may impact the scheduling of $T_{i,v}^w$, directly or indirectly, through precedence constraints.

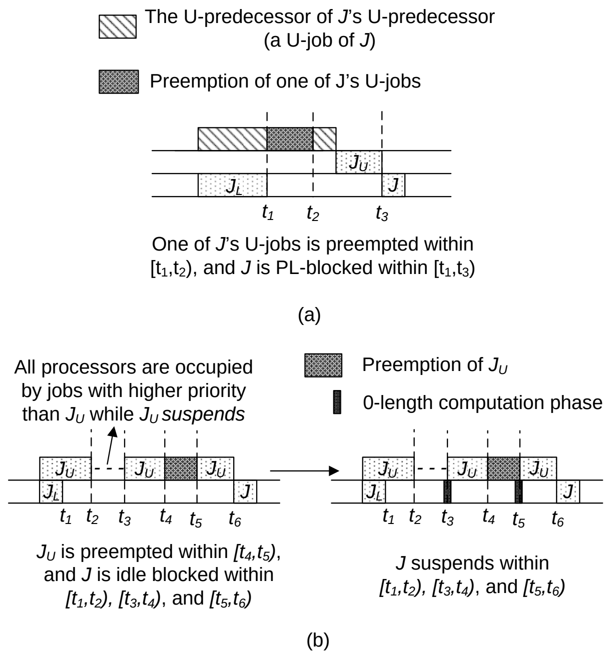

The U-predecessor of J's U-predecessor (a U-job of J)
Preemption of one of J's U-jobs
J_U
J_L
J
t_1 t_2 t_3
One of J's U-jobs is preempted within $[t_1,t_2)$, and J is PL-blocked within $[t_1,t_3)$
(a)
All processors are occupied by jobs with higher priority than J_U while J_U suspends
Preemption of J_U
0-length computation phase
t_1 t_2 t_3 t_4 t_5 t_6
J_U is preempted within $[t_4,t_5)$, and J is idle blocked within $[t_1,t_2)$, $[t_3,t_4)$, and $[t_5,t_6)$
J suspends within $[t_1,t_2)$, $[t_3,t_4)$, and $[t_5,t_6)$
(b)

Figure 4: **(a)** The preemption of a U-job of J may increase its PL-blocking time. **(b)** Treating idle blocking time as suspensions.

The preemption of any U-job of J may increase J's PL-blocking time. Consider, for example, Fig. 4(a). Here, one of J's U-jobs (other than J_U) is preempted within $[t_1, t_2)$, which causes J's PL-blocking time to be $t_3 - t_1$.

Fortunately, although preemption causes problems in upper-bounding PL-blocking times, it is only necessary to upper-bound idle blocking times, as defined below, in order to transform a pipeline task into independent suspending tasks. This is because, if one of $T^w_{i,v}$'s U-jobs is preempted at time t while $T^w_{i,v}$ is blocked, then there can be no idle processors at t. Hence, it is not necessary to consider $T^w_{i,v}$ to be suspended at that time.

Definition 6. If $T^w_{i,v}$ is PL-blocked at time t, and at least one processor is idle or occupied by a job with lower priority than $T^w_{i,v}$, then $T^w_{i,v}$ is *idle blocked* at t. The total time for which $T^w_{i,v}$ is idle blocked within a time interval $[t, t')$ is called its *idle blocking time* within $[t, t')$.

To see that it is only necessary to treat a job's total idle blocking time as suspensions, consider Fig. 4(b). As seen in the figure, we can consider J to suspend within $[t_1, t_2)$, and then execute a 0-length computation phase, which must be scheduled by GEDF at some t' within $[t_3, t_6)$, where t_6 is the first time at which J is scheduled. ($t' = t_3$ in Fig. 4(b).) If $t' < t_4$, then J can be defined to suspend within $[t', t_4)$. Then J executes another 0-length computation phase, which must be scheduled by GEDF at some t'' within $[t'', t_6)$. ($t'' = t_5$ in Fig. 4(b).) Finally J can then be defined to suspend within $[t_5, t_6)$. Note that the total needed suspension length is upper-bounded by the total idle-blocking time of J. The same is true of more complicated scenarios. (Later, we consider the possibility of adding some of the needed suspension time to account for J's idle blocking to J_L instead of J.)

To treat idle blocking times as suspensions, an upper bound on the total idle blocking time of a job must be determined. This is dealt with in Lemma 2 below.

Definition 7. If $T^w_{i,v}$'s first phase is an execution (suspension) phase and it begins executing (a suspension) for the first time at t, then t is called its *start time*, denoted $S(T^w_{i,v})$. If $T^w_{i,v}$'s last phase (be it execution or suspension) completes at time t', then t' is called its *finish time*, denoted $F(T^w_{i,v})$. Let $ET(T^w_{i,v})$ denote the first time instant when when $T^w_{i,v}$ is enabled.

Lemma 2. *The total idle blocking time of any job $T^k_{i,v}$ in τ^{PS} is at most $\sum_{j=1}^{j=k-1}(e^j_i + s^j_i)$.*

Proof. We prove the claim by induction on k. The base case is trivial: $T^1_{i,v}$ has no U-predecessor and thus is not PL-blocked. We shall now prove the induction step, $k > 1$. Note that if J_L does not exist, then J is released at time 0 and will not be blocked, by Def. 3. So, assume that J_L exists. For simplicity, let J_{UL} denote $T^{k-1}_{i,v-2}$, and J_{UU} denote $T^{k-2}_{i,v-2}$, as shown in Fig. 5 (a). Let $e_U = e^{k-1}_i$, $e_L = e^k_i$, $s_U = s^{k-1}_i$, and $s_L = s^k_i$, and let $t = S(J_L)$ and $t' = F(J_L)$. Also let e'_U (e'_L) denote the actual run-time execution time of J_U (J_L). Note that if J is not released at or before t', then J is not blocked at t', but may be after t', in which case J's idle blocking time may not be maximal. Thus, it suffices to assume that J is released at or before t'.

Case 1: J_{UU} is complete at t (or, it does not exist), as shown in Fig. 5 (b). Observe that J_{UL} is complete by time t (otherwise, J_L could not execute). Thus, by the condition of Case 1, J_U is enabled at or before t, which implies that all U-jobs of J other than J_U complete at or before t. If J_U completes by time t', then J is not PL-blocked, so assume otherwise. By our priority definition, in $[t, t')$, J_U executes or suspends whenever J_L executes (for e'_L time units). Therefore, after t', J_U has at most $\max(0, e'_U + s'_U - e'_L)$ computation and suspension time left. Given that J_{UU} has completed at t, J_U is the only U-job of J that may not have completed by time t'. By Def. 3, job J's total idle blocking time is given by the unfinished computation and suspension time of J's U-jobs after t'. Thus, $T^k_{i,v}$'s total idle blocking time is at most[2] $\max(0, e'_U + s_U - e'_L) \le e_U + s_U$, which equals $e^{k-1}_i + s^{k-1}_i$. Therefore, $T^k_{i,v}$'s total idle blocking time is at most $\sum_{j=1}^{j=k-1}(e^j_i + s^j_i)$.

[2]Since we do not assume that every job executes for its worst-case execution time, J_L could execute for zero time units at run-time.

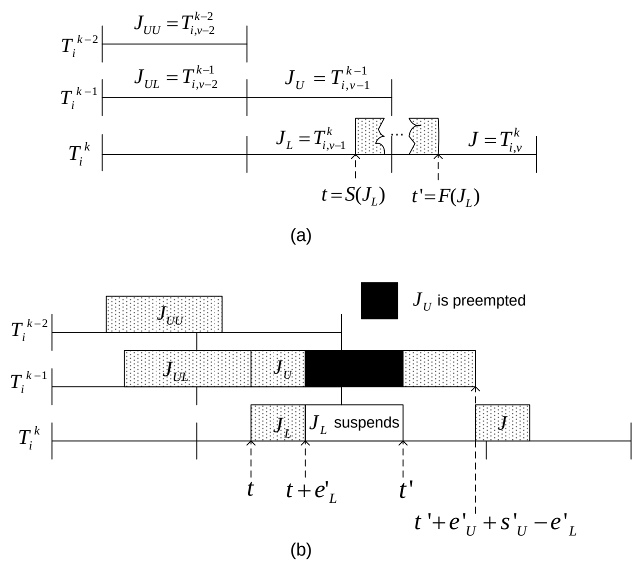
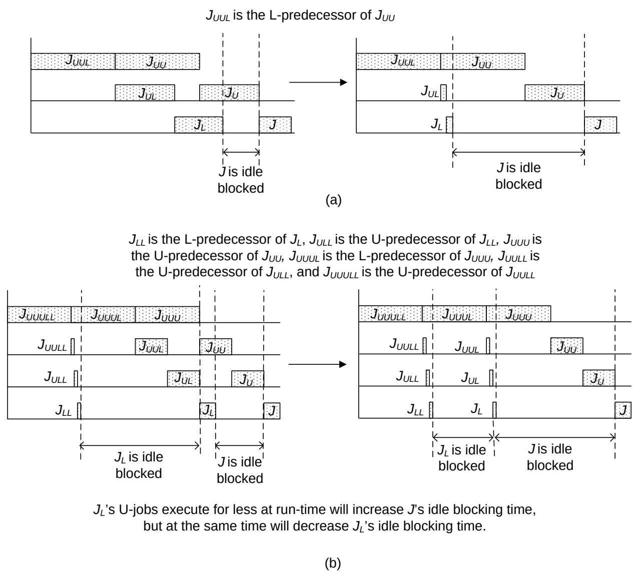


Figure 5: **(a)** Upper bounding the idle blocking time. **(b)** Case 1.

Case 2: J_{UU} is not complete at t. Since J_{UL} must have been complete at t (otherwise, J_L cannot start at t), J_U is blocked at or after t. By the induction hypothesis, J_U idle blocks for at most $\sum_{j=1}^{j=k-2}(e_i^j + s_i^j)$ time at or after t. Thus, $T_{i,v}^k$'s total idle blocking time is at most the computation and suspension time left by J's U-jobs after t, which is at most J_U's total idle blocking time plus J_U's computation and suspension time, which is $\sum_{j=1}^{j=k-1}(e_i^j + s_i^j)$. $\square$

If J_L or J_L's U-jobs execute or suspend for less than their worst-case costs at run-time, then J's idle blocking time can increase, as illustrated in Fig. 6(a). In Case 1 of Lemma 2, we assumed that J_L could execute for zero time units (see the footnote). And in Case 2 of Lemma 2, by applying the induction hypothesis to J_U, we are assuming that J_L and all U-jobs of J_L could execute for zero time units, in which case J_L completes at the earliest possible time point and J's idle blocking time becomes maximal, as shown in Fig. 6(b). However, the proof in Lemma 2 is too pessimistic because it ignores the fact that if J_L's U-jobs execute for less at run-time, then the idle blocking time of J_L will be decreased at the same time, as shown in Fig. 6(b). Lemma 4 deals with this pessimism and minimizes the per-job idle blocking time needed to be treated as suspensions. Before proving Lemma 4, we first prove Lemma 3, which gives the latest possible enabled time of any tardy job $T_{i,v}^k$.

Definition 8. Let $\max_k = \max\{j \mid 1 \leq j \leq k \leq m \wedge (\forall w : 1 \leq w \leq k : e_i^j + s_i^j \geq e_i^w + s_i^w)\}$. That is, the maximum total execution cost and suspension length among the subtasks $\{T_i^1, T_i^2, ..., T_i^k\}$ is maximal for $T_i^{max_k}$. If $k = 0$, then let $max_k = 1$.



J_{UUL} is the L-predecessor of J_{UU}
J is idle blocked
(a)
J_{LL} is the L-predecessor of J_L, J_{ULL} is the U-predecessor of J_{LL}, J_{UUU} is the U-predecessor of J_{UU}, J_{UUUL} is the L-predecessor of J_{UUU}, J_{UULL} is the U-predecessor of J_{ULL}, and J_{UUUUL} is the U-predecessor of J_{UULL}
J_L is idle blocked
J is idle blocked
J_L's U-jobs execute for less at run-time will increase J's idle blocking time, but at the same time will decrease J_L's idle blocking time.
(b)

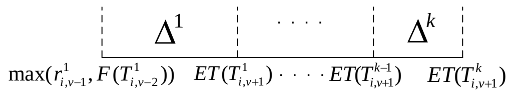
Figure 6: **(a)** J's idle blocking time can be increased if J_L and J_L's U-jobs execute for less at run-time. **(b)** The idle blocking time of J_L will be decreased if J_L's U-jobs execute for less at run-time.

Definition 9. Let Δ^1 be the total length of all maximal sub-intervals within $[max(r_{i,v-1}^1, F(T_{i,v-2}^1)), ET(T_{i,v+1}^1))$ during which some U-job of $T_{i,v+1}^k$ or $T_{i,v}^k$ is preempted. Let Δ^j (where $1 < j \leq k$) be the total length of all maximal sub-intervals within $[ET(T_{i,v+1}^{j-1})), ET(T_{i,v+1}^j))$ during which some U-job of $T_{i,v+1}^k$ or $T_{i,v}^k$ is preempted. This is illustrated in Fig. 7.

Lemma 3. *If $v \geq 3 - k$ (recall that by adding "extra" initial jobs, $T_{i,3-k}^k$ is the second job of T_i^k), $k > 1$, and L3.1 and L3.2 below hold, then $ET(T_{i,v+1}^k) \leq max(r_{i,v}^1, F(T_{i,v-1}^1)) + \sum_{j=1}^{k}(\Delta^j) + k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$.*
***L3.1**: $T_{i,v}^{k-1}$ and $T_{i,v+1}^{k-1}$ are both tardy.*
***L3.2**: $T_{i,v}^k$ completes before $T_{i,v+1}^{k-1}$.*

Proof. We prove the claim by induction on k. For the base case, $k = 2$, we have $ET(T_{i,v+1}^2) \overset{L3.1 \text{ and } L3.2}{=} F(T_{i,v+1}^1) \leq ET(T_{i,v+1}^1) + \Delta^2 + e_i^1 + s_i^1 \overset{L3.1}{=} F(T_{i,v}^1) + \Delta^2 + e_i^1 + s_i^1 \leq max(r_{i,v}^1, F(T_{i,v-1}^1)) + \Delta^1 + e_i^1 + s_i^1 + \Delta^2 + e_i^1 + s_i^1 \leq max(r_{i,v-1}^1, F(T_{i,v-2}^1)) + \Delta^1 + \Delta^2 + 2 \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$.

For the induction step, $k > 2$, we have $ET(T_{i,v+1}^k) \overset{L3.1 \text{ and } L3.2}{=} F(T_{i,v+1}^{k-1}) \leq ET(T_{i,v+1}^{k-1}) + \Delta^k + e_i^{k-1} + s_i^{k-1} \leq ET(T_{i,v+1}^{k-1}) + \Delta^k + e_i^{max_{k-1}} + s_i^{max_{k-1}} \overset{Ind.\ Hyp.}{\leq} max(r_{i,v-1}^1, F(T_{i,v-2}^1)) + \sum_{j=1}^{k-1}\Delta^j + (k-1) \cdot (e_i^{max_{k-2}} + s_i^{max_{k-2}}) + \Delta^k + e_i^{max_{k-1}} + s_i^{max_{k-1}} \leq$



Δ^1
$\cdots$
Δ^k
$\max(r^1_{i,v-1}, F(T^1_{i,v-2}))$
$ET(T^1_{i,v+1}) \cdots ET(T^{k-1}_{i,v+1})$
$ET(T^k_{i,v+1})$

Figure 7: Def. 9.

$max(r^1_{i,v-1}, F(T^1_{i,v-2})) + \sum_{j=1}^{k} \Delta^j + k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$. $\square$

Lemma 4. *The total idle blocking time*[3] *of any two consecutive jobs $T^k_{i,v}$ and $T^k_{i,v+1}$ in τ^{PS} is at most $k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$.*

Proof. If $k = 1$, then $T^k_{i,v}$ and $T^k_{i,v+1}$ cannot be idle blocked, so assume $k > 1$. We prove the lemma in this case by induction on v.

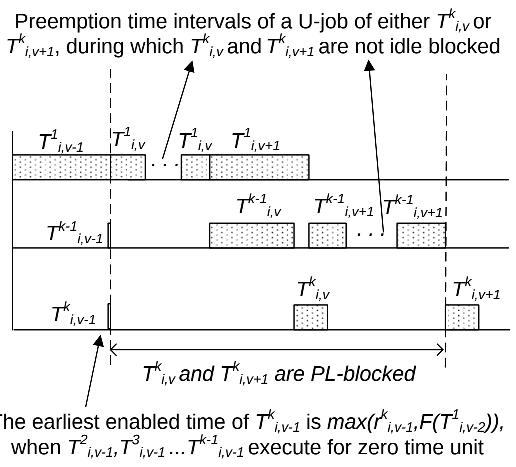
If at most one of $T^k_{i,v}$ and $T^k_{i,v+1}$ is idle-blocked, then by Lemma 2, the idle blocking time for both is at most $\sum_{j=1}^{j=k-1}(e_i^j + s_i^j)$, which is at most $k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$. This reasoning applies in the base case (where $v = 2 - k$) because $T^k_{i,2-k}$ has no L-predecessor and thus is not idle blocked. In the rest of the proof, we consider the induction step, $v > 2 - k$, and assume that $T^k_{i,v}$ and $T^k_{i,v+1}$ are both idle blocked. By Def. 6, this implies that both $T^{k-1}_{i,v}$ and $T^{k-1}_{i,v+1}$ are tardy, and $T^k_{i,v}$ completes before $T^{k-1}_{i,v+1}$.

As shown in Fig. 8, by Def. 3, $T^k_{i,v}$ and $T^k_{i,v+1}$ can be PL-blocked only in the interval $[ET(T^k_{i,v-1}), ET(T^k_{i,v+1}))$. Given that $T^1_{i,v-1}$ is the first-stage U-job of $T^k_{i,v-1}$, $ET(T^k_{i,v-1}) \geq max(r^1_{i,v-1}, F(T^1_{i,v-2}))$. Thus, $T^k_{i,v}$ and $T^k_{i,v+1}$ can be PL-blocked only within $[max(r^1_{i,v-1}, F(T^1_{i,v-2})), ET(T^k_{i,v+1}))$. By Def. 6, $T^k_{i,v}$ (respectively, $T^k_{i,v+1}$) is idle blocked at $t \in [max(r^1_{i,v-1}, F(T^1_{i,v-2})), ET(T^k_{i,v+1}))$ only if, at t, not all processors are occupied by jobs with higher priority than $T^k_{i,v}$ (respectively, $T^k_{i,v+1}$).

Thus, for any time $t \in [max(r^1_{i,v-1}, F(T^1_{i,v-2})), ET(T^k_{i,v+1}))$, if any U-job of $T^k_{i,v}$ or $T^k_{i,v+1}$ is preempted at t, then neither $T^k_{i,v}$ nor $T^k_{i,v+1}$ is idle blocked at t. This is because, by our priority definition, any U-job of $T^k_{i,v}$ or $T^k_{i,v+1}$ has higher priority than $T^k_{i,v}$ and $T^k_{i,v+1}$. Thus, all processors are occupied by jobs with higher priority than $T^k_{i,v}$ and $T^k_{i,v+1}$ at t. The total length of such time intervals within $[max(r^1_{i,v-1}, F(T^1_{i,v-2})), ET(T^k_{i,v+1}))$, during which some U-job of $T^k_{i,v}$ or $T^k_{i,v+1}$ is preempted, is given by $\sum_{j=1}^{k} \Delta^j$, by Def. 9.

By Lemma 3, the latest possible enabled time of $T^k_{i,v+1}$ is $ET(T^k_{i,v+1}) \leq max(r^1_{i,v-1}, F(T^1_{i,v-2})) + \sum_{j=1}^{k} \Delta^j + k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$. Thus, the total idle blocking time of $T^k_{i,v}$ and $T^k_{i,v+1}$ is at most

[3]By this, we mean the sum of all maximal sub-intervals during which at least one of $T^k_{i,v}$ and $T^k_{i,v+1}$ is idle-blocked.



Preemption time intervals of a U-job of either $T^k_{i,v}$ or $T^k_{i,v+1}$, during which $T^k_{i,v}$ and $T^k_{i,v+1}$ are not idle blocked
$T^1_{i,v-1}$
$T^1_{i,v}$
$T^1_{i,v}$
$T^1_{i,v+1}$
$T^{k-1}_{i,v-1}$
$T^{k-1}_{i,v}$
$T^{k-1}_{i,v+1}$
$T^{k-1}_{i,v+1}$
$T^k_{i,v-1}$
$T^k_{i,v}$
$T^k_{i,v+1}$
$T^k_{i,v}$ and $T^k_{i,v+1}$ are PL-blocked
The earliest enabled time of $T^k_{i,v-1}$ is $max(r^k_{i,v-1}, F(T^1_{i,v-2}))$, when $T^2_{i,v-1}, T^3_{i,v-1} \ldots T^{k-1}_{i,v-1}$ execute for zero time unit

Figure 8: Lemma 4.

$ET(T^k_{i,v+1}) - max(r^1_{i,v-1}, F(T^1_{i,v-2}))) - \sum_{j=1}^{k} \Delta^j \leq max(r^1_{i,v-1}, F(T^1_{i,v-2})) + \sum_{j=1}^{k} \Delta^j + k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}}) - max(r^1_{i,v-1}, F(T^1_{i,v-2})) - \sum_{j=1}^{k} \Delta^j = k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$. $\square$

Thus, for any job $T^k_{i,v}$, if we increase s^k_i by $\dfrac{k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})}{2}$, then by Lemma 4, we are able to treat every job's idle blocking time as suspension time. Note that it might be necessary to treat J's idle blocking time as J_L's suspension time (this suspension time would be added to the end of J_L's execution instead of the beginning of J's). However, the total idle blocking time for any two consecutive jobs are upper-bounded by $k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})$. In this way, we can transform each subtask T^h_l to an independent suspending task. Note that we do not need to transform first-stage subtasks because they cannot be PL-blocked (see Def. 3). By transforming every subtask in τ^{PS} in this way, we obtain τ^S.

Example 2. Consider a pipeline task T_1 with three stages, where $(e^1_1, s^1_1) = (1, 1)$, $(e^2_1, s^2_1) = (2, 1)$, and $(e^3_1, s^3_1) = (1, 1)$. T^3_1 can be tramsformed into an independent suspending task by adding $\dfrac{k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})}{2} = 4.5$ time units to s^3_1, and T^2_1 can be transformed into an independent suspending task by adding $\dfrac{k \cdot (e_i^{max_{k-1}} + s_i^{max_{k-1}})}{2} = 3$ time units to s^2_1. Thus, this pipeline task can be transformed into three independent suspending tasks T_1, T_2, and T_3, where $(e_1, s_1) = (1, 1)$, $(e_2, s_2) = (2, 4)$, and $(e_3, s_3) = (1, 5.5)$.

Theorem 2. *For any concrete instantiation $\overline{\tau}^{PS}$ of τ^{PS}, there exists a concrete instantiation $\overline{\tau}^{S}$ of τ^S such that $\overline{\tau}^{PS}$ and $\overline{\tau}^{S}$ have equivalent GEDF schedules.*

Note again that this transformation strongly exploits the fact that, in our task model, suspension phases are upper-bounded, and hence, can be reduced to reflect actual PL-blocking times. Note also that the "reverse" of this theorem may not hold: for a concrete instantiation $\overline{\tau}^S$ of τ^S, there may not exist a concrete instantiation $\overline{\tau}^{PS}$ of τ^{PS} such that $\overline{\tau}^S$ and $\overline{\tau}^{PS}$ have equivalent GEDF schedules.

Corollary 2. *For any subtask T_l^h, if the maximum response time of any job of T_l^h in any GEDF schedule for τ^S is z time units, then the maximum response time of any such job in any GEDF schedule for τ^{PS} is at most z time units.*

5 Tardiness Analysis

In this section, we first overview some recent results regarding tardiness in task systems with only suspensions [9], as stated in Theorem 3. Then we analyze the tardiness of NPS task systems by transforming such systems into systems with only suspensions.

5.1 Tardiness Bound for Suspending Task Systems

A suspending task system is just a special case of our system model as described in Sec. 2, where $b_{max} = 0$ and each task is a pipeline task consisting of only one subtask. For conciseness, we drop the superscript "1" as used in the pipeline task model when referring to a first-stage subtask, and simply use $T_{i,j}$, $r_{l,j}$, $d_{l,j}$, e_l, and s_l. We require $e_l+s_l \leq p_l$ and $U_{sum} \leq m$; otherwise, tardiness can grow unboundedly. A common case for real-time workloads is that both suspending tasks and computational tasks (which do not suspend) co-exist. To reflect this, we let U_{sum}^s denote the total utilization of all suspending tasks, and U_{sum}^c denote the total utilization of all computational tasks.

Definition 10. Let $s_{max} = \max\{s_1, s_2, ..., s_n\}$. Let $\xi_i = \dfrac{s_{max}}{s_{max} + e_i}$ be the *suspension ratio* of T_i. Let $\xi_{max} = \max\{\xi_1, \xi_2, ..., \xi_n\}$ be the *maximum suspension ratio*.

Definition 11. Let E_{sum}^s be the total execution cost of all suspending tasks in τ^S. Let E_{sum} be the total execution cost of all tasks in τ^S. Let S_{sum}^s be the total suspension length of all tasks in τ^S. Let u_{max}^s be the maximum utilization of any suspending task in τ^S.

Definition 12. Let U_L^c be the sum of the $\min(m-1, c)$ largest computational task utilizations, where c is the number of computational tasks in τ^S. Let E_L^c be the sum of the $\min(m-1, c)$ largest computational task execution costs in τ^S.

Definition 13. Let $V = E_{sum}^s + E_L^c + u_{max}^s \cdot S_{sum}^s + (m-1)e_l + m \cdot s_l + 3n \cdot s_{max}$.

Theorem 3. *[9] The tardiness of any task T_l in τ^S scheduled under GEDF is at most $x + e_l + s_l$, where*

$$x \geq \frac{V}{(1-\xi_{max}) \cdot m - U_{sum}^s - U_L^c}, \tag{1}$$

provided $U_{sum}^s + U_L^c < (1-\xi_{max}) \cdot m$.

x is well-defined provided $U_{sum} + U_L^c < (1-\xi_{max}) \cdot m$. If this condition holds and x equals the right-hand side of (1), then the tardiness of $T_{l,j}$ will not exceed $x + e_l + s_l$. A value for x that is independent of the parameters of T_l can be obtained by replacing $(m-1)e_l + m \cdot s_l$ with $max_l((m-1)e_l + m \cdot s_l)$ in V.

It is worth noting that this approach allows certain suspending tasks to be designated as computational tasks. In particular, all suspension phases of any task T_l can be treated as computation phases, and T_l can be considered to be a computational task with execution cost of $e_l + s_l$ time units and $u_l = \dfrac{e_l + s_l}{p_l}$.

5.2 Tardiness Bound for NPS Task Systems

By transforming a periodic NPS task system into a periodic task system with only suspensions, we can apply Theorem 3 to the transformed task system and derive a tardiness bound for the NPS task system.

Definition 14. *Ordinary tasks* are those tasks in τ^{NPS} that do not contain suspensions, pipelines, and non-preemptive sections.

According to the transformations described in Secs. 3-5, for any subtask T_l^k in τ^{NPS}, we increase s_l^k by $c_l^k \cdot b_{max}$ (the maximum NP-blocking time as stated in Lemma 1). Then for any non-first-stage subtask T_l^h (where $h \geq 2$), we increase s_l^h by $(h \cdot (e_l^{max_{h-1}} + s_l^{max_{h-1}}))/2$ (to account for idle blocking times, which are bounded as in Lemma 4). Then every subtask in τ^{NPS} can be transformed into an independent task with only suspensions, which is mapped to a task in τ^S with the same task parameters. Since an ordinary task has only one computation phase, it can be NP-blocked for at most b_{max} time units. Thus, each ordinary task becomes a suspending task with a suspension length of b_{max} time units. Since in real systems most scenarios where non-preemptivity arises are due to system calls and critical sections that are very short compared to job execution times, we can consider each ordinary task to be a computational task after the transformations by treating the suspension phase of any ordinary task T_l as a computation phase. That is, T_i is considered to be a computational task with execution cost of $e_l + b_{max}$ time units and $u_i = (e_i + b_{max})/p_i$.

By Corollaries 1–2, any maximum per-subtask tardiness bound for τ^S holds for τ^{NPS} as well. After transforming τ^{NPS} into τ^S, we can apply Theorem 3 to τ^S and obtain a tardiness bound for τ^{NPS}, as stated in the following theorem.

Theorem 4. *The tardiness of any task T_l in τ^S (and thus τ^{NPS}) scheduled under GEDF is at most $x + e_l + s_l$, where x is defined in Eq. (1), provided $U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot m$.*

According to the suspending task model defined in Sec. 5.1, for any task T_l, it is required that $e_l + s_l \leq p_l$. Therefore, given that any subtask T^h_l in τ^{NPS} is transformed into an independent suspending task by increasing its suspension length, in order to apply Theorem 4 to τ^{NPS}, it is required that, for any subtask T^h_l, $c^h_l \cdot b_{max} + e^h_l + s^h_l + (h \cdot (e^{max_{l-1}}_l + s^{max_{h-1}}_l))/2 \leq p_l$. For any ordinary task T_i, it is required that $e_i + b_{max} \leq p_i$.

6 Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the applicability of the tardiness bound in Theorem 4. Our goal is to examine how restrictive the theorem's utilization cap is, for varying parameter choices.

Definition 15. A task set is said to be *schedulable* if it satisfies the utilization cap as stated in Theorem 4 and thus can have bounded tardiness.

Our methodology for generating random task sets was guided by practical considerations in two ways. First, we assume that non-preemptive sections are relatively short, for reasons discussed earlier. Second, we assume that for each pipeline, suspensions occur only in first- and last-stage subtasks. This is reflective of real-world scenarios in which input data is read in the first stage and output data is written in the last stage. With these considerations in mind, task sets were generated as follows. Task periods were uniformly distributed over [200*ms*,300*ms*].[4] Subtask utilizations were uniformly distributed over $[0.001, 0.3]$. Suspension lengths and non-preemptive section lengths were controlled by two parameters, R_{SE} and R_{NPE}. R_{SE} is defined as S/E, where S is the suspension length of a subtask and E is the execution cost of that subtask. R_{SE} was varied as follows: 0.01 (suspensions are short), 0.05 (suspensions are moderate), and 0.1 (suspensions are long). The suspension-length ranges generated by these parameters can reasonably model the suspension lengths of real-world applications [9], as shown in Table 1. R_{NPE} is defined as b_{max}/e_{min}, where e_{min} is the minimum execution cost among all subtasks. Given the above comments about non-preemptive sections being typically short, R_{NPE} was set to be 0.01 in our experiments. To control the maximum idle-blocking time, a parameter s_{max}, called *pipeline stretch*, was used, as defined below.

[4]Although we are using 1*ms* as a time unit here, the definition of a time unit is flexible. For example, we could re-define a time unit to be 0.1*ms*, giving a period range of [20*ms*,30*ms*]. Such a range might be more suitable for multimedia applications.

Table 1: Per-job suspension-length ranges.

suspension length	short suspensions *R_{SE} = 0.01*	moderate suspensions *R_{SE} = 0.05*	long suspensions *R_{SE} = 0.1*
min:	2 *μs*	10 *μs*	20 *μs*
avg:	570 *μs*	3 ms	5.6 *ms*
max:	1.5 *ms*	7.5 *ms*	15 *ms*

Table 2: Average tardiness as computed via Theorem 4.

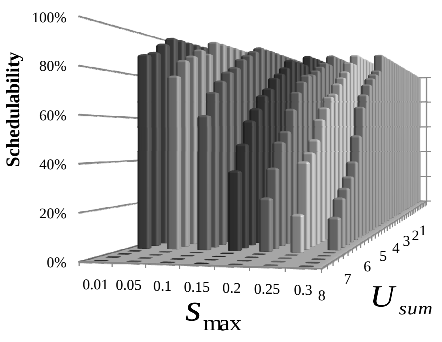
suspension length	short suspensions $R_{SE} = 0.01$	moderate suspensions $R_{SE} = 0.05$	long suspensions $R_{SE} = 0.1$
Tardiness	167.5 *ms*	404.8 *ms*	824.2 *ms*

Definition 16. Let $s^w_i = \dfrac{e^{max_w}_i + s^{max_w}_i - e^w_i - s^w_i}{e^{max_w}_i + s^{max_w}_i}$. s^w_i is called the *subtask stretch* of T^w_i. Let $s_i = \max\{s^1_i, s^2_i, ..., s^m_i\}$. s_i is called the *task stretch* of pipeline task T_i. Let $s_{max} = \max\{s_1, s_2, ..., s_n\}$. s_{max} is called the *pipeline stretch.*

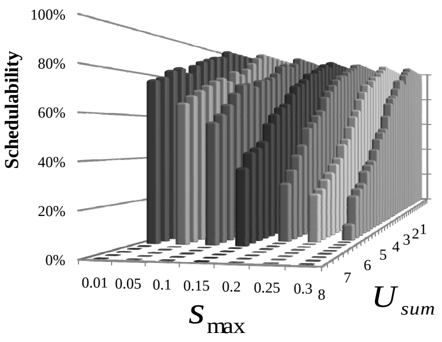
s_{max} was varied over (0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3). A larger value of s_{max} indicates a larger value of the maximum idle blocking time (as stated in Lemma 2), which negatively impacts schedulability. We also varied U_{sum} within $\{1, 2, 3, 4, 5, 6, 7, 8\}$ and let 90% of the tasks in each task set be ordinary tasks. For each combination of (R_{SE}, s_{max}, U_{sum}), 1,000 task sets were generated for an eight-processor system.

The schedulability results that were obtained are shown in Fig. 9. Each column in this figure shows the percentage of the generated task sets for that set of parameter choices that were deemed to be schedulable. In general, very high schedulability can be achieved when R_{SE} and s_{max} are kept small. For instance, when $R_{SE} = 0.01$ and $s_{max} \leq 0.05$, almost 100% of all task sets are schedulable if the system is not heavily-utilized, as shown in Fig. 9 (a). Moreover, high schedulability can be achieved when s_{max} is kept small and the system is moderately loaded. For instance, when $s_{max} \leq 0.05$ and $U_{sum} \leq 4$, around 100% (respectively, 90%/70%) of all task sets are schedulable when suspensions are short (respectively, moderate/long). On the other hand, our analysis is negatively impacted by increasing any one of these three parameters. Given that the utilization constraint stated in Theorem 3 depends crucially on the maximum suspension length and the total system utilization, increasing any one of these parameters in the original NPS task system will either increase the suspension length of the corresponding tasks in the transformed system or increase the total utilization, thus further increasing the possibility for a task set to violate the utilization constraint.

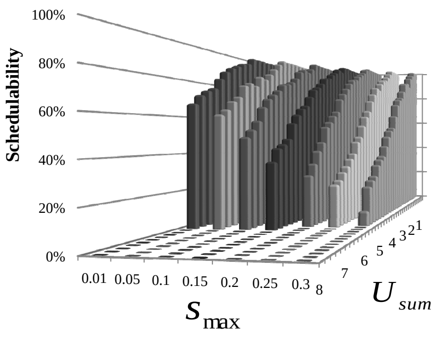
In addition to schedulability, the magnitude of tardiness,



(a)



(b)



(c)

Figure 9: Schedulability results. **(a)** Short suspensions ($R_{SE} = 0.01$). **(b)** Moderate suspensions ($R_{SE} = 0.05$). **(c)** Long suspensions ($R_{SE} = 0.1$). Each figure gives a 3-D graph in which schedulability is shown under different combinations of two parameters: s_{max} and U_{sum}.

as computed using Theorem 4, is of importance. Table. 2 depicts the average of the computed bounds for each of the tested scenarios in our experimental framework for the case where $U_{sum} = 4$ and $s_{max} = 0.05$ (that is, for each scenario in this case, an average of all bounds for all tasks in all schedulable task sets is given). As can be seen, tardiness is reasonable if the total utilization is moderate and suspensions are short or moderate. However, as suspension lengths increase, tardiness increases, as an examination of the bound in Theorem 4 suggests should be the case.

7 Conclusion

In this paper, we presented a method for transforming a periodic NPS task system into a simpler periodic task system with only suspensions. The transformation allows maximum response-time bounds derived for periodic suspending task systems to be applied to periodic NPS task systems. This allowed us to derive a deadline tardiness bound under GEDF for such systems. This bound shows that NPS task systems can be supported with bounded tardiness provided certain utilization constraints are met.

References

[1] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proc. of the 15th Euromicro Conf. on Real-Time Systems*, pp. 23-30, 2003.

[2] U. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Int'l Real-Time Systems Symp.*, pp. 330-341, 2005.

[3] P. Jayachandran and T. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *Proc. of the 20th Euromicro Conf. on Real-Time Systems*, pp. 233-242, 2008.

[4] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *Proc. of the 19th Euromicro Conf. on Real-Time Systems*, pp. 29-38, 2007.

[5] I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *Proc. of the 2nd Int'l Workshop on Real-Time Computing Systems and Applications*, pp. 54-59, 1995.

[6] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proc. of the 28th Real-Time Systems Symp.*, pp. 413-422, 2007.

[7] C. Liu and J. H. Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *Proc. of the 21st Euromicro Conf. on Real-Time Systems*, pp. 269–278, 2009.

[8] C. Liu and J. H. Anderson. Supporting sporadic pipelined tasks with early-releasing in soft real-time multiprocessor systems. In *Proc. of the 15th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, pp. 284-293, 2009.

[9] C. Liu and J. H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proc. of the 30th Real-Time Systems Symp.*, pp. 425-436, 2009.

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

[11] J. C. Palencia and M. Gonzlez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the 19th IEEE Real-Time Systems Symp.*, pp. 26-37, 1998.

[12] J. C. Palencia and M. Gonzlez Harbour. Response time analysis of EDF distributed real-time systems. In *J. Embedded Comput.*, Vol.1, pp. 225-237, 2005.

[13] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *Proc. of the 11th IEEE Int'l Real Time and Embedded Technology and Applications Symp.*, pp. 66-75, 2005.

[14] R. Rajkumar. Dealing with Suspending Periodic Tasks. IBM Thomas J. Watson Research Center, 1991.

[15] F. Ridouard and P. Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *Proc. of the 14th Real-Time and Network Systems*, pp. 15-24, 2006.

[16] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proc. of the 25th IEEE Real-Time Systems Symp.*, pp. 47-56, 2004.