An O(m) Analysis Technique for Supporting Real-Time Self-Suspending Task Systems *

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

Abstract

In many real-time and embedded systems, suspension delays may occur when tasks block to access shared resources or interact with external devices. Unfortunately, prior analysis methods for dealing with suspensions are quite pessimistic. In this paper, a novel technique is presented for analyzing soft real-time sporadic self-suspending task systems, for which bounded deadline tardiness is required, scheduled under global schedulers such as global EDF on multiprocessors (or EDF on uniprocessors). This technique is used to derive a new schedulability test that results in only $O(m)$ suspension-related utilization loss, where m is the number of processors. The derived test theoretically dominates prior tests with respect to schedulability. Furthermore, experiments presented herein show that the improvement over prior tests is often quite significant.

1 Introduction

In many real-time and embedded systems, suspension delays may occur when tasks block to access shared resources or interact with external devices such as I/O devices and network cards. For example, delays introduced by disk I/O range from $15\mu s$ (for NAND flash) to $15ms$ (for magnetic disks) per read [4]. Even longer delays are possible when accessing special-purpose devices such as digital signal processors or graphics processing units. Unfortunately, such delays cause intractabilities in schedulability analysis, even on uniprocessors [8]. Such negative results may explain the limited attention the general problem of analyzing real-time self-suspending task systems has received during the past 20 years. In this paper, we consider this problem in the context of analyzing globally scheduled soft real-time (SRT) sporadic self-suspending (SSS) multiprocessor task systems; under the definition of SRT considered in this paper, bounded deadline tardiness is required.

Perhaps the most commonly used approach for dealing with suspensions is *suspension-oblivious* analysis [7], which simply integrates suspensions into per-task worst-case execution time requirements. However, this approach yields $\Omega(n)$ utilization loss where n is the number of self-suspending tasks in the system. Unless the number of tasks is small and suspension delays are short, this approach may sacrifice significant system capacity. The alternative is to explicitly consider suspensions in the task model and resulting schedulability analysis; this is known as *suspension-aware* analysis. In prior work [6], we presented multiprocessor suspension-aware analysis for globally scheduled SRT SSS task systems. This analysis is an improvement over suspension-oblivious analysis for many task systems, but does not fully address the root cause of pessimism due to suspensions, and thus may still cause significant utilization loss.

The cause of pessimism in prior analysis. A key step in prior suspension-aware analysis [6] involves bounding the number of tasks that have enabled jobs (i.e., eligible for executing or suspending) at a specifically defined non-busy time instant t (i.e., at least one processor is idle at t). For ordinary task systems without suspensions, this number of tasks can be safely upper-bounded by $m-1$, where m is the number of processors, for otherwise, t would be busy. For SSS task systems, however, idle instants can exist due to suspensions even if m or more tasks have enabled jobs. The worst-case scenario that serves as the root source of pessimism in prior analysis is the following: *all n self-suspending tasks have jobs that suspend at some time t simultaneously, thus causing t to be non-busy*.

Key observation that motivates this research. Interestingly, the suspension-oblivious approach eliminates the worst-case scenario just discussed, albeit at the expense of pessimism elsewhere in the analysis. That is, by converting all n tasks' suspensions into computation, the worst-case scenario is avoided because then at most $m-1$ tasks can have enabled jobs at any non-busy time instant. *However, converting all n tasks' suspensions into computation is clearly overkill when attempting to avoid the worst-case scenario; rather, converting at most m tasks' suspensions into computation should suffice.* This observation motivates the new analysis technique we propose, which yields a much improved schedulability test with only $O(m)$ suspension-related utilization loss. Recent experimental results suggest that global algorithms should be limited to (sub-)systems with modest core counts (e.g., up to eight cores) [1]. Thus, m is likely to be small and much less than n in many settings.

Overview of related work. An overview of work on scheduling SSS task systems on uniprocessors (which we omit here due to space constraints) can be found in [6]. Such analysis (although pessimistic) can be applied on a per-processor basis to deal with suspensions under partitioning approaches. On globally scheduled multiprocessors, other than the suspension-oblivious approach, the only existing

*Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

suspension-aware approach known to us is that mentioned earlier [6], which is applicable to SRT systems. Unfortunately, as discussed earlier, utilization loss under these approaches can be significant.

Contributions. In this paper, we derive a schedulability test that shows that any given SSS task system is schedulable under global earliest-deadline-first (GEDF) scheduling with bounded tardiness if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ holds, where U_{sum} is the total system utilization and v^j is the j^{th} maximum suspension ratio, where a task's *suspension ratio* is given by the ratio of its suspension time over its period. We show that our derived schedulability test theoretically dominates prior approaches [6, 7]. Moreover, we show via a counterexample that task systems that violate our utilization constraint may have unbounded tardiness. As demonstrated by experiments, our proposed test significantly improves upon prior methods with respect to schedulability, and is often able to guarantee schedulability with little or no utilization loss while providing low predicted tardiness.

For readability and conciseness, we have chosen to focus upon a specific global scheduling algorithm: GEDF. However, our suspension-analysis technique can also be applied to any window-constrained [5] global scheduling algorithm with minor modifications. Moreover, recent work [3] proposed a slightly different analysis framework, *compliant vector analysis*, which enables tighter tardiness bounds for ordinary sporadic task systems scheduled under GEDF compared to the framework we employ. This new analysis framework, along with the proposed technique in this paper, can be applied to provide tighter tardiness bounds for scheduling SSS task systems as well.

The rest of this paper is organized as follows. In Sec. 2, we present the SSS task model. Then, in Sec. 3, we present our $O(m)$ analysis technique and the corresponding SRT schedulability test. In Sec. 4, we show that our derived test theoretically dominates prior tests. In Sec. 5, we present experimental results. We conclude in Sec. 6.

2 System Model

We consider the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n independent SSS tasks on $m \geq 1$ identical processors $M_1, M_2, ..., M_m$. Each task is released repeatedly, with each such invocation called a *job*. Jobs alternate between computation and suspension phases. We assume that each job of τ_i executes for at most e_i time units (across all of its execution phases) and suspends for at most s_i time units (across all of its suspension phases). We place no restrictions on how these phases interleave (a job can even begin or end with a suspension phase). The j^{th} job of τ_i, denoted $\tau_{i,j}$, is released at time $r_{i,j}$ and has a deadline at time $d_{i,j}$. Associated with each task τ_i are a period p_i, which specifies the minimum time between two consecutive job releases of τ_i, and a deadline d_i, which specifies the relative deadline of each such job, i.e., $d_{i,j} = r_{i,j} + d_i$. The utilization of a task τ_i is defined as $u_i = e_i/p_i$, and the utilization of the task system τ as $U_{sum} = \sum_{\tau_i \in \tau} u_i$. An SSS task system τ is said to be an *implicit-deadline* system if $d_i = p_i$ holds for each τ_i. Due to space constraints, we limit attention to implicit-deadline SSS task systems in this paper.

Successive jobs of the same task are required to execute in sequence. If a job $\tau_{i,j}$ completes at time t, then its *tardiness* is $max(0, t - d_{i,j})$. A task's tardiness is the maximum tardiness of any of its jobs. Note that, when a job of a task misses its deadline, the release time of the next job of that task is not altered. We require $e_i + s_i \leq p_i$, $u_i \leq 1$, and $U_{sum} \leq m$; otherwise, tardiness can grow unboundedly.

For simplicity, we henceforth assume that each job of any task τ_i executes for *exactly* e_i time units. As shown in [6], any tardiness bound derived for an SSS task system by considering only schedules meeting this assumption applies to other schedules as well.

Throughout the paper, we assume that time is integral, and for any task $\tau_i \in \tau$, each of $e_i \geq 1$, $s_i \geq 0$, $p_i \geq 1$, and $d_i \geq 1$ is a non-negative integer. Thus, a job that executes or suspends at time instant t executes or suspends during the entire time interval $[t, t + 1)$.

Under GEDF, released jobs are prioritized by their absolute deadlines. We assume that ties are broken by task ID (lower IDs are favored).

3 Schedulability Analysis

We now present our proposed new schedulability analysis for SRT SSS task systems. Our analysis draws inspiration from the seminal work of Devi [2], and follows the same general framework. We first describe the proof setup, then present our new $O(m)$ analysis technique, and finally derive a resulting schedulability test.

We focus on a given SSS task system τ. Let $\tau_{l,j}$ be a job of task τ_l in τ, $t_d = d_{l,j}$, and S be a GEDF schedule for τ with the following property.

(P) The tardiness of every job $\tau_{i,k}$, where $\tau_{i,k}$ has higher priority than $\tau_{l,j}$, is at most $x + e_i + s_i$ in S, where $x \geq 0$.

Our objective is to determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x+e_l+s_l$. This would by induction imply a tardiness of at most $x + e_i + s_i$ for all jobs of every task τ_i, where $\tau_i \in \tau$. We assume that $\tau_{l,j}$ finishes after t_d, for otherwise, its tardiness is trivially zero. The steps for determining the value for x are as follows.

1. Determine a lower bound on the amount of work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d, required for the tardiness of $\tau_{l,j}$ to exceed $x + e_l + s_l$. This is dealt with in Lemma 2 in Sec. 3.2.
2. Determine an upper bound on the work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d. This is dealt with in Lemmas 3 and 4 in Sec. 3.3.
3. Determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$, using the above lower and upper

bounds. This is dealt with in Theorem 1 in Sec. 3.4.

Definition 1. A task τ_i is *active* at time t if there exists a job $\tau_{i,v}$ such that $r_{i,v} \leq t < d_{i,v}$.

Definition 2. A job is considered to be *completed* if it has finished its last phase (be it suspension or computation). We let $f_{i,v}$ denote the completion time of job $\tau_{i,v}$. Job $\tau_{i,v}$ is *tardy* if it completes after its deadline.

Definition 3. Job $\tau_{i,v}$ is *pending* at time t if $r_{i,v} \leq t < f_{i,v}$. Job $\tau_{i,v}$ is *enabled* at t if $r_{i,v} \leq t < f_{i,v}$, and its predecessor (if any) has completed by t.

Definition 4. If job $\tau_{i,v}$ is enabled and not suspended at time t but does not execute at t, then it is *preempted* at t.

Definition 5. We categorize jobs based on the relationship between their priorities and those of $\tau_{l,j}$:

$$\mathbf{d} = \{\tau_{i,v} : (d_{i,v} < t_d) \vee (d_{i,v} = t_d \wedge i \leq l)\}.$$

Definition 6. For any given SSS task system τ, a *processor share* (PS) schedule is an ideal schedule where each task τ_i executes with a rate equal to u_i when it is active (which ensures that each of its jobs completes exactly at its deadline). *Note that suspensions are not considered in the PS schedule*. A valid PS schedule exists for τ if $U_{sum} \leq m$ holds.

By Def. 5, $\mathbf{d}$ is the set of jobs with deadlines at most t_d with priority at least that of $\tau_{l,j}$. These jobs do not execute beyond t_d in the PS schedule. Note that $\tau_{l,j}$ is in $\mathbf{d}$. Also note that jobs not in $\mathbf{d}$ have lower priority than those in $\mathbf{d}$ and thus do not affect the scheduling of jobs in $\mathbf{d}$. For simplicity, we will henceforth assume that jobs not in $\mathbf{d}$ do not execute in either the GEDF schedule S or the corresponding PS schedule. To avoid distracting "boundary cases," we also assume that the schedule being analyzed is prepended with a schedule in which no deadlines are missed that is long enough to ensure that all previously released jobs referenced in the proof exist.

Our schedulability test is obtained by comparing the allocations to $\mathbf{d}$ in the GEDF schedule S and the corresponding PS schedule, both on m processors, and quantifying the difference between the two. We analyze task allocations on a per-task basis. Let $A(\tau_{i,v}, t_1, t_2, S)$ denote the total allocation to job $\tau_{i,v}$ in S in $[t_1, t_2)$. Then, the total time allocated to all jobs of τ_i in $[t_1, t_2)$ in S is given by

$$A(\tau_i, t_1, t_2, S) = \sum_{v \geq 1} A(\tau_{i,v}, t_1, t_2, S).$$

Let *PS* denote the PS schedule that corresponds to the GEDF schedule S (i.e., the total allocation to any job of any task in *PS* is identical to the total allocation of the job in S).

The difference between the allocation to a job $\tau_{i,v}$ up to time t in *PS* and S, denoted *the lag of job $\tau_{i,v}$ at time t in schedule S*, is defined by

$$lag(\tau_{i,v}, t, S) = A(\tau_{i,v}, 0, t, PS) - A(\tau_{i,v}, 0, t, S).$$

Similarly, the difference between the allocation to a task τ_i up to time t in *PS* and S, denoted *the lag of task τ_i at time t in schedule S*, is defined by

$$\begin{aligned} lag(\tau_i, t, S) &= \sum_{v \geq 1} lag(\tau_{i,v}, t, S) \\ &= \sum_{v \geq 1} (A(\tau_{i,v}, 0, t, PS) - A(\tau_{i,v}, 0, t, S)). \quad (1) \end{aligned}$$

The concept of lag is important because, if lags remain bounded, then tardiness is bounded as well. The *LAG* for $\mathbf{d}$ at time t in schedule S is defined as

$$LAG(\mathbf{d}, t, S) = \sum_{\tau_i : \tau_{i,v} \in \mathbf{d}} lag(\tau_i, t, S). \quad (2)$$

Definition 7. A time instant t is *busy* (resp. *non-busy*) for a job set J if all (resp. not all) m processors execute jobs in J at t. A time interval is *busy* (resp. *non-busy*) for J if each instant within it is busy (resp. non-busy) for J. A time instant t is *busy on processor M_k* (resp. *non-busy on processor M_k*) for J if M_k executes (resp. does not execute) a job in J at t. A time interval is *busy on processor M_k* (resp. *non-busy on processor M_k*) for J if each instant within it is busy (resp., non-busy) on M_k for J.

The following claim follows from the definition of LAG.

Claim 1. *If $LAG(\boldsymbol{d}, t_2, S) > LAG(\boldsymbol{d}, t_1, S)$, where $t_2 > t_1$, then $[t_1, t_2)$ is non-busy for $\boldsymbol{d}$. In other words, LAG for $\boldsymbol{d}$ can increase only throughout a non-busy interval for $\boldsymbol{d}$.*

3.1 New $O(m)$ Analysis Technique

By Claim 1 and the discussion in Sec. 1, the pessimism of analyzing SSS task systems is due to the worst-case scenario where all SSS tasks might have enabled jobs that suspend at a time instant t, making t non-busy; this can result in non-busy intervals in which LAG for $\mathbf{d}$ increases. Specifically, the worst-case scenario happens when *at least m suspending tasks have enabled tardy jobs with deadlines at or before a non-busy time instant t where such jobs suspend at t.* (As seen in the analysis in Secs. 3.2 and 3.3, suspensions of non-tardy jobs are not problematic.)

Thus, our goal is to avoid such a worst case. The key idea behind our new technique is the following: *At any non-busy time t, if k processors ($1 \leq k \leq m$) are idle at t while at least k suspending tasks have enabled tardy jobs with deadlines at or before t that suspend simultaneously at t, then, by treating suspensions of k jobs of k such tasks to be computation at t, t becomes busy. Treating the suspensions of all such tasks to be computation is needlessly pessimistic.*

Let t_f denote the end time of the schedule S. Our new technique involves transforming the entire schedule S within $[0, t_f)$ from right to left (i.e., from time t_f to time 0) to obtain a new schedule $\overline{S}$ as described below. The goal of this transformation is to convert certain tardy jobs' suspensions into computation in non-busy time intervals to eliminate idleness

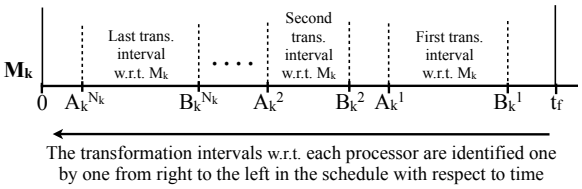


Figure 1: Transformation intervals with respect to M_k.

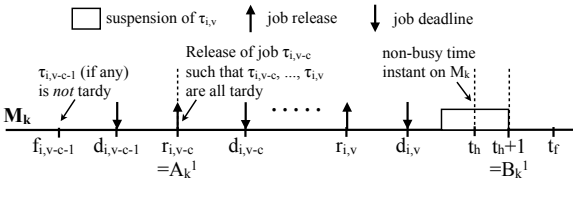


Figure 2: Defining t_h and $r_{i,v-c}$ in the transformation method.

as discussed above. For any job $\tau_{i,v}$, if its suspensions are converted into computation in a time interval $[t_1, t_2)$, then $\tau_{i,v}$ is considered to execute in $[t_1, t_2)$. We transform S to $\overline{S}$ by applying m *transformation steps*, where in the k^{th} step, the schedule is transformed with respect to processor M_k. Let $S^0 = S$ denote the original schedule, S^k denote the transformed schedule after performing the k^{th} transformation step, and $S^m = \overline{S}$ denote the final transformed schedule. The k^{th} transformation step works as follows.

Transformation method. By analyzing the schedule S^{k-1} on M_k, we first define $N_k \geq 0$ *transformation intervals* denoted $[A_k^1, B_k^1)$, $[A_k^2, B_k^2)$, ..., $[A_k^{N_k}, B_k^{N_k})$ ordered from right to left with respect to time, as illustrated in Fig. 1. These transformation intervals are the only intervals that are affected by the k^{th} transformation step. We identify these transformation intervals by moving from right to left with respect to time in the schedule S^{k-1} considering allocations on processor M_k. (An extended example illustrating the entire transformation method will be given later.)

Moving from time t_f to the left in S^{k-1}, let t_h denote the first encountered non-busy time instant on M_k where at least one task τ_i has an enabled job $\tau_{i,v}$ suspending at t_h where

$$d_{i,v} \leq t_h. \tag{3}$$

(If t_h does not exist, then we have $S^k = S^{k-1}$.) Let $v - c$ ($0 \leq c \leq v - 1$) denote the minimum job index of τ_i such that all jobs $\tau_{i,v-c}, \tau_{i,v-c+1}, ..., \tau_{i,v}$ are tardy, as illustrated in Fig. 2. Then, $[r_{i,v-c}, t_h + 1)$ is the first transformation interval with respect to M_k, i.e., $[A_k^1, B_k^1) = [r_{i,v-c}, t_h + 1)$.

To find the next transformation interval on M_k, further moving from A_k^1 to the left in S^{k-1}, find the next t_h, $\tau_{i,v}$, and $\tau_{i,v-c}$ applying the same definitions given above. If they exist, then the newly founded interval $[r_{i,v-c}, t_h + 1)$ is the second transformation interval $[A_k^2, B_k^2)$. Such a process continues, moving from right to the left in S^{k-1}, until time 0 is reached before any such t_h is found. If task τ_i is selected to define the transformation interval $[A_k^q, B_k^q)$ in the manner just described, then we say that τ_i is *associated* with this transformation interval; each transformation interval has exactly one associated task. (As seen below, when performing transformation steps after the k^{th} step, τ_i cannot be selected again for defining transformation intervals within $[A_k^q, B_k^q)$.) According to the way we identify transformation intervals as described above, the following property holds.

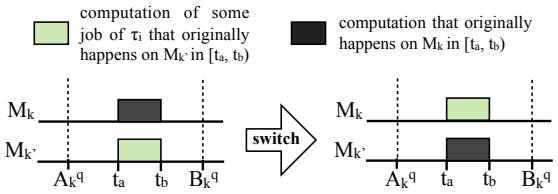


Figure 3: *Switch:* switch the computation of τ_i originally executed on $M_{k'}$ to M_k.

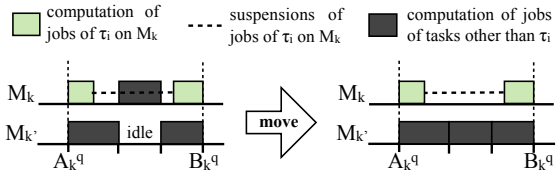


Figure 4: *Move:* move the computation of tasks other than τ_i from M_k to some idle processor $M_{k'}$.

(G1) Successive transformation intervals with respect to processor M_k do not overlap, i.e., $B_k^q \leq A_k^{q-1}$ holds where $2 \leq q \leq N_k$.

After identifying all transformation intervals with respect to M_k, we perform the following three operations on each of these N_k transformation intervals, starting with $[A_k^1, B_k^1)$. In the following, let $[A_k^q, B_k^q)$ ($1 \leq q \leq N_k$) denote the currently considered transformation interval and let τ_i be the associated task.

1. Switch: We assume that all computations of jobs of τ_i occurring within $[A_k^q, B_k^q)$ happen on M_k. This is achieved by switching any computation of τ_i in any interval $[t_a, t_b) \subseteq [A_k^q, B_k^q)$ originally executed on some processor $M_{k'}$ other than M_k with the computation (if any) occurring in $[t_a, t_b)$ on M_k, as illustrated in Fig. 3.

2. Move: Then for all intervals in $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute while some job of τ_i suspends, if any of such interval is non-busy (at least one processor is idle in this interval), then we also move the computation occurring within this interval on M_k to some processor $M_{k'}$ that is idle in the same interval, as illustrated in Fig. 4. This guarantees that all intervals in $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute are busy on all processors.

Due to the fact that all jobs of τ_i enabled in $[A_k^q, B_k^q)$ (i.e., $\tau_{i,v-c}, \tau_{i,v-c+1}, ..., \tau_{i,v}$)[1] are tardy, interval $[A_k^q, B_k^q)$ on M_k

[1]Note that, according to the way we select $v - c$, job $\tau_{i,v-c-1}$ is not enabled within $[A_k^q, B_k^q)$ because it is not tardy and thus completes at or before $d_{i,v-c-1} \leq r_{i,v-c} = A_k^q$.

consists of three types of subintervals: (*i*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are executing, (*ii*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are suspending (note that jobs of tasks other than τ_i may also execute on M_k in such subintervals; if this is the case, then the move operation ensures that any such subinterval is busy on all processors), and (*iii*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are preempted. Thus, within any non-busy interval on M_k in $[A_k^q, B_k^q)$, jobs of τ_i must be suspending (for otherwise this interval would be busy on M_k). Therefore, we perform the third transformation operation as follows.

3. Convert: Within all time intervals that are non-busy on M_k in $[A_k^q, B_k^q)$, convert the suspensions of all jobs of τ_i enabled within $[A_k^q, B_k^q)$ into computation, as illustrated in Fig. 5. This guarantees that M_k is busy within $[A_k^q, B_k^q)$. Since all such jobs belong to the associated task τ_i of $[A_k^q, B_k^q)$, by the definitions of A_k^q and B_k^q as described above, the following property holds.

> **(G2)** For any transformation interval $[A_k^q, B_k^q)$, jobs whose suspensions are converted into computation in this interval in the k^{th} transformation step have releases and deadlines within this interval.

When performing any later transformation step $k' > k$ (i.e., with respect to processor $M_{k'}$), τ_i clearly cannot be selected again for its suspensions to be converted into computation in idle intervals on $M_{k'}$ within $[A_k^q, B_k^q)$. Moreover, since all intervals within $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute are busy on all processors, any switch or move operation performed in later transformation steps does not change the fact that $[A_k^q, B_k^q)$ is busy on M_k in the final transformed schedule $\overline{S}$. Note that the above switch, move, and convert operations do not affect the start and completion times of any job.

After performing the switch, move, and convert operations on $[A_k^q, B_k^q)$ as described above, the transformation within $[A_k^q, B_k^q)$ is complete. We then consider the next transformation interval $[A_k^{q+1}, B_k^{q+1})$, and so on. The k^{th} transformation step is complete when all such transformation intervals have been considered, from which we obtain S^k.

Repeating this entire process, we similarly obtain S^{k+1}, S^{k+2}, ..., $S^m = \overline{S}$.

Analysis. The transformation method above ensures the following.

Claim 2. *At any non-busy time instant $t \in [0, t_f)$ in the transformed schedule $\overline{S}$, at most $m-1$ tasks can have enabled tardy jobs with deadlines at or before t.*

Proof. Suppose that $t \in [0, t_f)$ is non-busy in $\overline{S}$, and there are z idle processors at t. Assume that m or more tasks have enabled tardy jobs at t with deadlines at or before t. Then at least z such tasks have enabled tardy jobs suspending at t in order for t to be non-busy. However, our transformation method would have converted the suspension time at t of z such jobs into computation, which makes t busy on M_k, a contradiction. □

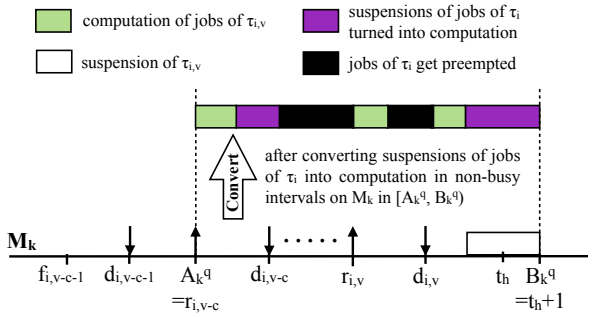


Figure 5: *Convert:* convert the suspensions of all jobs of τ_i that are enabled within $[A_k^q, B_k^q)$ (i.e., $\tau_{i,v-c}$, ..., $\tau_{i,v}$) into computation within all non-busy time intervals on M_k in $[A_k^q, B_k^q)$.

Example 1. Consider a two-processor task set τ that consists of three tasks: τ_1 = ((4(exec.), 2(susp.), 4(exec.)), 10(period)), and τ_2 = τ_3 = ((2(exec.), 6(susp.), 2(exec.)), 10(period)). Fig. 6(a) shows the original GEDF schedule S for the time interval [0,34). Assume $\tau_{3,3}$ is the analyzed job so that $t_d = 30$ and $t_f = 34$. By the transformation method, we first transform the schedule in $[0, 34)$ with respect to processor M_1 (i.e., the first transformation step). Moving from $t_f = 34$ to the left in S, the first idle time instant on M_1 is time 31. At time 31, two jobs $\tau_{2,3}$ and $\tau_{3,3}$ are suspending and both jobs satisfy condition (3) since $d_{2,3} = d_{3,3} = 30$. We arbitrarily choose τ_3 for this transformation step. Since job $\tau_{3,1}$ has the minimum job index of τ_3 such that all jobs $\tau_{3,1}, \tau_{3,2}, \tau_{3,3}$ are tardy, we use τ_3 for the transformation with respect to M_1 up to the release time of $\tau_{3,1}$, which is time 0. Thus, M_1 has only one transformation interval, i.e., $[0, 32)$. By the transformation method, we first perform the switch operation, which switches any computation of jobs of τ_3 in $[0, 32)$ that is not occurring on M_1 to M_1; this includes the computation in $[2, 4)$, $[10, 12)$, and $[22, 24)$. Accordingly, the computations that originally occur in these three intervals on M_1, which are due to jobs of τ_1, are switched to M_2. The resulting schedule after this switching is shown in Fig. 6(b). After this switching, we apply the move operation, which affects non-busy intervals in $[0, 32)$ in which jobs of τ_3 are suspending while jobs of tasks other than τ_3 are executing on M_1. For this example schedule, $[6, 8)$, $[16, 20)$, and $[26, 30)$ must be considered, and we move the computation of τ_1 in these three intervals from M_1 to M_2. (Note that since intervals $[4, 6)$ and $[30, 32)$ are non-busy on both processors, they are not considered.) The resulting schedule after this moving is shown in Fig. 6(c). Finally, within all non-busy intervals on M_1 in $[0, 32)$, which include $[4, 8)$, $[16, 20)$, and $[26, 32)$, we convert the suspensions of all enabled jobs of τ_3 in $[0, 32)$ into computation. We thus complete the first transformation step and obtain the schedule S^1, which is shown in Fig. 6(d). As seen in Fig. 6(d), after applying the transformation method with respect to M_1, M_1 is fully occupied in

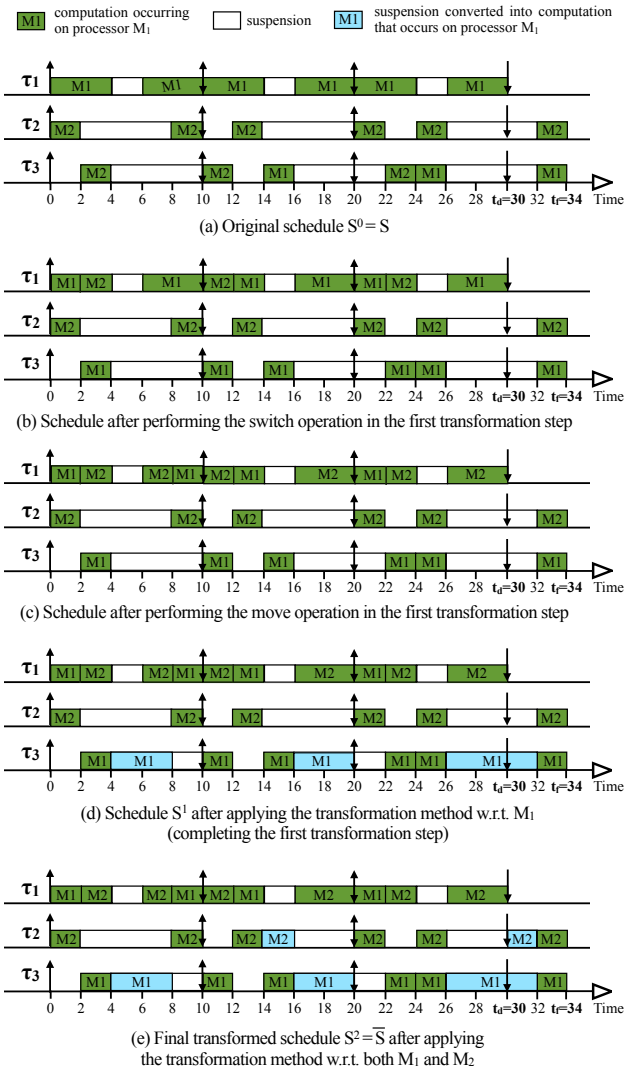


Figure 6: Example schedule transformation.

interval $[0, 32)$ by the computation of jobs of τ_3, suspensions of jobs of τ_3 that are converted into computation, the computations of jobs not belonging to τ_3 that preempt jobs of τ_3, and the computations of jobs not belonging to τ_3 that occur on M_1 while jobs of τ_3 are suspending. Next, we perform the second transformation step and transform S^1 in $[0, 34)$ with respect to M_2. This yields the final transformed schedule $S^2 = \overline{S}$, as shown in Fig. 6(e). Notice that M_2 is idle in $[4, 6)$ in $\overline{S}$ because jobs $\tau_{1,1}$ and $\tau_{2,1}$, which suspend in $[4, 6)$, have deadlines (at time 10) after time 6; thus, by the transformation method (see (3)) these jobs' suspensions are not turned into computation.

Definition 8. Let $\overline{u_i} = \dfrac{e_i + s_i}{p_i} = u_i + \dfrac{s_i}{p_i}$.

Note that in the above definition, $\overline{u_i} \leq 1$ holds for any task $\tau_i \in \tau$ because $e_i + s_i \leq p_i$, as discussed in Sec. 2.

Definition 9. The interval $[r_{i,j}, d_{i,j})$ is called the *job execution window* for job $\tau_{i,j}$.

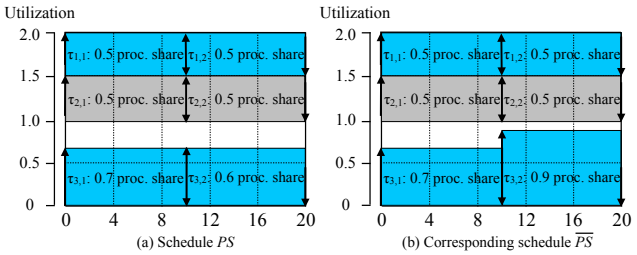


Figure 7: An example task system containing three tasks, τ_1 and τ_2 of utilization 0.5, and τ_3 of utilization 0.6. All three tasks have a period of 10 time units. (a) shows the PS schedule PS for this system where each task executes according to its utilization rate when it is active. Assume that in the transformed schedule $\overline{S}$ for this system, three time units of suspensions of $\tau_{3,2}$ are turned into computation, which causes the utilization of τ_3 to increase to 0.9 in $\tau_{3,2}$'s job execution window $[r_{3,2}, d_{3,2}) = [10, 20)$. (b) shows the corresponding schedule $\overline{PS}$ after this transformation. As seen, in $\overline{PS}$, task τ_3 executes with a rate of 0.9 in $[10, 20)$.

Defining $\overline{PS}$. Now we define the PS schedule $\overline{PS}$ corresponding to the transformed schedule $\overline{S}$. Without applying the transformation method, any task τ_i executes in a PS schedule with the rate u_i in any of its job execution windows (i.e., when τ_i is active). Thus, if τ_i is active throughout $[t_1, t_2)$ in PS, then $A(\tau_{i,j}, t_1, t_2, PS) = (t_2 - t_1)u_i$ holds.

On the other hand, after applying the transformation method, since we may convert a certain portion of the suspension time of any job of any task into computation, a task may have different utilizations within different job execution windows, but within the range of $[u_i, \overline{u_i}]$. The upper bound of this range is $\overline{u_i}$ because all suspensions of a job could be turned into computation. Thus, for any task τ_i that is active throughout $[t_1, t_2)$ in $\overline{PS}$, we have

$$(t_2 - t_1) \cdot u_i \leq A(\tau_{i,j}, t_1, t_2, \overline{PS}) \leq (t_2 - t_1) \cdot \overline{u_i}. \quad (4)$$

An example is given in Fig. 7.

Note that our analysis does not require $\overline{PS}$ to be specifically defined; rather, only (4) is needed in our analysis.

Definition 10. Let $v_i = \dfrac{s_i}{p_i}$ denote the *suspension ratio* for task $\tau_i \in \tau$. Let v^j denote the j^{th} maximum suspension ratio among tasks in τ.

In order for our analysis to be correct, we have to ensure that such a valid $\overline{PS}$ schedule exists. That is, we have to guarantee that the total utilization of τ is at most m at any time instant t in $\overline{PS}$. The following lemma gives a sufficient condition that can provide such a guarantee.

Lemma 1. *If $U_{sum} + \sum_{j=1}^{m} v^j \leq m$, then a valid PS schedule $\overline{PS}$ corresponding to $\overline{S}$ exists.*

Proof. By the transformation method, for any processor M_k, jobs whose suspensions are converted into computation with

respect to M_k do not have overlapping job execution windows. This is because such jobs either belong to the same task (in which case such jobs clearly do not have overlapping job execution windows), or belong to different tasks each of which is associated with a different transformation interval with respect to M_k. In the latter case, non-overlap follows by Properties (G1) and (G2) stated earlier. Since there are m processors and the jobs used for the transformation with respect to each processor do not have overlapping job execution windows, at any time t in $\overline{PS}$, there exist at most m jobs with overlapping job execution windows whose suspensions are converted into computation, which can increase total utilization by at most $\sum_{j=1}^{m} v^j$. This implies that at any time t in $\overline{PS}$, the total utilization of τ is at most $U_{sum} + \sum_{j=1}^{m} v^j$, which is at most m according to the claim statement. □

We next derive a lower bound (Sec. 3.2) and an upper bound (Sec. 3.3) on the pending work that can compete with $\tau_{l,j}$ after t_d in schedule $\overline{S}$, which is given by $LAG(\mathbf{d}, t_d, \overline{S})$, as $\mathbf{d}$ includes all jobs of higher priority than $\tau_{l,j}$.

3.2 Lower Bound

Lemma 2 below establishes the desired lower bound on $LAG(\mathbf{d}, t_d, \overline{S})$.

Lemma 2. *If the tardiness of $\tau_{l,j}$ exceeds $x + e_l + s_l$, then $LAG(\boldsymbol{d}, t_d, \overline{S}) > m \cdot x + e_l + s_l$.*

Proof. We prove the contrapositive: we assume that

$$LAG(\mathbf{d}, t_d, \overline{S}) \quad \leq \quad m \cdot x + e_l + s_l \tag{5}$$

holds and show that the tardiness of $\tau_{l,j}$ cannot exceed $x + e_l + s_l$. Let η_l be the amount of work $\tau_{l,j}$ performs by time t_d in $\overline{S}$. Note that by the transformation method, $0 \leq \eta_l < e_l + s_l$. Define y as follows.

$$y = x + \frac{\eta_l}{m} \tag{6}$$

We consider two cases.

Case 1. $[t_d, t_d + y)$ *is a busy interval for* $\mathbf{d}$. In this case, the amount of work completed in $[t_d, t_d + y)$ is exactly my. Hence, the amount of work pending at $t_d + y$ is at most $LAG(\mathbf{d}, t_d, \overline{S}) - my \overset{\{\text{by (5) and (6)}\}}{\leq} mx + e_l + s_l - mx - \eta_l = e_l + s_l - \eta_l$. This remaining work will be completed no later than $t_d + y + e_l + s_l - \eta_l \overset{\{\text{by (6)}\}}{=} t_d + x + \frac{\eta_l}{m} + e_l + s_l - \eta_l \leq t_d + x + e_l + s_l$. Since this remaining work includes the work due for $\tau_{l,j}$, $\tau_{l,j}$ thus completes by $t_d + x + e_l + s_l$.

Case 2. $[t_d, t_d + y)$ *is a non-busy interval for* $\mathbf{d}$. Let t_s be the earliest non-busy instant in $[t_d, t_d + y)$. If more than $m - 1$ tasks have enabled tardy jobs in $\mathbf{d}$ at t_s, then since all such enabled tardy jobs in $\mathbf{d}$ at t_s have deadlines at or before $t_d \leq t_s$, by Claim 2, t_s cannot be non-busy. Thus, at most $m-1$ tasks can have enabled tardy jobs in $\mathbf{d}$ at t_s. Moreover, since the number of tasks that have enabled jobs in $\mathbf{d}$ does not increase after t_d, we have

(Z) At most $m - 1$ tasks have enabled tardy jobs in $\mathbf{d}$ at or after t_s.

If $\tau_{l,j}$ completes by t_s, then $f_{l,j} \leq t_s < t_d + y \overset{\{\text{by (6)}\}}{=} t_d + x + \frac{\eta_l}{m} < t_d + x + e_l + s_l$. In the rest of the proof, assume that $\tau_{l,j}$ completes after t_s. Let t_p be the completion time of $\tau_{l,j}$'s predecessor (i.e., $\tau_{l,j-1}$). If $t_p \leq t_s$, then $\tau_{l,j}$ is enabled at t_s and will execute or suspend at t_s because t_s is non-busy. Furthermore, by (Z), $\tau_{l,j}$ is not preempted after t_s. Thus, by the definition of η_l and t_s, we have $f_{l,j} \leq t_s + e_l + s_l - \eta_l < t_d + y + e_l + s_l - \eta_l \overset{\{\text{by (6)}\}}{=} t_d + x + \frac{\eta_l}{m} + e_l + s_l - \eta_l \leq t_d + x + e_l + s_l$.

The remaining possiblity is that $t_p > t_s$. In this case, $\tau_{l,j}$ will begin its first phase at t_p and by (Z) finish by time $t_p + e_l + s_l$. By Property (P) (applied to $\tau_{l,j}$'s predecessor), $t_p \leq t_d - p_l + x + e_l + s_l \leq t_d + x$. Thus, the tardiness of $\tau_{l,j}$ is $f_{l,j} - t_d \leq t_p + e_l + s_l - t_d \leq x + e_l + s_l$. □

3.3 Upper Bound

In this section, we determine an upper bound on $LAG(\mathbf{d}, t_d, \overline{S})$.

Definition 11. Let t_n be the end of the latest non-busy interval for $\mathbf{d}$ before t_d, if any; otherwise, $t_n = 0$.

By the above definition and Claim 1, we have

$$LAG(\mathbf{d}, t_d, \overline{S}) \leq LAG(\mathbf{d}, t_n, \overline{S}). \tag{7}$$

Lemma 3. *For any task τ_i and $t \in [0, t_d]$, if τ_i has pending jobs at t in the schedule $\overline{S}$, then we have*

$$lag(\tau_i, t, \overline{S}) \leq \begin{cases} e_i + s_i & \text{if } d_{i,k} \geq t \\ \overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i & \text{if } d_{i,k} < t \end{cases}$$

where $d_{i,k}$ is the deadline of the earliest pending job of τ_i, $\tau_{i,k}$, at time t in $\overline{S}$. If such a job does not exist, then $lag(\tau_i, t, \overline{S}) \leq 0$.

Proof. If τ_i does not have a pending job at t in $\overline{S}$, then by Def. 3 and (1), $lag(\tau_i, t, \overline{S}) \leq 0$ holds. So assume such a job exists. Let γ_i be the amount of work $\tau_{i,k}$ performs before t. By the transformation method, $\gamma_i < e_i + s_i$ holds.

By the selection of $\tau_{i,k}$, we have $lag(\tau_i, t, \overline{S}) = \sum_{h \geq k} lag(\tau_{i,h}, t, \overline{S}) = \sum_{h \geq k} \big(A(\tau_{i,h}, 0, t, PS) - A(\tau_{i,h}, 0, t, \overline{S})\big)$. Given that no job executes before its release time, $A(\tau_{i,h}, 0, t, \overline{S}) = A(\tau_{i,h}, r_{i,h}, t, \overline{S})$. Thus,

$$\begin{aligned} lag(\tau_i, t, \overline{S}) \quad = \quad & A(\tau_{i,k}, r_{i,k}, t, \overline{PS}) - A(\tau_{i,k}, r_{i,k}, t, \overline{S}) \\ & + \sum_{h>k} \big(A(\tau_{i,h}, r_{i,h}, t, \overline{PS}) \\ & - A(\tau_{i,h}, r_{i,h}, t, \overline{S})\big). \end{aligned} \tag{8}$$

By the definition of $\overline{PS}$, (4), and Def. 8, $A(\tau_{i,k}, r_{i,k}, t, \overline{PS}) \leq e_i + s_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, \overline{PS})$

$\leq \overline{u_i} \cdot \max(0, t - d_{i,k})$. By the selection of $\tau_{i,k}$, $A(\tau_{i,k}, r_{i,k}, t, \overline{S}) = \gamma_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, \overline{S}) = 0$. By setting these values into (8), we have

$$lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i + \overline{u_i} \cdot \max(0, t - d_{i,k}). \quad (9)$$

There are two cases to consider.

Case 1. $d_{i,k} \geq t$. In this case, (9) implies $lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i \leq e_i + s_i$.

Case 2. $d_{i,k} < t$. In this case, because $t \leq t_d$ and $d_{l,j} = t_d$, $\tau_{i,k}$ is not the job $\tau_{l,j}$. Thus, by Property (P), $\tau_{i,k}$ has a tardiness of at most $x + e_i + s_i$. Since $\tau_{i,k}$ is the earliest pending job of τ_i at time t, the earliest possible completion time of $\tau_{i,k}$ is at $t + e_i - \gamma_i$ ($\tau_{i,k}$ may suspend for zero time at run-time). Thus, we have $t + e_i - \gamma_i \leq d_{i,k} + x + e_i + s_i$, which gives $t - d_{i,k} \leq x + \gamma_i + s_i$. Setting this value into (9), we have $lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i + \overline{u_i} \cdot (x + \gamma_i + s_i) \leq \overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i$. □

Definition 12. Let $\overline{U}_{m-1}$ be the sum of the $m-1$ largest $\overline{u_i}$ values among tasks in τ. Let $\overline{E}$ be the largest value of the expression $\sum_{\tau_i \in \tau}(e_i + s_i) + \sum_{\tau_i \in \psi} \overline{u_i} \cdot s_i$, where ψ denotes any set of $m-1$ tasks in τ.

Lemma 4 below upper bounds $LAG(\mathbf{d}, t_d, \overline{S})$.

Lemma 4. $LAG(\boldsymbol{d}, t_d, \overline{S}) \leq \overline{U}_{m-1} \cdot x + \overline{E}$.

Proof. By (7), we have $LAG(\mathbf{d}, t_d, \overline{S}) \leq LAG(\mathbf{d}, t_n, \overline{S})$. By summing individual task lags at t_n, we can bound $LAG(\mathbf{d}, t_n, \overline{S})$. If $t_n = 0$, then $LAG(\mathbf{d}, t_n, \overline{S}) = 0$, so assume $t_n > 0$.

Given that the instant $t_n - 1$ is non-busy, by Claim 2, at most $m-1$ tasks can have enabled tardy jobs at $t_n - 1$ with deadlines at or before $t_n - 1$. Let θ denote the set of such tasks. Therefore, we have $LAG(\mathbf{d}, t_d, \overline{S}) \overset{\{\text{by (7)}\}}{\leq} LAG(\mathbf{d}, t_n, \overline{S}) \overset{\{\text{by (2)}\}}{=} \sum_{\tau_i : \tau_{i,v} \in \mathbf{d}} lag(\tau_i, t_n, \overline{S}) \overset{\{\text{by Lemma 3}\}}{\leq} \sum_{\tau_i \in \theta}(\overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i) + \sum_{\tau_j \in \tau - \theta}(e_j + s_j) \overset{\{\text{by Def. 12}\}}{\leq} \overline{U}_{m-1} \cdot x + \overline{E}$. □

3.4 Determining x

Setting the upper bound on $LAG(\mathbf{d}, t_d, \overline{S})$ in Lemma 4 to be at most the lower bound in Lemma 2 will ensure that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. The resulting inequality can be used to determine a value for x. By Lemmas 2 and 4, this inequality is $m \cdot x + e_l + s_l \geq \overline{U}_{m-1} \cdot x + \overline{E}$. Solving for x, we have

$$x \geq \frac{\overline{E} - e_l - s_l}{m - \overline{U}_{m-1}}. \quad (10)$$

By Defs. 8 and 12, $\overline{U}_{m-1} < m$ clearly holds. Thus, if x equals the right-hand side of (10), then the tardiness of $\tau_{l,j}$ will not exceed $x + e_l + s_l$ in $\overline{S}$. A value for x that is independent of the parameters of τ_l can be obtained by replacing $-e_l - s_l$ with $max_l(-e_l - s_l)$. Moreover, in order for our analysis to be valid, the condition $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ as stated in Lemma 1 must hold.

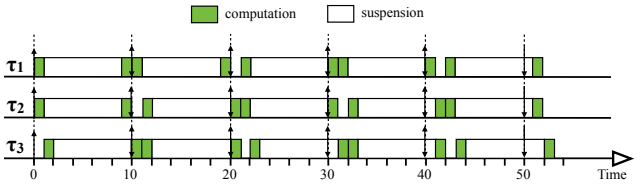


Figure 8: GEDF schedule of the counterexample.

Since the transformation method used to obtain $\overline{S}$ does not alter the tardiness of any job, the claim below follows.

Claim 3. *The tardiness of $\tau_{l,j}$ in the original schedule S is the same as that in the transformed schedule $\overline{S}$.*

By the above discussion, the theorem below follows.

Theorem 1. *With x as defined in (10), the tardiness of any task τ_l scheduled under GEDF is at most $x + e_l + s_l$, provided $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ where v^j is defined in Def. 10.*

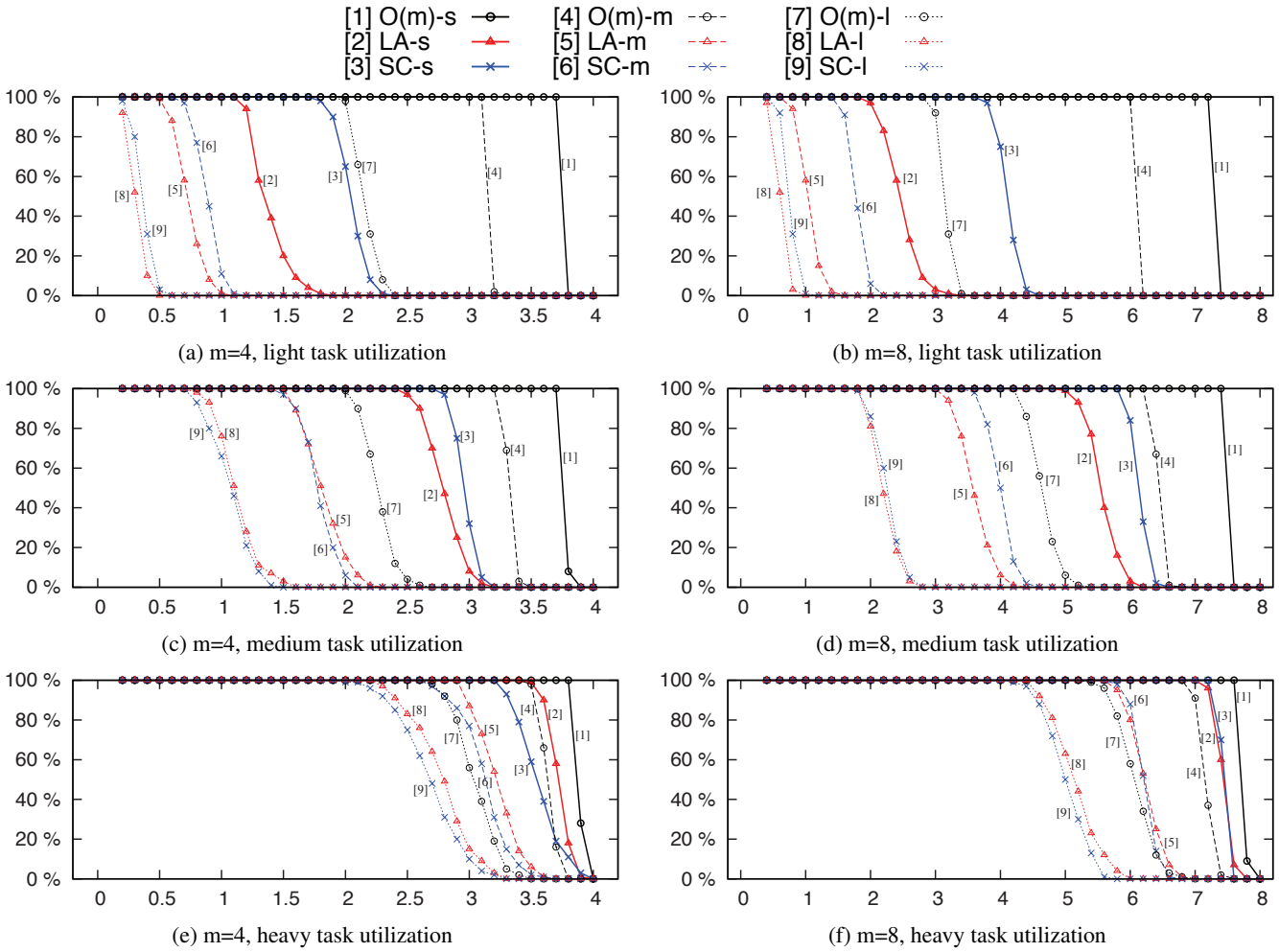
4 Theoretical Dominance over Prior Tests

We now show that our derived schedulability test theoretically dominates the two existing prior approaches [6, 7] with respect to schedulability. Moreover, we show via a counterexample that task systems that violate the utilization constraint stated in Theorem 1 may have unbounded tardiness.

When using the approach of treating all suspensions as computation [7], which transforms all SSS tasks into ordinary sporadic tasks, and then applying prior SRT schedulability analysis [2], the resulting utilization constraint is given by $U_{sum} + \sum_{i=1}^{n} \frac{s_i}{p_i} \leq m$. Clearly the constraint $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ from Theorem 1 is less restrictive than this prior approach. Moreover, the resulting utilization constraint[2] when using the technique presented in [6] is given by $U_{sum} < \left(1 - max_i\left(\frac{s_i}{e_i + s_i}\right)\right) \cdot m$. Since $\sum_{i=1}^{m} v^j \leq max_i\left(\frac{s_i}{e_i + s_i}\right) \cdot m$, our schedulability test is also less restrictive than this prior approach. Note that a major reason the prior approach in [6] causes significant capacity loss for some task systems is the fact that the term $max_i\left(\frac{s_i}{e_i + s_i}\right) \cdot m$ is not associated with any task period parameter. As a result, many task systems with even small utilizations may be deemed to be unschedulable.

Counterexample. Consider a two-processor task set τ that consists of three identical self-suspending tasks: $\tau_1 = \tau_2 =$

[2]Note that for generality, we consider here SSS task systems where all n tasks in the system are SSS tasks. As shown in [6], if many tasks are ordinary computational tasks (with no suspensions), then the schedulability test presented in [6] can be improved.



(a) m=4, light task utilization

(b) m=8, light task utilization

(c) m=4, medium task utilization

(d) m=8, medium task utilization

(e) m=4, heavy task utilization

(f) m=8, heavy task utilization

Figure 9: SRT schedulability results. In all six graphs, the x-axis denotes the task set utilization cap and the y-axis denotes the fraction of generated task sets that were schedulable with bounded deadline tardiness. In the first (respectively, second) column of graphs, $m = 4$ (respectively, $m = 8$) is assumed. In the first (respectively, second and third) row of graphs, light (respectively, medium and heavy) per-task utilizations are assumed. Each graph gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. As seen at the top of the figure, the label "**O(m)-s(m/l)**" indicates the approach of O(m) assuming short (moderate/long) suspensions. Similar "**LA**" and "**SC**" labels are used for LA and SC.

τ_3 = ((1(exec.), 8(susp.), 1(exec.)), 10(period)). For this system, $U_{sum} + \sum_{i=1}^{m} v^j = 0.6 + 1.6 = 2.2 > m = 2$, which violates the condition stated in Theorem 1. Fig. 8 shows the GEDF schedule of this task system. As seen, the tardiness of each task in this system grows with increasing job index.

5 Experiments

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the applicability of Theorem 1. Our goal is to examine how restrictive the derived schedulability test's utilization cap is, and how large the magnitude of tardiness is, and to compare it with prior methods [6, 7]. In the following, we denote our $O(m)$ schedulability test, the test presented in [6], and that presented in [7], as "O(m)," "LA," and "SC," respectively.

Experimental setup. In our experiments, task sets were generated as follows. Task periods were uniformly distributed over [50*ms*,200*ms*]. Task utilizations were distributed differently for each experiment using three uniform distributions. The ranges for the uniform distributions were [0.005,0.1] (light), [0.1,0.3] (medium), and [0.3,0.8] (heavy). Task execution costs were calculated from periods and utilizations. Suspension lengths of tasks were also distributed using three uniform distributions: $[0.005 \cdot (1 - u_i) \cdot p_i, 0.1 \cdot (1 - u_i) \cdot p_i]$ (suspensions are short), $[0.1 \cdot (1-u_i) \cdot p_i, 0.3 \cdot (1-u_i) \cdot p_i]$ (suspensions are moderate), and $[0.3 \cdot (1-u_i) \cdot p_i, 0.8 \cdot (1-u_i) \cdot p_i]$ (suspensions are long).[3] We varied the total system utilization U_{sum} within $\{0.1, 0.2, ..., m\}$. For each combination of task utilization distribution, suspension length distribution, and U_{sum}, 1,000 task sets were generated for systems with

[3]Note that any s_i is upper-bounded by $(1 - u_i) \cdot p_i$.

four and eight processors.[4] Each such task set was generated by creating tasks until total utilization exceeded the corresponding utilization cap, and by then reducing the last task's utilization so that the total utilization equalled the utilization cap. For each generated system, SRT schedulability (i.e., the ability to ensure bounded tardiness) was checked for O(m), LA, and SC.

Results. The obtained schedulability results are shown in Fig. 9 (the organization of which is explained in the figure's caption). Each curve plots the fraction of the generated task sets the corresponding approach successfully scheduled, as a function of total utilization. As seen, in all tested scenarios, O(m) significantly improves upon LA and SC by a substantial margin. For example, as seen in Fig. 9(a), when task utilizations are light and $m = 4$, O(m) can achieve 100% schedulability when U_{sum} equals 3.7, 3.1, and 2.1 when suspension lengths are short, moderate, and long, respectively, while LA and SC fail to do so when U_{sum} merely exceeds 1.8, 0.7, and 0.3, respectively. Note that when task utilizations are lighter, the improvement margin by O(m) over LA and SC increases. This is because in this case, the $s_i/(e_i+s_i)$ term that cause utilization loss in LA becomes large since e_i is small; similarly, for SC, more tasks are generated under light utilizations, which causes more utilization loss since all generated tasks' suspensions must be converted into computation. On the other hand, O(m)'s utilization loss is determined by the m largest suspension ratios, and thus is not similarly affected. In general, when suspension lengths are short and moderate, O(m) achieves little or no utilization loss in all scenarios. Even when suspension lengths become long, utilization loss under O(m) is still reasonable, especially when compared to the loss under LA and SC. Observe that all three approaches perform better when using heavier per-task utilization distributions. This is because when u_i is larger, s_i becomes smaller since $e_i + s_i \leq p_i$ must hold, which helps all three approaches yield smaller utilization loss.

In addition to schedulability, the magnitude of tardiness, as computed using the bound in Theorem 1, is of importance. Table 1 (the organization of which is explained in the table's caption) depicts the average of the computed bounds for each of the nine tested scenarios (three utilization distributions combined with three suspension length distributions) when $m = 4$ under O(m), SC, and LA, respectively. In all tested scenarios, O(m) provides reasonable predicted tardiness, which is comparable to SC and improves upon LA. For cases where suspensions are short, the predicted tardiness under O(m) is low. Moreover, as Fig. 9 implies, O(m) can often ensure bounded tardiness when SC or LA cannot. To conclude, our proposed analysis technique not only often guarantees schedulability with little or no utilization loss, but can provide such a guarantee with reasonable predicted tardiness.

[4] As noted earlier in Sec. 1, recent experimental work [1] suggests that global algorithms are viable only for small to moderate processor counts.

Table 1: Average tardiness bounds under O(m), SC, and LA when $m = 4$. The labels "**u-L**" /"**u-M**"/"**u-H**" indicate light/medium/heavy task utilizations, respectively. Within the row "**O(m)**," the three sub-rows "**n-O(m)**" /"**n-SC**"/"**n-LA**" represent the average tardiness bound achieved by O(m) as computed for all task sets that can be successfully scheduled under O(m)/SC/LA, respectively. The rows "**SC**"/"**LA**" represent the average tardiness bound achieved by SC/LA as computed for all task sets that can be successfully scheduled under SC/LA, respectively. All time units are in *ms*.

Method		Short susp.			Moderate susp.			Long susp.		
		u-L	u-M	U-H	u-L	u-M	U-H	u-L	u-M	U-H
O(m)	n-O(m)	140	125	191	301	173	286	667	286	377
	n-SC	59	83	178	82	113	247	181	167	289
	n-LA	22	34	158	50	64	197	106	172	203
SC		36	58	153	63	85	213	126	148	243
LA		97	144	316	149	287	375	231	626	892

6 Conclusion

We have presented a novel technique for analyzing SRT SSS task systems. The resulting schedulability test yields only an $O(m)$ suspension-related utilization loss, which theoretically dominates prior approaches [6, 7]. As demonstrated by experiments presented herein, our proposed test significantly improves upon prior methods with respect to schedulability, and is often able to guarantee schedulability with little or no utilization loss while providing low predicted tardiness.

In future work, we hope to extend the ideas of this paper to apply to HRT SSS task systems. In the HRT case, it may be similarly possible to transform the analyzed schedule on a processor-by-processor basis. However, transformation intervals cannot be defined on the basis of tardy jobs, as done in this paper.

References

[1] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *Proc. of the 31st RTSS*, pp. 14-24, 2010.

[2] U. Devi. Soft real-time scheduling on multiprocessors. In *Ph.D. Dissertation, UNC Chapel Hill*, 2006.

[3] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *Proc. of the 22nd Euromicro Conf. in Real-Time Sys.*, pp. 14-23, 2010.

[4] W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th RTSS*, pp. 277-287, 2007.

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

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

[7] J. Liu. *Real-time systems*. Prentice Hall, 2000.

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