A New Technique for Analyzing Soft Real-Time Self-Suspending Task Systems *

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

Abstract

We consider the problem of globally scheduling soft real-time sporadic self-suspending task systems on multiprocessors. Existing analysis methods are pessimistic, yielding $O(n)$ utilization loss where n is the number of tasks in the system. Unless the number of tasks is small and suspension delays are short, such methods entail significant capacity loss. We identify the fundamental sources that cause pessimism in existing methods, and propose a new analysis technique that entails only $O(m)$ suspension-related utilization loss, where m is the number of processors.

1 Introduction

In many real-time systems, suspension delays may occur when tasks interact with external devices such as I/O devices. Unfortunately, schedulability in real-time systems is negatively impacted by such delays if deadline misses cannot be tolerated [4]. In this paper, we consider whether, on multiprocessor platforms, such negative impacts can be ameliorated if task deadlines are soft. Our focus on multiprocessors is motivated by the advent of multicore platforms. There is currently great interest in providing operating-system support to enable real-time workloads to be hosted on such platforms. Many such workloads can be expected to include self-suspending tasks. Moreover, in many settings, such workloads can be expected to have soft timing constraints. The soft timing constraint considered in this paper pertains to implicit-deadline sporadic task systems and requires that deadline tardiness be bounded.

Two analysis approaches can be applied to analyze soft real-time (SRT) sporadic self-suspending (SSS) task systems on multiprocessors. Perhaps the most commonly used is the *suspension-oblivious* approach [3], which simply integrates suspensions into per-task worst-case-execution-time requirements. However, this approach clearly yields $O(n)$ utilization loss where n is the number of self-suspending tasks in the system. The alternative is to explicitly consider suspensions in the task model and the corresponding schedulability analysis; this is known as *suspension-aware* analysis. Although suspension-aware analysis can improve schedulability in many cases compared to the suspension-oblivious approach, existing suspension-aware analysis [2] still entails significant utilization loss.

To analyze SSS task systems more efficiently, we propose a new suspension-aware analysis technique that yields $O(m)$ suspension-related utilization loss, where m is the number of processors. Our technique is derived by identifying and then eliminating the fundamental sources that cause pessimism (i.e., $O(n)$ utilization loss) in previous analysis. Specifically, we derive a schedulability test under the proposed technique showing that any given SSS task system can be supported under global-earliest-deadline-first (GEDF) with bounded tardiness if $U_{sum} + \sum_{j=1}^{m} v^j \leq m$ holds, where U_{sum} is the total system utilization and v^j is the j^{th} maximum ratio of a task's suspension time over its period among tasks in the system.

The rest of this paper is organized as follows. In Sec. 2, we present the SSS task model. Then, in Sec. 3, we identify the fundamental sources causing pessimism in prior analysis and present our new analysis technique and a resulting schedulability test. We conclude in Sec. 4.

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 $\{M_1, M_2, ..., M_m\}$. We assume $n > m$; otherwise we can simply assign each task to one processor. 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_l executes for at most e_l time units (across all of its execution phases) and suspends for at most s_l 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_l, denoted $\tau_{l,j}$, is released at time $r_{l,j}$ and has a deadline at time $d_{l,j}$. Associated with each task τ_l is a period p_l, which specifies both the minimum time between two consecutive job releases of τ_l and the relative deadline of each such job, i.e., $d_{l,j} = r_{l,j} + p_l$. The utilization of a task τ_l is defined as $u_l = e_l/p_l$, 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.[1] In this

*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.

[1] τ is said to be a *constrained-deadline* system if, for each task $\tau_i \in \tau$, $d_i \leq p_i$, and an *arbitrary-deadline* system if, for each τ_i, the relation

paper, we consider implicit-deadline SSS task systems.

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 that $e_i + s_i \leq p_i$ and $u_i \leq 1$ hold for any task $\tau_i \in \tau$, and that $U_{sum} \leq m$ holds for τ; otherwise, tardiness can grow unboundedly.

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

3 An $O(m)$ Analysis Technique

In this section, we present our proposed $O(m)$ analysis technique and a resulting schedulability test for SRT SSS task systems. We first provide brief summaries of existing analysis approaches and highlight sources of pessimism in them. Efforts to overcome such pessimism will drive the design of the proposed new technique.

There are two existing approaches for dealing with globally-scheduled SRT multiprocessor SSS task systems: the suspension-oblivious approach, denoted SC, which converts all suspensions to computation [3], and a suspension-aware analysis approach presented by Liu and Anderson in [2], denoted LA.

Overview of the SC approach. The SC approach converts all suspensions into computation. After transforming all SSS tasks into ordinary sporadic tasks with only computation, prior SRT schedulability analysis [1] can be applied, resulting a utilization constraint of $U_{sum} + \sum_{i=1}^{n} \frac{s_i}{p_i} \leq m$.

Overview of the LA approach. The LA test is built around the following general strategy, first introduced by Devi and Anderson [1]. First, let $\tau_{l,j}$ be a job of a task τ_l in τ, and S be a GEDF schedule for τ with the following property: the tardiness of every job $\tau_{i,k}$ with priority greater than $\tau_{l,j}$ is at most $x + e_i + s_i$, where $x \geq 0$. Then, determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. This by induction implies a tardiness of at most $x + e_i + s_i$ for all jobs of every task τ_i in τ. The smallest x is determined by computing an upper and a lower bound on the pending work at $d_{l,j}$ for tasks in τ that can compete with $\tau_{l,j}$ after its deadline $d_{l,j}$. Next, we briefly summarize the process of obtaining such upper and lower bounds, and then identify sources causing pessimism in them.

The upper and lower bounds are obtained by comparing the allocations to jobs with priority at least that of $\tau_{l,j}$ in S and a processor share (PS) schedule, both on m processors, and quantifying the difference between the two. The PS schedule is an ideal schedule where each job of each task in τ completes exactly at its deadline. In the PS schedule, each task τ_i executes with a rate equal to u_i in any job execution window $[r_{i,j}, d_{i,j})$, which ensures that each job $\tau_{i,j}$ 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.

The upper bound on the pending work at $d_{l,j}$ can be obtained by bounding the pending work at time t_n, where t_n is defined to be the end of the latest non-busy interval (i.e., at least one processor is idle at any instant within this interval). This is because the amount of pending work (in comparison to the PS schedule) cannot increase throughout a busy interval (as all processors are busy at any instant within this interval). To bound the pending work at t_n, we have to bound the number of tasks that have enabled tardy jobs at t_n.[2] For ordinary task systems with no self-suspensions, the number of such tasks can be upper bounded by $m-1$, for otherwise t_n would be busy. For SSS task systems, however, all n tasks can have enabled tardy jobs suspending at t_n and t_n can still be non-busy. Since such a worst-case scenario may happen and thus must be considered in the analysis, significant pessimism is incurred in the obtained upper bound.

In lower-bounding the pending work at $d_{l,j}$, we need to bound the least amount of the pending work that executes within $[d_{l,j}, f_{l,j})$, where $f_{l,j}$ is the completion time of our analyzed job $\tau_{l,j}$. For ordinary task systems with no self-suspensions, such a bound is straightforward to obtain because within $[d_{l,j}, f_{l,j})$[3] a non-busy time instant could exist if and only if there are fewer than m tasks that have enabled jobs waiting for execution after $d_{l,j}$. For SSS task systems, unfortunately, idle intervals could exist within $[d_{l,j}, f_{l,j})$ due to suspensions even if at least m tasks have enabled tardy jobs. Thus, to lower bound the pending work, we need to upper bound the idleness that could possibly exist within $[t_{l,j}, f_{l,j})$. The worst-case scenario as mentioned above, where all tasks have multiple tardy jobs suspending simultaneously within $[t_{l,j}, f_{l,j})$, is the main source causing pessimism in the obtained lower bound.

The fundamental cause of pessimism in prior analysis. By the above discussion, we can identify the fundamental source causing pessimism in prior analysis, which is the following worst-case scenario: *all n self-suspending tasks have tardy jobs that suspend at some time t simultaneously, thus causing t to be non-busy; this creates idleness that results in pessimism in the analysis*.

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 tardy jobs at any non-busy time instant. *However, converting all n tasks' suspensions to computation is clearly*

between d_i and p_i is not constrained (e.g., $d_i > p_i$ is possible).

[2] Job $\tau_{i,v}$ is *enabled* at t if $r_{i,v} \leq t$, $\tau_{i,v}$ has not completed by t, and its predecessor job $\tau_{i,v-1}$ (if any) has completed by t. Job $\tau_{i,v}$ is *tardy* at t if $d_{i,v} < t$.

[3] Note that all jobs considered here have deadlines no later than $d_{l,j}$ since their priorities are at least that of $\tau_{l,j}$.

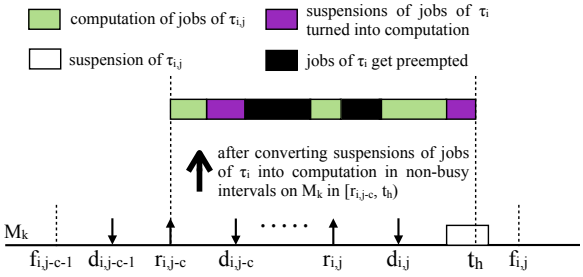

Figure 1: The transformation method.

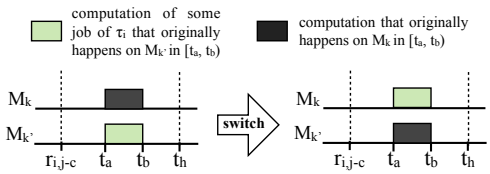

Figure 2: Switching the computation of τ_i originally executed on $M_{k'}$ to M_k.

overkill when attempting to avoid the worst-case scenario; rather, converting at most m tasks' suspensions to 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.

New Analysis Technique. We now sketch the new $O(m)$ analysis technique. Motivated by the above discussion, 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 that suspend simultaneously at t, then, by converting suspensions of k jobs of k such tasks into computation at t, t becomes busy. Converting the suspensions of all such tasks into computation is clearly unnecessary and pessimistic.*

Similar to [2], our analysis draws inspiration from the seminal work of Devi and Anderson [1], and follows the same general framework (which has been described earlier). Due to space constraints, we cannot provide every detail concerning the derivations of the upper and lower bounds. Instead, we focus on explaining how the proposed analysis technique eliminates the worst-case scenario and thus leads to a schedulability test with only $O(m)$ suspension-related utilization loss.

As described earlier, we apply the same proof setup to define our analyzed job $\tau_{l,j}$ and the GEDF schedule S. The part of the schedule S that needs to be analyzed is $[0, f_{l,j})$. We transform this part of the schedule from right to left (i.e., from time $f_{l,j}$ to time 0) to obtain a new schedule $\overline{S}$ as described next. The goal of this transformation is to convert certain tardy jobs' suspensions into computation in non-busy time intervals to eliminate idleness as discussed above. For any job $\tau_{i,k}$, if its suspensions are converted into computation in a time interval $[t_1, t_2)$, then $\tau_{i,k}$ is considered to execute in $[t_1, t_2)$. We transform S to $\overline{S}$ by applying the following

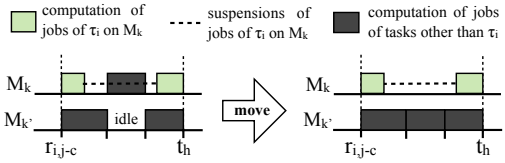

Figure 3: Moving the computation of tasks other than τ_i from M_k to some idle processor $M_{k'}$.

transformation method to each processor in turn (ordered by processor ID). In the following, M_k denotes the current considered processor (initially M_1). For simplicity, we use "S" to denote the updated schedule after each intermediate transformation step (the final transformed schedule $\overline{S}$ is obtained after the whole transformation process completes).

Transformation method. Moving from $f_{l,j}$ to the left in S on M_k, 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 $\tau_{i,j}$ suspending at t_h where

$$d_{i,j} < t_h. \tag{1}$$

Let $j-c$ ($0 \leq c \leq j-1$) denote the minimum job index of τ_i such that all jobs $\{\tau_{i,j-c}, \tau_{i,j-c+1}, ..., \tau_{i,j}\}$ are tardy, as illustrated in Fig. 1. We assume that all the computation and suspensions of jobs of τ_i occurring within $[r_{i,j-c}, t_h)$ happen on M_k. This can be achieved by switching any computation of τ_i in some interval $[t_a, t_b) \in [r_{i,j-c}, t_h)$ 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. 2, which is valid from an analysis point of view. Then for all intervals in $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute while some job of τ_i suspends, if any of such intervals 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. 3. This step guarantees that all intervals in $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute are busy on all processors. (Note that after performing the above switching and moving steps, the start and the completion times of jobs remain unchanged.) Due to the fact that all jobs of τ_i enabled in $[r_{i,j-c}, t_h)$ are tardy, interval $[r_{i,j-c}, t_h)$ on M_k consists of three types of subintervals: (*i*) those in which jobs of τ_i are executing, (*ii*) those in which jobs of τ_i 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 note that any such subinterval is busy on M_k), and (*iii*) those in which jobs of τ_i are preempted. Thus, within any non-busy interval on M_k in $[r_{i,j-c}, t_h)$, jobs of τ_i must be suspending (for otherwise this interval would be busy on M_k). Therefore, within all non-busy time intervals on M_k in $[r_{i,j-c}, t_k)$, we convert the suspensions of all jobs of τ_i that are enabled within $[r_{i,j}, t_h)$ (i.e., $\{\tau_{i,j-c}, \tau_{j-c+1}, ..., \tau_{i,j}\}$) into computation, as illustrated in Fig. 1. This transformation guarantees that M_k is busy within $[r_{i,j-c}, t_k)$. Note that when applying the rule on the next processor M_{k+1} (if any),

τ_i clearly cannot be chosen again for the same transformation process (i.e., converting suspensions into computation in idle intervals) within $[r_{i,j-c}, t_h)$. Moreover, since all intervals within $[r_{i,j-c}, t_h)$ on M_k where jobs not belonging to τ_i execute are busy on all processors, any later switch or move does not change the fact that $[r_{i,j-c}, t_h)$ is busy on M_k in the final transformed schedule $\overline{S}$. Next, further moving from $r_{l,j-c}$ to the left in S on M_k, find the next t_h, $\tau_{i,j}$, and $\tau_{i,j-c}$ following the same definitions given above, transform the schedule in the newly defined interval $[r_{i,j-c}, t_h)$ on M_k using the same approach. This process is repeatedly performed on M_k until $t_h = 0$. As mentioned earlier, this process will be applied on each processor in turn, after which we obtain the transformed schedule $\overline{S}$.

Analysis. By transforming S into $\overline{S}$ according to the above rule, we are able to eliminate the worst-case scenario where at least m tasks have enabled tardy jobs suspending at the same non-busy instant, as formerly presented by the following claim.

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

Proof. For any non-busy time instant $t_a \in [0, f_{l,j})$ in $\overline{S}$, there is at least one processor that is idle at t. Let M_k denote such a processor. Assume more than $m-1$ tasks have enabled jobs at t_a with deadlines before t_a. If m such jobs execute at t_a, then t_a would be busy. Thus, at most $m-1$ such jobs execute and at least one such job is suspending at t_a. Since t_a is non-busy on M_k, by our transformation method, one of the tasks that has an enabled job suspending at t_a with a deadline before t_a would be chosen at t_a such that the suspension of the enabled job of this task at t_a is converted to computation at t_a,[4] which makes t_a busy on M_k, a contradiction. □

Our analysis proceeds by comparing the allocations to jobs with priority at least that of $\tau_{l,j}$ in $\overline{S}$ and the corresponding PS schedule $\overline{PS}$ after the transformation.[5] A crucial step for our analysis to be valid is to show that such a $\overline{PS}$ exists. That is, we need to guarantee that at any time t in $\overline{PS}$, the total utilization of τ is at most m. The following claim provides a necessary condition that can provide such a guarantee.

Claim 2. *If $U_{sum} + \sum_{j=1}^{m} v^j \leq m$, then after the transformation, a PS schedule $\overline{PS}$ exists.*

Proof. Consider any processor M_k. Moving from right to left in $[0, f_{l,j})$ on M_k, whenever we first choose a task τ_i for the transformation process, we use tardy jobs of this same task until $r_{i,j-c}$ (which is the earliest job release time among all tardy jobs of τ_i used in one transformation step). Clearly all these jobs have non-overlapping periods since they belong to the same task. Next, moving further left from $r_{i,j-c}$ in the schedule on M_k to a new time t_h (as defined in the transformation method), if we use some task τ_j other than τ_i for the next transformation step on M_k, then the enabled job of τ_j at t_h must satisfy (1). Since this new t_h occurs before $r_{i,j-c}$, the enabled job of τ_j at t_h must have a deadline before $r_{i,j-c}$. This implies that jobs of τ_j whose suspensions are converted into computation have non-overlapping periods with those jobs of τ_i used for the transformation with respect to M_k. By applying the same reasoning to the rest of the schedule $\overline{S}$ on M_k, it follows that all jobs whose suspensions are converted into computation on M_k do not have overlapping periods. This same reasoning can be applied to all other processors. Since there are m processors and the jobs used for the transformation on each processor do not have overlapping periods, at any time t in $\overline{PS}$, there exist at most m jobs with overlapping periods 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. □

In order to correctly apply the transformation technique as described above, the expense we have to pay is the potential utilization loss of at most $\sum_{j=1}^{m} v^j$ due to the conversion of any m tasks' suspensions into computation. Thus, any SRT SSS task system that can accommodate this expense can be proved to be GEDF-schedulable, which is formerly described by the following schedulability test.

Theorem 1. *Any SRT SSS task system τ is GEDF-schedulable with bound tardiness on m processors if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$.*

4 Summary

In this paper, we presented a new multiprocessor schedulability analysis technique for globally-scheduled SRT SSS task systems. By identifying and eliminating the sources causing pessimism in prior analysis, our proposed analysis technique achieves a much improved schedulability test with only $O(m)$ suspension-related utilization loss. Given that m is often small in practice (typically two, four, or eight cores per chip), this technique is significant and is applicable to real systems.

References

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

[2] 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.

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

[4] 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.

[4] If there are h processors that idle at t_a, then at least h such tasks have enabled jobs suspending at t_a, and hence each processor is guaranteed to have one task available at t_a for the transformation

[5] Note that in $\overline{PS}$, any job of any task still completes exactly at its deadline. To ensure this, each task τ_i executes with a rate between u_i and $u_i + \overline{s}_{i,j}/p_i$ in any job execution window $[r_{i,j}, d_{i,j})$, where $\overline{s}_{i,j} \leq s_i$ is the amount of suspension time of job $\tau_{i,j}$ that is converted into computation in $\overline{S}$.