Improving the Schedulability of Sporadic Self-Suspending Soft Real-Time Multiprocessor Task Systems *

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

Abstract

In work on globally-scheduled soft real-time multiprocessor systems, analysis has been presented for dealing with self-suspensions, but this analysis can be pessimistic. In this paper, we present an approach that is designed to improve the schedulability of such systems. In experimental results that are presented, the proposed approach significantly improved schedulability in most considered scenarios.

1 Introduction

In many real-time systems, tasks self-suspend due to interactions with external devices such as disks and network cards. The suspension delays introduced by such behaviors can quite negatively impact schedulability if deadline misses cannot be tolerated [13]. The most common way to deal with such suspensions in analysis is to treat them as computation. However, if suspension times can be long, then this can result in severe system utilization loss. In recent work [8], we showed that such negative schedulability impacts can be ameliorated on multiprocessor platforms if task deadlines are soft. However, the resulting analysis still sometimes gives more pessimistic schedulability results than the simple approach of treating all suspensions as computation. In this paper, we present techniques that allow the analysis in [8] to be applied with even less pessimism.

Our focus on multiprocessors is motivated by the growing prevalence 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. 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. Under global scheduling, tasks are scheduled from a single run queue and may migrate across processors. A variety of global-scheduling approaches are capable of ensuring bounded tardiness in ordinary sporadic systems (without self-suspending tasks), as long as the system is not over-utilized [5]. (The same is not true of partitioning schemes, which statically assign tasks to processors.) In this paper, we focus specifically on two global scheduling algorithms that are capable of ensuring bounded tardiness for ordinary (suspension-less) sporadic systems with no utilization loss [3,6], namely, the *global earliest-deadline-first* (GEDF) algorithm and the *global first-in first-out* (GFIFO) algorithm.

In the prior work cited above [8], we extended previous work on tardiness under GEDF and GFIFO to show that, in fully-preemptive sporadic systems, bounded tardiness can be ensured even if tasks self-suspend. Specifically it is shown in [8] that tardiness in such a system is bounded provided

$$U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot m, \tag{1}$$

where U^s_{sum} is the total utilization of all self-suspending tasks, c is the number of computational tasks (which do not self-suspend), m is the number of processors, U^c_L is the sum of the $\min(m-1, c)$ largest computational task utilizations, and ξ_{max} is a parameter ranging over $[0, 1]$ called the *maximum suspension ratio*, which is defined to be the maximum value among all tasks' suspension ratios. For any task T_i, its suspension ratio, denoted ξ_i, is defined to be $\xi_i = \frac{s_i}{s_i + e_i}$, where s_i is T_i's suspension length and e_i is its execution cost. Significant utilization loss may occur when using (1) if ξ_{max} is large. Unfortunately, it is unavoidable that many self-suspending task systems will have large ξ_{max} values. For example, consider a task system with three tasks scheduled on two processors: T_1 has an execution cost of 5, a suspension length of 5, and a period of 10, T_2 has an execution cost of 2, a suspension length of 0, and a period of 8, and T_3 has an execution cost of 2, a suspension length of 2, and a period of 8. For this system,

*Work supported by AT&T, IBM, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

$U^s_{sum} = u_1 + u_3 = \frac{5}{10} + \frac{2}{8} = 0.75, U_L = u_2 = \frac{2}{8} = 0.25,$ $\xi_{max} = \xi_1 = \frac{5}{5+5} = 0.5$. Although the total utilization of this task system is only half of the overall processor capacity, it is not schedulable using the prior analysis since it violates the utilization constraint in (1) (since $U^s_{sum} + U_L = 1 = (1 - \xi_{max}) \cdot m$). Notice that the main reason for the violation is a large value of ξ_{max}. In this paper, we propose an approach that relaxes the utilization constraint in (1).

Related work. As noted earlier, the most common approach for dealing with self-suspensions is to simply treat all suspensions as computation. From [3, 6], tardiness is bounded under a pure computational task system (no suspensions) provided $U_{sum} \leq m$ where U_{sum} is the total utilization. A downside of treating all suspensions as computation is that this causes utilizations to be higher, which in many cases may cause total utilization to exceed m.

In work pertaining to uniprocessors (and by extension multiprocessors scheduled via partitioning), several schedulability tests have been presented for analyzing tasks with self-suspensions. These include a utilization-based test for EDF [2] and response-time-bound tests for fixed-priority systems [4, 9–11] and EDF-scheduled systems [11]. On a more negative note, Ridouard et. al [14] have shown that the feasibility problem for hard real-time, independent tasks with self-suspensions is NP-hard in the strong sense. Ridouard and Richard [13] have also shown that no optimal on-line algorithm exists for task systems with self-suspensions. In fixed-priority systems, scheduling penalties associated with self-suspensions can be lessened by using a technique called the *period enforcer* [12], which forces suspensions to occur more predictably.

In recent work, we showed that any task system with self-suspensions, pipelines, and non-preemptive sections can be transformed for analysis purposes into a system with only self-suspensions [7]. The transformation process treats delays caused by pipeline-based precedence constraints and non-preemptivity as self-suspension delays. It follows from this work that, improving the schedulability analysis of self-suspending task systems can also result in improved analysis for systems with other non-trivial behaviors.

Contributions. By observing that the utilization loss seen in (1) is mainly caused by a large value of ξ_{max}, our goal is to decrease the value of this parameter. We show that this can be done by treating *partial* suspensions as computation. That is, we consider intermediate choices between the two currently-available extremes of treating *all* (as is commonly done) or *no* (using the analysis in [8]) suspensions as computation. This approach is often able to decrease ξ_{max} at the cost of at most a slight increase in the left side of (1). We present both a linear programming solution for determining how much suspension time to be treated as computation as well as an optimal algorithm that runs in $O((N^s)^2)$ time, where N^s is the number of self-suspending tasks. We analyze the schedulability improvement brought by the proposed approach via an experimental study involving randomly-generated task systems. In all scenarios considered in this study, this approach was able to improve schedulability, and in most scenarios, by a substantial margin.

Organization. The rest of this paper is organized as follows. Sec. 2 presents needed definitions. In Sec. 3, the proposed approach is presented in detail. Sec. 4 evaluates the approach via an experimental study. Sec. 5 concludes.

2 Preliminaries

We consider the problem of scheduling a set $\tau = \{T_1, ..., T_n\}$ of n independent sporadic tasks on $m \geq 2$ identical processors. 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 T_l executes for at most e_l time units (across all of its computation phases) and suspends for at most s_l time units (across all of its suspension phases). (All time values considered in this paper are assumed to be real numbers.) 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 T_l, denoted $T_{l,j}$, is released at time $r_{l,j}$ and has a deadline at time $d_{l,j}$. Associated with each task T_l is a period p_l, which specifies both the minimum time between two consecutive job releases of T_l and the relative deadline of each such job, i.e., $d_{l,j} = r_{l,j} + p_l$. The utilization of a task T_l is defined as $u_l = e_l/p_l$, and the utilization of the task system τ as $U_{sum} = \sum_{T_i \in \tau} u_i$. Given that a task may have multiple computation and suspension phases, we sometimes use $T_l(ea, sb, ec, sd, ..., p)$ to denote the exact sequence of T_l's phases. For example, $T_1(e2, s1, e3, 10)$ denotes that task T_1 with a period of 10 time units first executes for up to 2 time units, then suspends for up to 1 time unit, and finally executes for up to 3 time units.

A common case for real-time workloads is that both self-suspending tasks and computational tasks (which do not suspend) co-exist. To reflect this, we let U^s_{sum} denote the total utilization of all self-suspending tasks, and U^c_{sum} denote the total utilization of all computational tasks.

Successive jobs of the same task are required to execute in sequence. If a job $T_{l,j}$ completes at time t, then its *response time* is $t - r_{l,j}$ and its *tardiness* is $max(0, t - d_{l,j})$. A task's tardiness is the maximum tardiness of any of its jobs. A task system is deemed to be *schedulable* if all tasks within it have bounded tardiness. Note that, when a job of a task

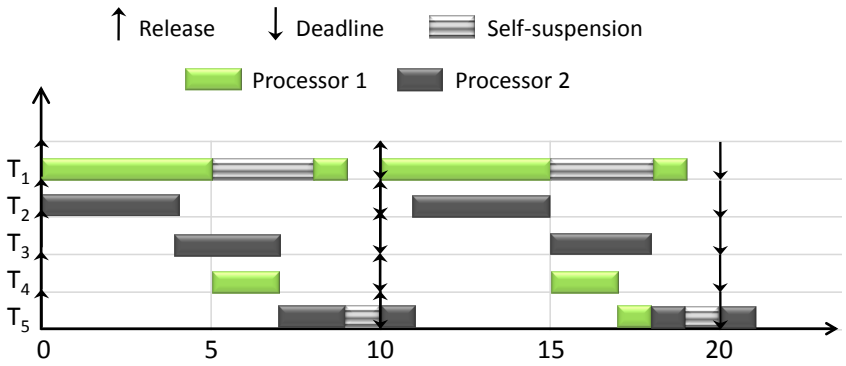


Figure 1: GEDF schedule for Eample 1.

misses its deadline, the release time of the next job of that task is not altered. We require for any task T_l, $e_l + s_l \leq p_l$, $u_l \leq 1$, and $U_{sum} \leq m$; otherwise, tardiness can grow unboundedly (if $e_l + s_l > p_l$ or $u_l > 1$, then T_l's tardiness grows unboundedly; if $U_{sum} > m$, then the system is overloaded, which implies that tardiness grows unboundedly for at least one task).

Under GEDF (GFIFO), released jobs are prioritized by their deadlines (release times). So that our results can be applied to both algorithms, we consider a generic scheduling algorithm (GSA) where each job is prioritized by some time point between its release time and deadline. Specifically, for any job $T_{i,j}$, we define a priority value $\rho_{i,j} = r_{i,j} + \kappa \cdot p_i$, where $0 \leq \kappa \leq 1$. Lower priority values denote higher priorities, and ties are assumed to be broken in favor of tasks with smaller indices. Note that GEDF and GFIFO are special cases of GSA where κ is set to 1 and 0, respectively.

Example 1. Consider a task system scheduled on two processors that contains five tasks: $T_1(e5, s3, e1, 10)$, $T_2(e4, 10)$, $T_3(e3, 10)$, $T_4(e2, 10)$, and $T_5(e2, s1, e1, 10)$. For this task system, $U^s_{sum} = u_1 + u_5 = 0.6 + 0.3 = 0.9$, $U_L = u_2 = 0.4$, and the maximum suspension ratio equals $\xi_1 = \frac{3}{6+3} = \frac{1}{3}$. Thus, it is schedulable since $U^s_{sum} + U_L = 1.3 < (1 - \xi_{max}) \cdot m = \frac{4}{3}$. Fig. 1 depicts a GEDF schedule for this system. As seen in the schedule, jobs of T_5 miss their deadlines by one time unit. Nonetheless, tardiness in this task system can be shown to be bounded so it is considered to be schedulable.

Tardiness bound for sporadic self-suspending task systems. Our previously-presented tardiness bound for sporadic self-suspending task systems [8] is given in Theorem 1 below.

Definition 1. Let E^s_{sum} (S^s_{sum}) be the total execution cost (total suspension length) of all suspending tasks in τ. Let E_{sum} be the total execution cost of all tasks in τ. Let u^s_{max} be the maximum utilization of any suspending task in τ^s.

Definition 2. Let U^c_L (E^c_L) be the sum of the $\min(m-1, c)$ largest computational task utilizations (computational task execution costs), where c is the number of computational tasks in τ.

Definition 3. Let $V = E^s_{sum} + E^c_L + u^s_{max} \cdot S^s_{sum} + (m-1)e_l + m \cdot s_l + 3n \cdot S_{max}$.

Theorem 1. *[8] The tardiness of any task T_l in τ scheduled under GSA is at most $x + e_l + s_l$, where*

$$x = \frac{V}{(1 - \xi_{max}) \cdot m - U^s_{sum} - U^c_L}, \tag{2}$$

provided $U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot m$.

x is well-defined provided (1) holds. If this condition holds and x equals the right-hand side of (2), 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.

3 Treating Partial Suspensions as Computation

In this section, we present our proposed approach, which we call PSAC (Partial Suspensions As Computation). As shown in [8], treating *all* suspensions as computation is superior to the approach in [8] when per-task utilizations are high and suspensions are long. By observing that the analysis in [8] actually treats no suspension as computation, our motivation is to investigate intermediate choices between the two extremes of treating all versus no suspensions as computation.

The goal of PSAC is to decrease ξ_{max} for any given task system τ. To motivate the details that follow, suppose that T_i has the maximum suspension ratio and it is the only task with that suspension ratio, and after converting z_i time units of its suspension time to computation, it still

has the maximum suspension ratio. Treating z_i time units of the suspension time of T_i as computation, (1) becomes $U^s_{sum} + U^c_L + z_i/p_i < (1 - \frac{s_i}{s_i+e_i} + \frac{z_i}{s_i+e_i}) \cdot m$. Given that $e_i + s_i \leq p_i$ and $m \geq 2$, $z_i/p_i < \frac{z_i}{s_i+e_i} \cdot m$ holds. Therefore, as long as $U_{sum} + z_i/p_i \leq m$ holds, the utilization constraint in (1) becomes less stringent if z_i time units of the suspension time of T_i are treated as computation.

Let c_i denote the length of the suspension time of task T_i that is treated as computation by PSAC. For any task T_i, after treating c_i time units of suspensions as computation, its suspension ratio becomes $\frac{s_i - c_i}{e_i + s_i}$, and its utilization becomes $u_i + \frac{c_i}{p_i}$. A set of values c_i ($1 \leq i \leq n$) is *valid* if the following conditions (3)-(5) hold.

$$U^s_{sum} + U^c_L + \sum_{i=1}^{n} \frac{c_i}{p_i} < (1 - \xi_{max}) \cdot m \tag{3}$$

$$U_{sum} + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq m \tag{4}$$

$$0 \leq c_i \leq s_i \tag{5}$$

If (3) and (4) hold, then, after treating c_i time units of the suspension time of each task T_i as computation, τ satisfies (1) and its total utilization is at most m (note that, in (3), ξ_{max} is the maximum suspension ratio after treating suspensions as computation). Thus, (3) and (4) guarantee that τ becomes schedulable. Moreover, (5) trivially must hold.

Therefore, our goal is to find a set of valid values c_i ($1 \leq i \leq n$). We present in this section a solution using linear programming. In an appendix, we present an alternative algorithm that optimally finds valid c_i values (if they exist) and that has low time complexity.

3.1 Linear Programming Approach

We formalize the problem of finding a set of valid c_i values by specifying a linear program as follows. First, note that we want to minimize $\sum_{i=1}^{n} c_i$ because this results in a lower tardiness bound, according to Theorem 1. Given this, an appropriate linear program can be specified based on the constraints (3)-(5) given earlier. Let ϵ be a constant that can be arbitrarily small. The linear program we wish to solve is as follows

$$\text{Minimize} \quad \sum_{i=1}^{n} c_i$$

subject to:

$$U^s_{sum} + U^c_L + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq (1 - \frac{s_i - c_i}{e_i + s_i}) \cdot m - \epsilon, \quad i = 1, ...n \tag{6}$$

$$U_{sum} + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq m \tag{7}$$

$$c_i \leq s_i, \ i = 1, ...n \tag{8}$$

$$c_i \geq 0, \ i = 1, ...n \tag{9}$$

In order to formalize the problem as a linear program, we replace the constraint in (3) by the set of constraints in (6). In particular, instead of a single constraint involving ξ_{max}, we have a constraint for each individual ξ_i. Also, these constraints are expressed so that equality is allowed, which is required in linear programs. This requires the introduction of a small ϵ term, as seen. This linear programming approach can find valid c_i values in polynomial time.

Henceforth, when considering algorithms for computing valid c_i values, we regard c_i as a variable and $\overline{c}_i$ as a value assigned to that variable. This is in keeping with how variables and values are denoted in work on linear programming.

Example 2. Consider a task system scheduled on four processors under GEDF that contains seven tasks: $T_1(e3, s6, e1, 10)$, $T_2(e4, s2, e2, 8)$, $T_3(e1, 3)$, $T_4(e8, s1, e6, 20)$, $T_5(e1, 6)$, $T_6(e2, 10)$, $T_7(e3, 15)$, $T_8(e1, 5)$, $T_9(e1, 10)$, and $T_{10}(e4, 20)$, where T_1, T_2, sand T_4 are self-suspending tasks and all other tasks are computational tasks. This task system is deemed to be unschedulable by treating all suspensions as computation because this causes total utilization to exceed 2.0. It is also deemed to be unschedulable by using the analysis presented in [8] because it violates (1): $U^s_{sum} + U_l = 0.4 + 0.75 + 0.7 + \frac{1}{3} + \frac{1}{5} + \frac{1}{5} > (1 - \xi_{max}) \cdot m = (1 - 0.6) \cdot 4 = 1.6$.

However, this task system is determined to be schedulable by PSAC. By solving the above linear program, a set of valid c_i values can be obtained, where $\overline{c}_1 = \frac{59}{18}$ and $\overline{c}_i = 0$ (where $2 \leq i \leq 7$). The existence of valid c_i values implies that the system is schedulable.

3.2 Tardiness Bound

If PSAC finds valid c_i values, then τ is schedulable. In this case, Theorem 1 can be applied to compute a tardiness bound, assuming execution costs and suspension times are updated in accordance with the c_i values PSAC produces.

4 Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the effectiveness of PSAC. We compare PSAC with the two extremes of treating no suspension as computation (i.e., using the analysis stated in [8]), denoted NSAC, and treating all suspensions as computation (i.e., the common approach), denoted ASAC. In the variant of PSAC examined here, if valid c_i values cannot be found, then schedulability is checked via ASAC. That is, we exploit the fact that if all suspensions must be viewed as computation, then the less stringent utilization constraint from [3] can be applied.

In our experiments, we generated task sets based upon distributions proposed by Baker [1]. Task periods were uniformly distributed over [10*ms*,100*ms*]. Task utilizations were distributed differently for each experiment using three uniform and three bimodal distributions. The ranges for the uniform distributions were [0.001,0.1] (light), [0.1,0.4] (medium), and [0.4,0.9] (heavy). In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.4) or [0.4, 0.9] with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). Task execution costs were calculated from periods and utilizations. Given that it is common case for real-time workloads to have both self-suspending tasks and computational tasks, we varied U^s_{sum} as follows: $U^s_{sum} = 0.1 \cdot U_{sum}$ (suspensions are relatively infrequent), $U^s_{sum} = 0.4 \cdot U_{sum}$ (suspensions are moderately frequent), and $U^s_{sum} = 0.7 \cdot U_{sum}$ (suspensions are frequent). Moreover, we varied ξ_{max} as follows: 0.1 (suspensions are short), 0.3 (suspensions are moderate), and 0.6 (suspensions are long). Table 1 shows suspension-length ranges generated by these parameters. U_{sum} was varied within {1, 2, ..., 8}. For each combination of (task utilization distribution, U^s_{sum}, U_{sum}, ξ_{max}), 1,000 task sets were generated for an eight-processor system. For each generated system, soft real-time schedulability (i.e., the ability to ensure bounded tardiness) was checked for PSAC, NSAC, and ASAC. In checking schedulability, system overheads were ignored (factoring overheads into our analysis is beyond the scope of this paper).

Schedulability results obtained using uniform and bimodal light task utilization distributions are shown in Fig. 2 (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.

suspension length / per-task utilization		short suspensions $\xi_{max} = 0.1$	moderate suspensions $\xi_{max} = 0.3$	long suspensions $\xi_{max} = 0.6$
light	min:	26 *µs*	409 *µs*	750 *µs*
	avg:	144 *µs*	2.25 *ms*	4.125 *ms*
	max:	263 *µs*	4.09 *ms*	7.5 *ms*
medium	min:	131 *µs*	2.05 *ms*	3.75 *ms*
	avg:	723 *µs*	11.25 *ms*	20.6 *ms*
	max:	1.3 *ms*	20.5 *ms*	37.5 *ms*
heavy	min:	342 *µs*	5.3 *ms*	9.75 *ms*
	avg:	1.9 *ms*	29.25 *ms*	53.625 *ms*
	max:	3.42 *ms*	53.2 *ms*	97.5 *ms*

Table 1: Per-job suspension-length ranges.

As shown in Fig. 2, PSAC proved to be able to significantly improve schedulability in most tested scenarios. Both NSAC and ASAC are negatively impacted when increasing task utilizations, U^s_{sum}, or ξ_{max}. PSAC tries to treat partial suspensions as computation, which effectively decreases the value of ξ_{max} at the cost of only marginally increasing the left side of (1). By examing the right side of (1), decreasing ξ_{max} can quite significantly relax the utilization constraint (due to the multiplication factor m). An interesting observation is that when suspensions are short, PSAC can guarantee 100% schedulability in all scenarios. Another interesting observation is that when the total utilization equals 8.0, PSAC yields the same schedulability as NSAC. This is because treating any suspension as computation in this case causes total utilization to exceed 8.0.

Schedulability results obtained using uniform and bimodal medium task utilization distributions are shown in Fig. 3. Again, PSAC proved to be able to significantly improve schedulability in all tested scenarios. When total utilization is no greater than 6.0 and suspensions are infrequent or moderately frequent, almost 100% of all task sets were schedulable using PSAC. Moreover, if suspensions are less frequent or total utilization is smaller, then PSAC is considerably better than ASAC and NSAC. These trends are due to the fact that when total utilization is small and suspensions are not frequent, the left side of (1) is small. This gives PSAC more freedom in treating suspensions as computation. Another interesting observation is that, in comparison to the light-utilization case, the improvement seen in PSAC is less. This may be due to the fact that when task utilizations become higher, U_L becomes larger, which constrains PSAC's ability to treat suspensions as computation.

Schedulability results obtained using uniform and bimodal heavy task utilization distributions are shown in Fig. 3. Once again, PSAC proved to be able to improve schedulability in all tested scenarios. When total utilization is less than 7.0 and suspensions are short, PSAC can achieve almost 100% schedulability. Interestingly, PSAC exhibited

greater improvement in the case of bimodal task utilization distributions. When using uniform heavy task utilization distributions, every self-suspending task or computational task has a heavy utilization and U^s_{sum} and U_L tend to be large, which constrains PSAC. On the other hand, when using bimodal heavy task utilization distributions, U^s_{sum} and U_L have a higher chance of being smaller than in the uniform case, which constrains PSAC less. Moreover, when using bimodal distributions, generated task sets tend to have fewer tasks that have long suspensions. Therefore, PSAC can treat less suspension time as computation and has a better chance of finding valid c_i values.

5 Conclusion

We have proposed an approach called PSAC that can improve the schedulability of soft real-time sporadic self-suspending task systems on a multiprocessor. PSAC seeks to decrease the maximum suspension ratio at a cost of some utilization increase. We have presented a linear programming approach as well as an optimal algorithm with low time complexity to implement PSAC. In all scenarios considered in our experimental study, PSAC was able to improve schedulability, and in most scenarios, by a substantial margin.

References

[1] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *Technical Report TR-051101, Florida State University*, 2005.

[2] 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.

[3] U. Devi and J. Anderson. Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symp.*, pp. 330-341, 2005.

[4] 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.

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

[6] H. Leontyev and J. Anderson. Tardiness Bounds for FIFO Scheduling on Multiprocessors. In *Proc. of the 19th Euromicro Conf. on Real-Time Systems*, pp. 71-80, 2007.

[7] C. Liu and J. H. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symp.*, 2010, to appear.

[8] 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.

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

[10] 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.

[11] 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.

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

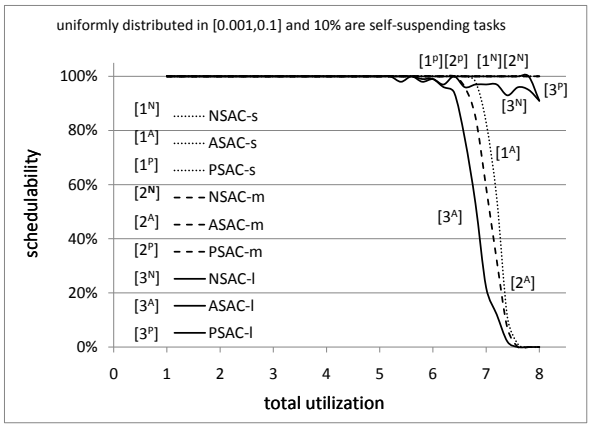
[13] F. Ridouard and P. Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *Proc. of the 14th Real-Time and Network Systems*, pp. 15-24, 2006.

[14] 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.

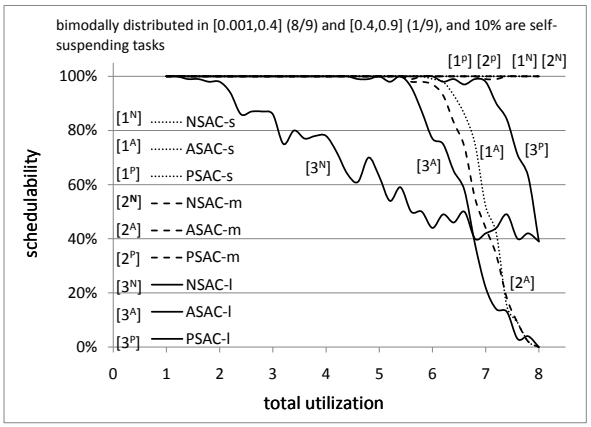
6 Appendix

Although the linear programming approach presented in Sec. 3 enables valid c_i values (if they exist) to be obtained in polynomial time, polynomial-time algorithms for solving linear programs typically have high runtime overheads. As an alternative, we present in this appendix an optimal algorithm that generates minimal valid c_i values in $O((N^s)^2)$ time. A set of valid values $\{\overline{c}_1, \overline{c}_2, ..., \overline{c}_n\}$ is said to be *minimal* if $\sum_{i=1}^{n} \overline{c}_i$ is minimal over all valid sets of c_i values. An algorithm is *optimal* if it can find a minimal valid set of c_i values if a valid set of c_i values exists.

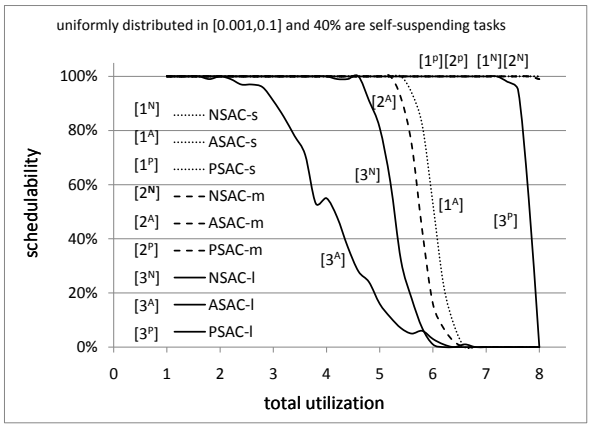
Let $\lambda_i = \{T_j \mid t_j \in \tau$ and ξ_j is ranked as the i^{th} largest suspension ratio$\}$. Thus, tasks in λ_1 have the same suspension ratio, which equals ξ_{max}. Obviously, we can decrease ξ_{max} by treating some suspension time of every task in λ_1 as computation, at the cost of increasing U^s_{sum} by an amount commensurate with the additional computation. Let



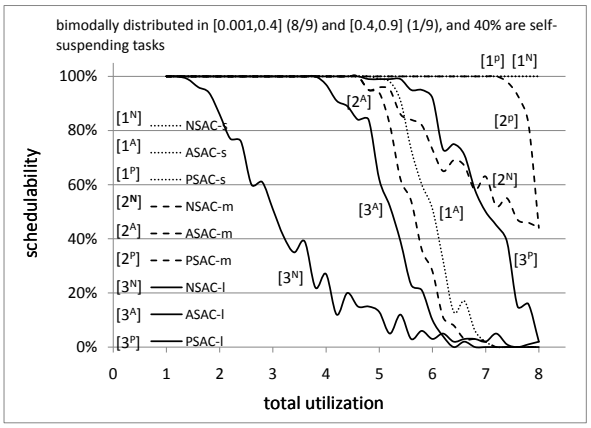
(a)



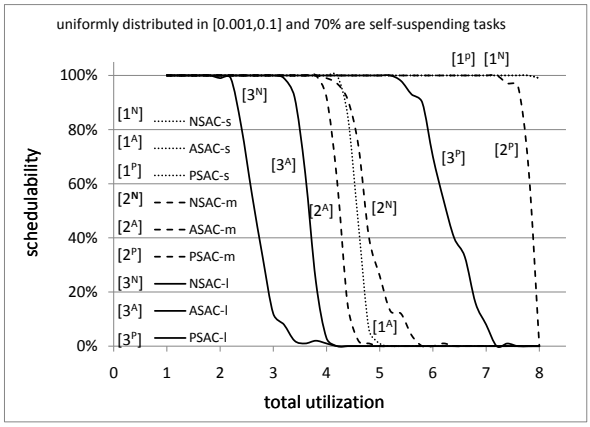
(b)



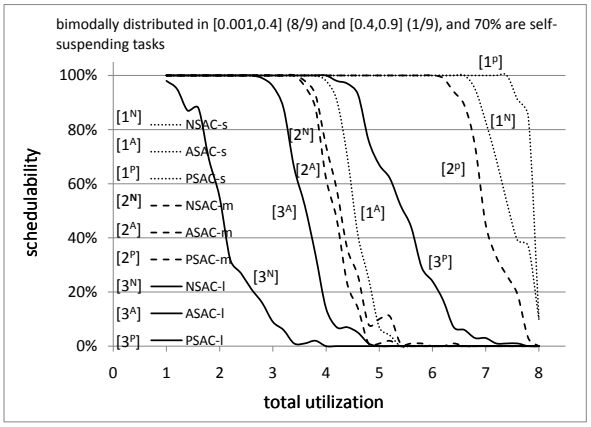
(c)



(d)



(e)



(f)

Figure 2: Soft real-time schedulability results for uniform and bimodal light task utilization distributions. In the first (respectively, second and third) row of graphs, relatively infrequent (respectively, moderately frequent and frequent) suspensions are assumed. Each graph gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. The label "**PSAC-s(m/l)**" indicates the approach of this paper assuming short (moderate/long) suspensions. Similar "**NSAC**" and "**ASAC**" labels are used for NSAC and ASAC.

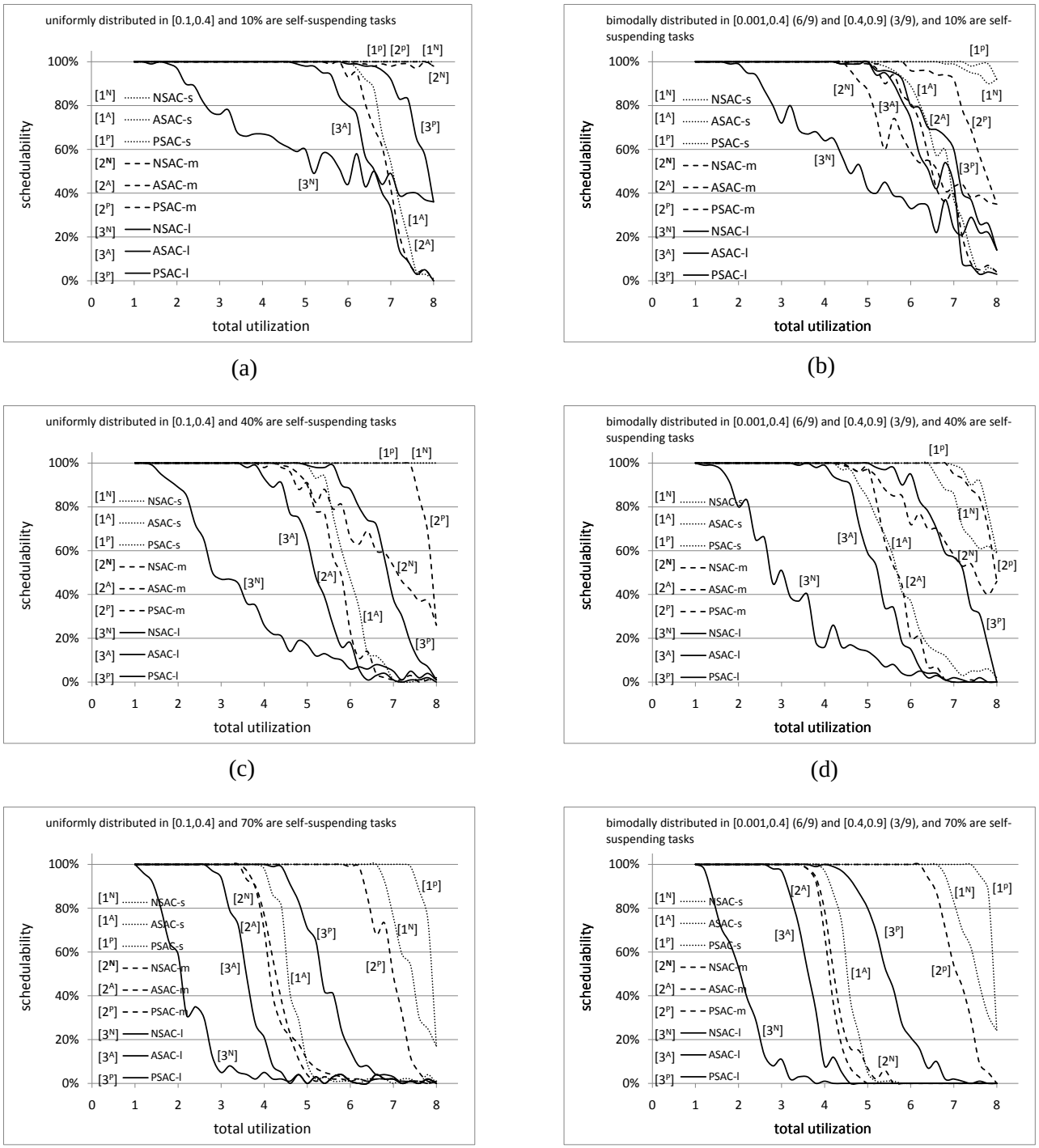


Figure 3: Soft real-time schedulability results for uniform and bimodal medium task utilization distributions. In the first (respectively, second and third) row of graphs, relatively infrequent (respectively, moderately frequent and frequent) suspensions are assumed. Each graph gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. The label "**PSAC-s(m/l)**" indicates the approach of this paper assuming short (moderate/long) suspensions. Similar "**NSAC**" and "**ASAC**" labels are used for NSAC and ASAC.

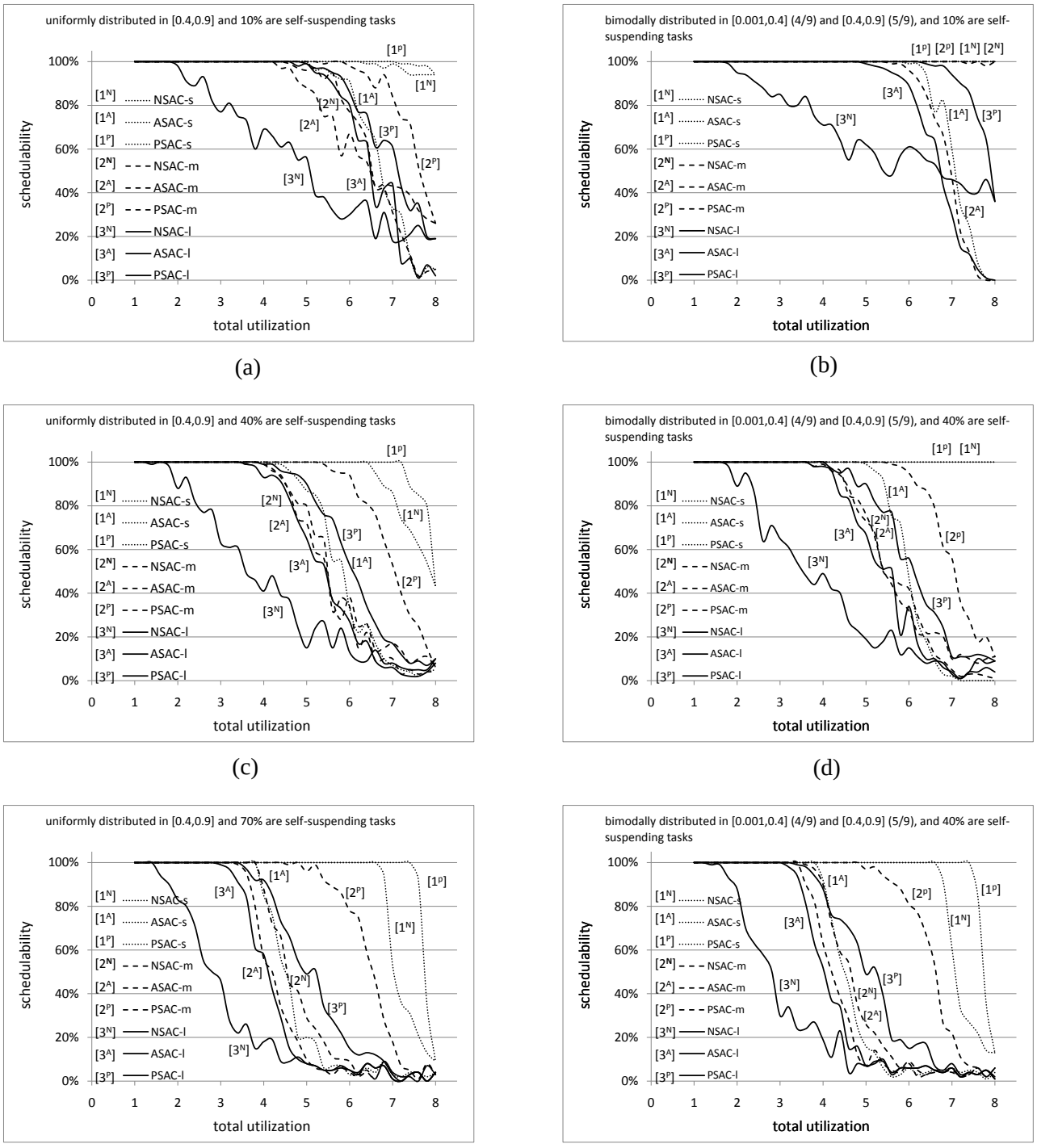


Figure 4: Soft real-time schedulability results for uniform and bimodal heavy task utilization distributions. In the first (respectively, second and third) row of graphs, relatively infrequent (respectively, moderately frequent and frequent) suspensions are assumed. Each graph gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. The label "**PSAC-s(m/l)**" indicates the approach of this paper assuming short (moderate/long) suspensions. Similar "**NSAC**" and "**ASAC**" labels are used for NSAC and ASAC.

$\{\lambda_1^1, \lambda_1^2 ... \lambda_1^{q_i}\}$ denote the tasks in λ_1. Let $c(\lambda_i^j)$ denote the amount of the suspension time of task λ_i^j that is treated as computation by PSAC, $e(\lambda_i^j)$ denote λ_i^j's execution cost, $s(\lambda_i^j)$ denote its suspension length, $p(\lambda_i^j)$ denote its period, $u(\lambda_i^j)$ denote its utilization, and $\xi(\lambda_i^j)$ denote its suspension ratio. Since we want to decrease ξ_{max}, we initially only care about tasks in λ_1.

We apply two constraints in order for the algorithm to run in $O((N^s)^2)$ time.

First, we desire to reduce the suspension ratio of every task in λ_1 by the same amount, as stated in (10). Reducing any task λ_1^i's suspension ratio by a greater amount than other tasks in λ_1 does not further decrease ξ_{max}, but only increases the utilization of λ_1^i.

$$\forall j, k :: \frac{s(\lambda_1^j) - c(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} = \frac{s(\lambda_1^k) - c(\lambda_1^k)}{e(\lambda_1^k) + s(\lambda_1^k)} \tag{10}$$

Second, we desire to reduce the suspension ratio of each task in λ_1 to no less than $\xi(\lambda_2^1)$ (or 0 if λ_2^1 does not exist), as stated in (11). Further reducing some $\xi(\lambda_1^i)$ does not decrease ξ_{max}, but only increases the utilization of λ_1^i.

$$\forall j :: \frac{s(\lambda_1^j) - c(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} \geq \xi(\lambda_2^1) \tag{11}$$

With the above preliminaries in place, the algorithm can be stated as a two-step procedure.

1. Find valid values $\{\overline{c}(\lambda_1^1), \overline{c}(\lambda_1^2), ..., \overline{c}(\lambda_1^{q_1})\}$ that also satisfy (10) and (11). If such values exist, then τ is schedulable. Otherwise, go to Step 2.

2. Decrease the suspension ratio of every task in λ_1 to $\xi(\lambda_2^1)$ by defining $\overline{c}(\lambda_1^j)$ for $\lambda_1^j \in \lambda_1$ so that $\frac{s(\lambda_1^j) - \overline{c}(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} = \xi(\lambda_2)$ and (4) hold. If valid values do not exist, then τ is unschedulable. Otherwise, update the set λ and go to Step 1.

Time complexity. In the worst case, Steps 1 and 2 can be executed for N^s times. Step 1 has $O(N^s)$ time complexity if valid c_i values are obtained by the following procedure. First, solve (10) for each task λ_1^j, where $j \neq 1$, to specify $c(\lambda_1^j)$ as a function of $c(\lambda_1^1)$. This takes $O(N^s)$ time. Second, substitute all $c(\lambda_1^j)$ values as specified into (3)-(5) and (11), and solve for $c(\lambda_1^1)$. This also takes $O(N^s)$ time. Step 2 has $O(N^s)$ time complexity since it only needs to solve at most N^s equations. Therefore, the overall time complexity of the proposed algorithm is $O((N^s)^2)$.

Example 3. Consider the task system described in Example 2. For this system, λ_1 contains only T_1, which has the maximum suspension ratio. We find valid values for $c(\lambda_1^1)$ that satisfy (3)-(5) and (10)-(11) (note that satisfying (10) is actually not required because T_1 is the only task in λ_1). (3) requires that $U_{sum}^s + U_L^c + \sum_{j=1}^{q_1} \frac{c(\lambda_1^j)}{p(\lambda_1^j)} = 1.85 + \frac{11}{15} + \frac{c(T_1)}{10} < \left(1 - \frac{6 - c(T_1)}{10}\right) \cdot 4$, that is, $c(T_1) > \frac{59}{28}$ must hold. (4) requires $U_{sum} + \sum_{j=1}^{q_1} \frac{c(\lambda_1^j)}{p(\lambda_1^j)} = \frac{13}{4} + \frac{c(T_1)}{10} \leq m = 4$, that is, $c(T_1) \leq 7.5$ must hold. (5) requires that $c_1 \leq s_i = 6$. (11) requires $\frac{s_1 - c(T_1)}{e_1 + s_1} = \frac{6 - c(T_1)}{4 + 6} > \xi(\lambda_2^1) = \xi_2 = 0.25$, that is, $c(T_1) < 3.5$ must hold. Thus, any value satisfying $59/18 \leq c(T_1) \leq 3.5$ is a valid value for $c(T_1)$ and $\overline{c}(T_i) = 59/18$ is minimal. The existence of a valid value implies that the system is schedulable.

Optimality of the proposed algorithm. The optimality of the proposed algorithm is proved in the following theorem.

Theorem 2. *For any sporadic self-suspending task system τ, the proposed algorithm will find a minimal valid set of c_i values if a valid set of c_i values exists.*

Proof. Let $\{\overline{c}_1, \overline{c}_2, ..., \overline{c}_n\}$ be a minimal valid set of c_i values. We first prove by contradiction that, for any j, if $\overline{c}_j > 0$, then $\frac{s_j - \overline{c}_j}{e_j + s_j} = \xi_{max}$.

For any $\overline{c}_j > 0$, if $\frac{s_j - \overline{c}_j}{e_j + s_j} < \xi_{max}$ holds, then less than $\overline{c}_j$ time units of the suspension time of T_j can be treated as computation to make ξ_j equal ξ_{max} or T_j's original suspension ratio, whichever is smaller. Note that this does not increase ξ_{max} but will decrease the utilization of T_j, which implies that (3)-(5) can still be satisfied. However, this contradicts with our assumption that $\{\overline{c}_1, \overline{c}_2, ..., \overline{c}_n\}$ is minimal.

Therefore, every task T_i with $\overline{c}_i > 0$ has the same suspension ratio, which is ξ_{max}. Thus, all such tasks are within the set λ_1, defined above.

The proposed algorithm determines c_i values in exactly the same way: it first checks whether it can find a valid set of c_i values by just considering tasks in λ_1. If there exists no such set, then it treats some suspension time of every task in λ_1 as computation to force them to have the same suspension ratio as tasks in λ_2. As a result λ_1 is redefined to include both the original λ_1 and λ_2. Continuing in this manner, it will find a minimal valid set of c_i values if one exists, because considering tasks in other λ_i sets does not decrease ξ_{max} but will increase both $\sum_{i=1}^{n} c_i$ and such a task's utilization. □