JOURNAL OF AEROSPACE INFORMATION SYSTEMS
Vol. 11, No. 12, December 2014

Schedulability Analysis of Task Sets with Upper- and Lower-Bound Temporal Constraints

Matthew C. Gombolay* and Julie A. Shah†
Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

DOI: 10.2514/1.I010202

Increasingly, real-time systems must handle the self-suspension of tasks (that is, lower-bound wait times between subtasks) in a timely and predictable manner. A fast schedulability test that does not significantly overestimate the temporal resources needed to execute self-suspending task sets would be of benefit to these modern computing systems. In this paper, a polynomial-time test is presented that is known to be the first to handle nonpreemptive self-suspending task sets with hard deadlines, where each task has any number of self-suspensions. To construct the test, a novel priority scheduling policy is leveraged, the *j*th subtask first, which restricts the behavior of the self-suspending model to provide an analytical basis for an informative schedulability test. In general, the problem of sequencing according to both upper-bound and lower-bound temporal constraints requires an idling scheduling policy and is known to be nondeterministic polynomial-time hard. However, the tightness of the schedulability test and scheduling algorithm are empirically validated, and it is shown that the processor is able to effectively use up to 95% of the self-suspension time to execute tasks.

I. Introduction

REAL-TIME scheduling systems are a vital component of many aerospace, medical, nuclear, manufacturing, and transportation systems. In general, real-time systems must be able to interact with their environment in a timely and predictable manner, and designers must engineer analyzable systems for which the timing properties can be predicted and mathematically proven correct [1,2]. An analysis is typically performed using schedulability tests, which are fast methods for determining whether a system can process a set of tasks within specified temporal constraints [1,3–5].

Increasingly real-time systems must handle the self-suspension of tasks, and new methods are required for testing the feasibility of these self-suspending task sets [6–9]. In processor scheduling, self-suspensions (i.e., lower-bound "wait times" between subtasks) can result, both due to hardware and software architectures. At the hardware level, the addition of multicore processors, dedicated cards (e.g., graphics processing units, physics processing units, etc.), and various input/output devices, such as external memory drives, can necessitate task self-suspensions. Furthermore, the software that uses these hardware systems can employ synchronization points and other algorithmic techniques that also result in self-suspensions [10]. Schedulability tests that do not significantly overestimate the temporal resources needed to execute self-suspending task sets would be of benefit to these modern computing systems.

The sequencing and scheduling of tasks according to upper-bound and lower-bound (self-suspension) temporal constraints are challenging problems with important applications outside of processor scheduling as well. Other examples include autonomous tasking of unmanned aerial and underwater vehicles [11,12], scheduling of factory operations [13,14], and scheduling of aircraft and flight crews [15]. New uses of robotics for flexible manufacturing are pushing the limits of current state-of-the-art methods in artificial intelligence (AI) and operations research (OR) and are spurring industrial interest in fast methods for sequencing and scheduling [13]. Solutions to these applications typically draw from methods in AI and OR [14–17], which provide complete search algorithms that require exponential time to compute a solution in the worst case. These methods cannot provide fast recomputation of the schedule in response to dynamic disturbances for large real-world task sets. Fast, sufficient schedulability tests, while widely used in processor scheduling, are underused in these applications.

In this paper, we present a uniprocessor schedulability test and complementary scheduling algorithm that handle periodic nonpreemptive self-suspending task sets. To our knowledge, our approach is the first polynomial-time test for nonpreemptive self-suspending task sets with any number of self-suspensions in each task. We also generalize our uniprocessor schedulability test and algorithm to handle deadline constraints not found in the traditional self-suspending task model but commonly found in artificial intelligence and operations research models.

Our schedulability test and scheduling algorithm use a novel scheduling policy to create a problem structure in self-suspending task networks. Restricting the behavior of the scheduler sacrifices completeness for this nondeterministic polynomial-time (NP) hard problem [9,18], in general. However, we show that this restriction enables the design of an informative schedulability test and scheduling algorithm, both of which produce near-optimal results for many real-world task systems.

We begin in Sec. II with the definition of a self-suspending task model. Section III reviews prior art in real-time scheduling of self-suspending task sets, and Sec. IV introduces terminology to describe our schedulability test and scheduling algorithm. Section V discusses how we restrict the behavior of the scheduler so as to enable the design of an informative schedulability test and scheduling algorithm.

In Sec. VI, we present our schedulability test with proof of correctness. Section VII describes our complementary scheduling algorithm, which successfully executes task sets that pass the schedulability test. In Sec. VIII, we empirically validate the performance of our schedulability test and scheduling algorithm. We show that our schedulability test is tight, meaning that it does not significantly overestimate the temporal resources needed to execute the task set. We also show that a processor operating under our scheduling algorithm incurs little processor idle time. Lastly, we demonstrate empirically that our schedulability test is fast, and we derive the computational complexity of our test and scheduling algorithm.

Presented as Paper 2012-2431 at the Infotech@Aerospace 2012, Garden Grove, CA, 19–21 June 2012; received 1 November 2013; revision received 22 June 2014; accepted for publication 2 September 2014; published online 15 December 2014.

*gombolay@csail.mit.edu (Corresponding Author).
†julie_a_shah@csail.mit.edu.

II. Self-Suspending Task Model

The basic model for the periodic self-suspending task set [6] is shown in Eq. (1):

$$\tau_i\colon (\phi_i, (C_i^1, E_i^1, C_i^2, E_i^2, \ldots, E_i^{m_{i-1}}, C_i^{m_i}), T_i, D_i) \tag{1}$$

In this model, there is a task set $\boldsymbol{\tau}$ where all tasks, $\tau_i \in \boldsymbol{\tau}$, must be executed by a uniprocessor. For each task, there are m_i subtasks with $m_i - 1$ self-suspension intervals. Subtasks within a task are executed in order, meaning that a subtask τ_i^{j+1} must execute after subtask τ_i^j and before subtask τ_i^{j+2}. Subtasks across tasks may be interleaved in any order that meets the temporal constraints of the model. Each self-suspension is a lower-bound temporal constraint, or wait time, between the finish and start of the consecutive subtasks and may result in processor idle time. C_i^j is the worst-case duration (i.e., called "cost") of the jth subtask of τ_i, and E_i^j is the worst-case cost of the jth self-suspension interval of τ_i.

A new instance of τ_i is released (meaning it becomes available for execution) each period T_i, and all subtasks in task τ_i must be completed within the deadline D_i from the release time of that task instance. The kth instance of subtask τ_i^j is denoted $\tau_{i,k}^j$. A phase offset delays the release of the first instance of a task τ_i by the duration ϕ_i. The hyperperiod of a set of tasks is the least common multiple of the set of deadlines D_i. This is the time after which the pattern of the set of tasks repeats. The hyperperiod H bounds the time horizon necessary for schedulability analysis.

The self-suspending task model shown in Eq. (1) provides a solid basis for describing many real-world processor scheduling problems of interest. In this work, we augment the traditional model to provide additional expressiveness by incorporating deadline constraints that upper bound the temporal difference between the start and finish of two subtasks within a task. We call these deadline constraints subtask-to-subtask deadlines. We define a subtask-to-subtask deadline as shown in Eq. (2):

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}\colon (f_i^b - s_i^a \le d^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}) \tag{2}$$

where f_i^b is the finish time of subtask τ_i^b, s_i^a is the start time of subtask τ_i^a, and

$$d^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

is the upper-bound temporal constraint between the start and finish times of these two subtasks, such that $b > a$.

Subtask-to-subtask constraints are commonly included in AI and operations research scheduling models (e.g., [19–21]) and are vital in modeling many real-world problems. We augment the self-suspending task model in this way to illustrate the relevance of our techniques to important applications other than processor scheduling. Consider the sequencing and scheduling of assembly manufacturing processes. In this case, each manufactured piece is represented by a uniprocessor and the work performed on the piece is represented by the subtasks. The goal is to sequence the work to assemble the piece subject to temporal and precedence constraints among subtasks. Self-suspensions (i.e., lower-bound wait times between subtasks) may arise due to, for example, "cure times" involved in the assembly process. Upper-bound temporal constraints also arise naturally; the build schedule may require that a sequence of tasks be grouped together and executed in a specified time window.

The problem of sequencing arriving and departing aircraft on a runway is also analogous to processor scheduling. Here, the runway represents the uniprocessor, and the constraints that landing aircraft be spaced by a minimum separation time are represented as self-suspensions. Upper-bound subtask-to-subtask deadlines encode the amount of time an aircraft can remain in a holding pattern based on fuel considerations. Although each domain has its own nuances in problem formulation, there is sufficient underlying commonality in the problem structure to investigate the application of real-time scheduling techniques to these problems.

Figure 1 and Table 1 present an illustrative augmented self-suspending task set. The set includes four tasks. For example, the first task τ_1 has a phase offset $\phi_1 = 0$, a deadline $D_1 = 22$, a period $T_1 = 24$, and a subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$$

requiring τ_1^3 to finish no later than six units of time after the start of τ_1^2. In Fig. 1, upward arrows indicate the releases of a task, and downward arrows indicate a task's deadlines D_i. Self-suspensions are represented by horizontal bars with corresponding labels. The execution cost of each subtask is shown as a block with a number corresponding to its subtask index. For example, in the figure, the block labeled "2" on the row labeled "τ_1" corresponds to τ_1^2. The subtask-to-subtask deadline is depicted with a brace and its corresponding label, e.g.,

$$D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$$

One hyperperiod H for this task set is the interval from $t = [0, 24]$.

In the remainder of this paper, we present a schedulability test and complementary scheduling algorithm that handle periodic self-suspending task sets. We develop the test for nonpreemptible subtasks, meaning the interruption of a subtask significantly degrades its quality. To our knowledge, this is the first polynomial-time test and algorithm that handles nonpreemptive self-suspending task sets with hard deadlines. We also generalize our schedulability test and algorithm to handle subtask-to-subtask deadlines, to increase the applicability of our techniques to real-time scheduling problems found in various application domains.

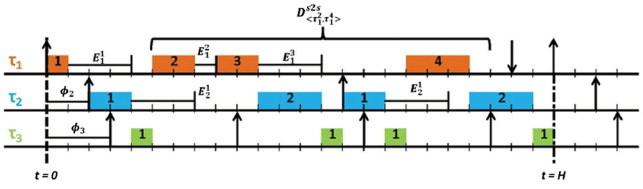


Fig. 1 Illustration of a schedule for the task set described in Table 1. The self-suspending task model is augmented with a subtask-to-subtask deadline $D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$.

Table 1 Parameters for an illustrative self-suspending task model[a]

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	E_i^2	C_i^3	E_i^3	C_i^4	D_i	T_i
$i=1$	0	1	3	2	1	2	3	3	22	24
$i=2$	2	2	3	3	0	0	0	0	12	12
$i=3$	3	1	0	0	0	0	0	0	6	6

[a]A schedule for this task set is depicted in Fig. 1. A "0" entry in the i row signifies that τ_i does not include the model feature. For example, τ_3 has only one subtask, so $C_3^2, E_3^2, \ldots, C_3^4$ are denoted 0.

III. Background

In this section, we briefly review the challenges for real-time scheduling of self-suspending tasks sets, including prior work in analytical schedulability tests and scheduling algorithms.

A. Challenge Posed by Task Self-Suspension

Scheduling a self-suspending task set is NP hard can be shown through an analysis of the interaction of self-suspensions and task deadlines [8,21,22]. Many uniprocessor priority-based scheduling algorithms, such as the earliest-deadline-first (EDF) or rate-monotonic (RM) algorithms, introduce scheduling anomalies, since they do not account for this interaction [6,18].

A scheduling anomaly arises when a scheduler can produce a feasible schedule for a task set $\boldsymbol{\tau}$ but not for a relaxation of the task set $\boldsymbol{\tau}'$. Relaxations include reducing task costs or decreasing phase offsets. These anomalies are present for both preemptive and nonpreemptive task sets. Lakshmanan et al. [6] reported that finding an anomaly-free scheduling priority for self-suspending task sets remains an open problem. The challenges posed by these anomalies motivate prior art in testing and scheduling the self-suspending task model, wherein scheduling priorities and structure restrictions are applied to support analysis and predictability [10,23].

We provide illustrations to exemplify different types of scheduling anomalies in Figs. 2 and 3. Each figure depicts a feasible schedule (top) and an infeasible schedule resulting from a scheduling anomaly (bottom).

1. *Scheduling Anomalies Produced by Reducing Task Cost*

The first type of scheduling anomaly occurs when a reduction in the cost of a subtask causes the processor to violate a deadline constraint. This type of scheduling anomaly was first described by Ridouard and Richard [18]. Figure 2 shows a scenario where the execution of three tasks under the earliest-deadline-first algorithm produces this type of scheduling anomaly.

In the top graph, we see a feasible schedule, with τ_2^1 interleaved during self-suspension E_1^1 and τ_1^2 interleaved during E_2^1. However, when the execution cost of τ_1^1 is decreased, τ_2^1 starts earlier. In turn, τ_2^2 and τ_3^1 are released at the same time. Because $D_2 < D_3$, τ_2^2 is prioritized over τ_3^1. The result is that the processor idles during E_3^1 and is unable to satisfy deadline D_3.

2. *Scheduling Anomalies Produced by Decreasing Phase Offsets*

Phase offsets also can cause scheduling anomalies. This type of anomaly occurs when the reduction of a phase offset duration allows a task to release earlier, and thus prevents the processor from satisfying all deadline constraints. Figure 3 shows a scenario where the execution of two tasks under the earliest-deadline-first algorithm produces this scheduling anomaly.

In the top graph, we see a feasible schedule with τ_3^1 interleaved during self-suspension E_2^1 and τ_2^2 interleaved during E_3^1. However, when the duration of phase offset ϕ_2 decreases to zero, the start time of τ_2 remains unchanged, despite the earlier deadline. Even though the subtasks are efficiently interleaved, the processor cannot satisfy the deadline for τ_2.

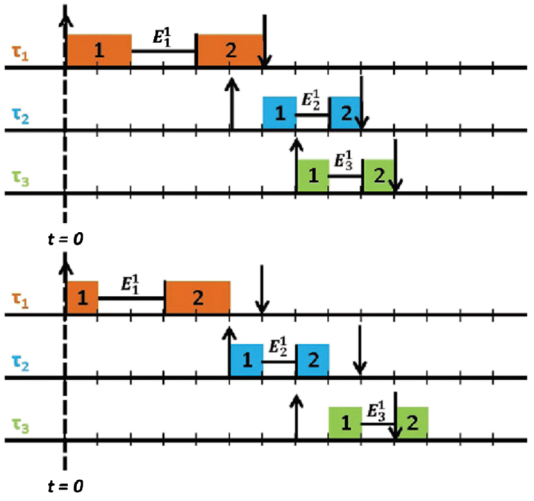


Fig. 2 Decreasing task cost makes this task set infeasible under EDF.

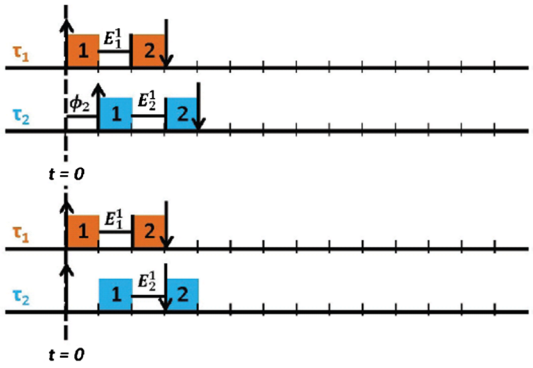


Fig. 3 Removing a phase offset makes this task set infeasible under EDF.

B. Schedulability Testing

Given sufficient time, the schedulability of a self-suspending task set may be computed offline using complete methods [24–26]. However, these approaches are not suitable for determining schedulability online in real time, as is necessary when the task set changes. To reduce computation time, many real-time systems use sufficient analytical schedulability tests that compute the feasibility of a given task set in polynomial time. These tests assume that the scheduler is using a specific scheduling priority, such as RM or EDF algorithms. The naive method for testing the schedulability of these task sets is to treat self-suspensions as a task cost; however, this can result in significant underutilization of the processor if the duration of the self-suspensions is large relative to task cost [7].

Fast polynomial-time schedulability tests have been studied for restrictions of the preemptive self-suspending task model. Kim et al. [4] presented two methods for testing task sets where each task has exactly one self-suspension. Their first method builds on work by Audsley et al. [27] to transform each task τ_i with two subtasks, τ_i^1 and τ_i^2, into two independent tasks. Both of the new tasks are released at time r_i, but τ_i^2 experiences release jitter to implicitly enforce the temporal dependency between τ_i^1 and τ_i^2. An iterative formula is developed [27] to calculate the worst-case response time for τ_i^1 and τ_i^2, and thereby the schedulability of the task set. The second method builds on this approach [28] to more tightly upper bound the amount of self-suspension time that must be considered as a task cost by analyzing which tasks can be interleaved during self-suspension time. Both these methods require a restriction be made on the specific time a task will self-suspend.

Next, Liu [1] and Devi [29] developed analyses for another restricted form of the preemptive task set, namely, where one self-suspension exists in the entire task set. Their approaches do not make an assumption on when a task will self-suspend. Liu's method [1] analyzes the schedulability of the task set when it is executed under the fixed-priority RM scheduling policy, and it treats delays of the tasks due to self-suspensions as external blocking events. This approach accounts for the situation where a higher-priority task self-suspends and the self-suspension terminates at the same time a lower-priority task is released, thus causing the lower-priority task to be delayed until the completion of the higher-priority task. Devi [29] developed a similar method for testing the schedulability of preemptive self-suspending task sets operating under the EDF dynamic-priority scheduling algorithm.

Recently, Abdeddaïm and Masson introduced an approach for testing preemptive self-suspending task sets using model checking with computational tree logic [24]. Although their method is easily extended to handle tasks with multiple self-suspensions, the runtime is exponential in the number of tasks. Thus, it does not currently scale to moderately sized task sets of interest for real-world applications. Lakshmanan and Rajkumar [10] also increased generality by developing a pseudo-polynomial-time test for the preemptive model to determine the worst-case interference imposed on a lower-priority self-suspending tasks by higher-priority nonsuspending tasks. However, Lakshmanan and Rajkumar reported that an exact-case test for multiple self-suspensions per task remains an open problem.

Finally, recent works by Liu and Anderson [7,30–32] analyzed preemptive task sets with multiple self-suspensions per task for soft real-time requirements. Liu and Anderson also considerrd nonpreemptive sections but for soft real-time systems [33]. We have not yet seen a schedulability test for hard, nonpreemptive task sets with multiple self-suspensions per task. Our approach seeks to fill this gap by providing the first such analytical schedulability test.

C. Scheduling Algorithms

Designing scheduling policies for self-suspending task sets also remains a challenge. Although not anomaly free, various priority-based scheduling policies have been shown to improve the online execution behavior in practice.

Rajkumar [23] presented an algorithm called the period enforcer for preemptive self-suspending task sets scheduled with the RM scheduling algorithm. Period enforcer works by adding preconditions to tasks in the processor queue that force the tasks to behave as ideal, periodic tasks. Period enforcer handles tasks that self-suspend during execution (i.e., creating discrete subtasks) by transforming the task τ_i into multiple tasks $\tau_{i'}$, $\tau_{i''}$, and $\tau_{i'''}$, each with the same deadline as τ_i. However, their approach does not handle nonpreemptive task sets, nor is there a complementary, analytical schedulability test.

Lakshmanan and Rajkumar [10] built on previous approaches to develop a static slack enforcement algorithm that delays the release times of subtasks to improve the schedulability of preemptive task sets. The static slack enforcement algorithm is optimal, in that it does not affect the worst-case response time of a self-suspending task and it prevents additional processing delays of lower-priority tasks due to higher-priority tasks.

Although these scheduling algorithms by Rajkumar [23] and Lakshmanan and Rajkumar [10] can handle preemptive self-suspending tasks sets with multiple suspensions per task, we have not yet seen a such an algorithm that is accompanied by a polynomial-time schedulability test. In this paper, we present a complementary schedulability test and scheduling algorithm. Furthermore, we extend our methods to handle subtask-to-subtask temporal constraints that are important in many scheduling problems outside of the processor scheduling domain.

IV. Terminology

In this section, we introduce new terminology to help describe our schedulability test and the execution behavior of self-suspending tasks, which in turn will help us intuitively describe the various components of our schedulability test.

Definition 1: A subtask is a free subtask, $\tau_i^j \in \boldsymbol{\tau}_{\text{free}}$, if there does not exist any deadline

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

such that $a < j \leq b$. In the example from Fig. 1, $\tau_{\text{free}} = \{\tau_1^1, \tau_1^2, \tau_2^1, \tau_2^2, \tau_3^1\}$.

Definition 2: A subtask is an embedded subtask, $\tau_i^j \in \boldsymbol{\tau}_{\text{embedded}}$, if there exists a deadline

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

such that $a < j \leq b$. In the example from Fig. 1, $\boldsymbol{\tau}_{\text{embedded}} = \{\tau_1^3, \tau_1^4\}$.

The intuitive difference between a free and an embedded subtask is as follows: a scheduler has the flexibility to sequence a free subtask relative to the other free subtasks without consideration of subtask-to-subtask deadlines. On the other hand, the scheduler must take extra consideration to satisfy subtask-to-subtask deadlines when sequencing an embedded subtask relative to other subtasks.

Definition 3: A self-suspension is a free self-suspension, $E_i^j \in \boldsymbol{E}_{\text{free}}$, if there does not exist any deadline

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

such that $a \leq j < b$. In the example from Fig. 1, $\boldsymbol{E}_{\text{free}} = \{E_1^1, E_2^1\}$.

Definition 4: A self-suspension is an embedded self-suspension, $E_i^j \in \boldsymbol{E}_{\text{embedded}}$, if there exists a deadline

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

such that $a \leq j < b$. In the example from Fig. 1, $\boldsymbol{E}_{\text{embedded}} = \{E_1^2, E_1^3\}$.

In Sec. VI, we describe how we can use $\boldsymbol{\tau}_{\text{free}}$ to reduce processor idle time due to $\boldsymbol{E}_{\text{free}}$ and, in turn, analytically upper bound the duration of the self-suspensions that needs to be treated as a task cost. We will also derive an upper bound on the processor idle time due to $\boldsymbol{E}_{\text{embedded}}$.

V. Motivating our jth-Subtask-First Priority Scheduling Policy

Scheduling of self-suspending task sets is challenging because polynomial-time priority-based approaches such as EDF can result in scheduling anomalies. To construct a tight schedulability test, we desire a priority method of restricting the execution behavior of the task set in a way that allows us to analytically bound the contributions of self-suspensions to the processor idle time, without unnecessarily sacrificing processor efficiency.

We restrict behavior using a novel scheduling priority, which we call the *j*th subtask first (JSF). We formally define the *j*th-subtask-first priority scheduling policy in Definition 5.

Definition 5: For the *j*th subtask first, we use j to correspond to the subtask index in τ_i^j. A processor executing a set of self-suspending tasks under the JSF must execute the *j*th subtask (free or embedded) of every task (i.e., $\tau_i^j, \forall\ i$) before any $(j+1)$th free subtask (i.e., τ_a^{j+1}). Furthermore, a processor does not idle if there is an available free subtask unless executing that free task results in temporal infeasibility due to a subtask-to-subtask deadline constraint.

Enforcing that all *j*th subtasks are completed before any $(j+1)$th-free subtasks allows the processor to execute any embedded *k*th subtasks where $k > j$ as necessary to ensure that subtask-to-subtask deadlines are satisfied. In other words, an embedded subtask τ_i^{j+1} may execute before all *j*th subtasks are executed, contingent on a temporal consistency check for subtask-to-subtask deadlines. The implication is that we cannot guarantee that embedded tasks (e.g., τ_i^j or τ_i^{j+1}) will be interleaved during their associated self-suspensions (e.g., $E_x^j, x \in N \setminus i$). The JSF priority scheduling policy offers a choice among consistency-checking algorithms. One simple algorithm that ensures deadlines are satisfied is as follows: when a free subtask that triggers a deadline constraint is executed (i.e., $\tau_i^j \in \boldsymbol{\tau}_{\text{free}}$, $\tau_i^{j+1} \in \boldsymbol{\tau}_{\text{embedded}}$), the subsequent embedded tasks for the associated deadline constraint are then scheduled as early as possible without the processor executing any other subtasks during this duration. Other consistency-check algorithms that use processor time more efficiently and operate on this structured task model exist [34–36].

VI. Uniprocessor Schedulability Test for Self-Suspending Task Sets

We build the schedulability test and prove its correctness in six steps, starting with a simplified task model and generalizing to the full model. Section VI.G then summarizes our test for the full task model. The six steps are as follows:

1) We restrict $\boldsymbol{\tau}$ such that each task only has two subtasks (i.e., $m_i = 2, \forall\ i$), there are no subtask-to-subtask deadlines, and all tasks are released at $t = 0$ (i.e., $\phi = 0, \forall\ i$). Additionally, we say that all tasks have the same period and deadline (i.e., $H = T_i = D_i = T_j = D_j, \forall\ i, j \in \{1, 2, \ldots, n\}$). Thus, the hyperperiod H of the task set, or the least common multiple of the set of task periods, is equal to the period of each task.

2) Next, we allow for general task release times (i.e., $\phi_i \geq 0, \forall\ i$). In this step, we upper bound the processor idle time due to phase offsets.

3) Third, we relax the restriction that each task has two subtasks and say that each task can have any number of subtasks.

4) Fourth, we incorporate subtask-to-subtask deadlines. In this step, we will describe how we calculate an upper bound on the processor idle time due to embedded self-suspensions.

5) Fifth, we relax the uniform task deadline restriction and allow for general task deadlines where $D_i \leq T_i, \forall\ i \in \{1, 2, \ldots, n\}$.

6) Lastly, we relax the uniform periodicity restriction and allow for general task periods where $T_i \neq T_j, \forall\ i, j \in \{1, 2, \ldots, n\}$.

A. Step 1: Two Subtasks Per Task, No Deadlines, and Zero Phase Offsets

In step 1, we consider a task set $\boldsymbol{\tau}$ with two subtasks per each of the n tasks, no subtask-to-subtask deadlines, and zero phase offsets (i.e., $\phi_i = 0, \forall\ i \in n$). Furthermore, task deadlines are equal to task periods, and all tasks have equal periods (i.e., $H = T_i = D_i = T_j = D_j, \forall\ i, j \in \{1, 2, \ldots, n\}$). We analyze the conditions under which a processor experiences idle times due to the set of self-suspensions $\{E_i^1 | i \in N\}$ in the restricted task set. We lay out a set of theorems and proofs that build to establishing an analytical upper bound on the idle time due to the set of self-suspensions $\{E_i^1 | i \in N\}$. Without loss of generality, we assume that the task index i in τ_i^j indicates the order of execution for first subtasks (i.e., τ_1^1 is processed first, τ_2^1 processed second, etc.).

Theorem VI.1: For a processor to experience idle time at

$$t = [\arg\max_{\tau_a^1} f_a^1, \arg\max_{E_j^1} f_j^1 + E_j^1]$$

then at least one self-suspension E_i^1 has at least $n-1$ subtasks to execute during its duration $[f_i^1, f_i^1 + E_i^1]$ and it subsumes idle time due to all other self-suspensions. We say that self-suspension E_n^m subsumes the idle time due to self-suspension E_q^r if the processor idle time during E_q^r is contained in the E_n^m interval $[f_n^m, f_n^m + E_n^m]$.

Proof 1 (proof by deduction for Theorem VI.1): All n first subtasks $\tau_i^1 | i \in N$ are released at $t = 0$, and the processor will not experience idle time at

$$t = [0, \arg\max_{\tau_a^1} f_a^1]$$

assuming $\phi_i = 0, \forall\ i \in n$. When the processor finishes τ_i^1 at $t = f_i^1$, there are at least $n - i$ released subtasks, $\{\tau_{i+1}^1, \tau_{i+2}^1, \ldots \tau_n^1\}$, that may be processed immediately. For the processor to experience idle time in the interval $[f_i^1, f_i^1 + E_i^1]$ that is not subsumed by any E_a^1 where $a < i$, then the finish time of E_i^1 must necessarily be greater than $t = f_a^1 + E_a^1$ for all $a < i$. Since τ_a^2 by definition becomes available for processing immediately after E_a^1, it follows that all τ_a^2 will become available for processing before $t = f_i^1 + E_i^1$. E_i^1 must then necessarily be greater than the sum of the cost of the next $n-1$ subtasks to execute, $\{\tau_{i+1}^1, \ldots \tau_n^1, \tau_1^2, \ldots, \tau_{i-1}^2\}$, if E_i^1 produces idle time not subsumed by another E_a^1 where $a < i$. Thus, it is necessary (but not sufficient) that the least $n-1$ subtasks execute during $[f_i^1, f_i^1 + E_i^1]$ for the processor to experience idle time during E_i^1 that is not subsumed by some E_a^1. Trivially, when $i = 1$ and $i = n$, the $n-1$ subtasks include $\{\tau_{i+1}^1, \ldots \tau_n^1\}$ and $\{\tau_1^2, \ldots, \tau_{i-1}^2\}$, respectively. Of the set of self-suspensions that have at least $n-1$ subtasks to execute during their duration, at least one self-suspension finishes last and subsumes the idle time due to all other self-suspensions in the set. □

Corollary VI.2 (processor idle time due to E_i^1): All processor idle time due to E_i^1, if E_i^1 is not subsumed by another self-suspension, is upper bounded by

$$\text{Idle}_{E_i^1} = \max\left(E_i^1 - \left(\left(\sum_{a=i+1}^{n} C_a^1\right) + \left(\sum_{a=1}^{i-1} C_a^2\right)\right), 0\right)$$

Corollary VI.3 (upper-bound processor idle time due to E_i^1): $\text{Idle}_{E_i^1}$ is upper bounded by

$$W_i^1 = E_i^1 - \sum_{k=1}^{n-1} B_i^1(k)$$

where B_i^1 is the set of of all subtasks that may execute during E_i^1, including $\{\tau_{i+1}^1, \ldots \tau_n^1, \tau_1^2, \ldots, \tau_{i-1}^2\}$; and $B_i^1(k)$ is the kth smallest-cost subtask in B_i^1.

Theorem VI.4: The processor idle time due to the set of self-suspensions $\{E_i^1 | i \in N\}$ is upper bounded by the maximum over $\{W_i^1 | i \in N\}$.

Proof 2: By Theorem VI.1, for a processor to experience idle time at

$$t = [\arg\max_{\tau_a^1} f_a^1, \arg\max_{E_j^1} f_j^1 + E_j^1]$$

then at least one self-suspension E_i^1 has at least $n-1$ subtasks to execute during its duration $[f_i^1, f_i^1 + E_i^1]$, and it subsumes idle time due to all other self-suspensions. By Corollary VI.3, W_i^1 upper bounds the processor idle time for each E_i^1 if E_i^1 is not subsumed by another self-suspension. Therefore, the maximum over $\{W_i^1 | i \in N\}$ provides the maximum idle time contributed by a self-suspension that subsumes all other self-suspensions. □

We mathematically represent Theorem VI.4 in Eqs. (3–6). We have only proved that these equations hold for the conditions in step 1 (i.e., $m_i = 2, \forall\ i, \phi_i = 0, \forall\ i, T_i = D_i = H, \forall\ i$, and with no subtask-to-subtask deadlines). We build to show that these equations hold more broadly in steps 2–6. B_i^j is the set of subtasks that could execute during E_i^j for it to create more idle time than any other self-suspension, and $B_i^j(k)$ is the kth-smallest-cost subtask in B_i^1. Then, η_i^1 is equal to our lower bound on the number of subtasks that would have to execute during E_i^1 for E_i^1 to create more idle time than any other self-suspension. Since there are no subtask-to-subtask deadlines and $m_i = 2, \forall\ i, |B_i^j| = 2(n-1)$. Thus, $|B_i^j|/2 = n - 1$. W_i^j is the upper bound on the processor idle time due to E_i^j, and W^j is the upper bound on the processor idle time due to $\{E_i^j | i \in N\}$:

$$B_i^j = \{C_a^j, C_a^{j+1} | a \in N, a \neq i, j+1 \leq m_a, \tau_a^j \in \tau_{\text{free}}, \tau_a^{j+1} \in \tau_{\text{free}}\} \tag{3}$$

$$\eta_i^1 = \frac{|B_i^j|}{2} \tag{4}$$

$$W_i^j = \max\left(\left(E_i^n - \sum_{k=1}^{\eta_i^j} B_i^j(k)\right), 0\right) \tag{5}$$

$$W_{\text{free}}^j = \max_{i \in N, E_i^j \in E_{\text{free}}} (W_i^j) \tag{6}$$

For an example for upper-bounding idle time due to $\{E_i^1 | i \in N\}$, we now consider three example task sets with three tasks: one where E_1^1 creates the most idle time, one where E_2^1 creates the most idle time, and one where E_3^1 creates the most idle time. We provide task set definitions in

Table 2 Example task set where E_1^1 creates more idle time than any other self-suspension, as shown in Fig. 4

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	B_i^1	η_i^1
$i=1$	0	1	12	2	$\{C_3^1, C_3^2, C_2^1, C_2^2\}$	2
$i=2$	0	2	4	4	$\{C_3^1, C_3^2, C_1^1, C_1^2\}$	2
$i=3$	0	1	1	1	$\{C_1^1, C_1^2, C_2^1, C_2^2\}$	2

Table 3 Example task set where E_2^1 creates more idle time than any other self-suspension, as shown in Fig. 5

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	B_i^1	η_i^1
$i=1$	0	1	5	2	$\{C_3^1, C_3^2, C_2^1, C_2^2\}$	2
$i=2$	0	2	7	4	$\{C_3^1, C_3^2, C_1^1, C_1^2\}$	2
$i=3$	0	1	4	1	$\{C_1^1, C_1^2, C_2^1, C_2^2\}$	2

Table 4 Example task set where E_3^1 creates more idle time than any other self-suspension, as shown in Fig. 6

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	B_i^1	η_i^1
$i=1$	0	1	5	2	$\{C_3^1, C_3^2, C_2^1, C_2^2\}$	2
$i=2$	0	2	7	4	$\{C_3^1, C_3^2, C_1^1, C_1^2\}$	2
$i=3$	0	1	11	1	$\{C_1^1, C_1^2, C_2^1, C_2^2\}$	2

Tables 2–4, illustrations in Figs. 4–6, and calculations in Eqs. (7–9), respectively. In all three examples, we can see that Eq. (6) correctly upper bounds the processor idle time due to the set of first self-suspensions $\{E_i^1 | 1 \le i \le n\}$. Specifically, $4 \le W^1 = 10$ (Table 2, Fig. 4), $3 \le W^1 = 5$ (Table 3, Fig. 5), and $5 \le W^1 = 8$ (Table 4, Fig. 6). Processor idle time is shown as a block without a number.

Example 1 in Fig. 4:

$$W_{\text{free}}^1 = \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_1^1, W_2^1, W_3^1) \quad = \max(10, 2, 0) \quad = 10 \tag{7}$$

Example 2 in Fig. 5:

$$W_{\text{free}}^1 = \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_1^1, W_2^1, W_3^1) \quad = \max(3, 5, 1) \quad = 5 \tag{8}$$

Example 3 in Fig. 6:

$$W_{\text{free}}^1 = \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_1^1, W_2^1, W_3^1) \quad = \max(3, 5, 8) \quad = 8 \tag{9}$$

B. Step 2: General Phase Offsets

Next, we allow for general task release times (i.e., $\phi_i \ge 0, \forall\ i$). Phase offsets may result in additional processor idle time. For example, if every task has a phase offset greater than zero, the processor is forced to idle at least until the first task is released. We also observe that, at the initial release of a task set, the largest phase offset of a task set will subsume the other phase offsets. We recall that the index i of the task τ_i corresponds to the ordering with which its first subtask is executed (i.e., $s_i^1 \le s_{i+1}^1$). We can therefore conservatively upper bound the idle time during $t = [0, f_n^1]$ due to the first instance of phase offsets by taking the maximum over all phase offsets, as shown in Eq. (10).

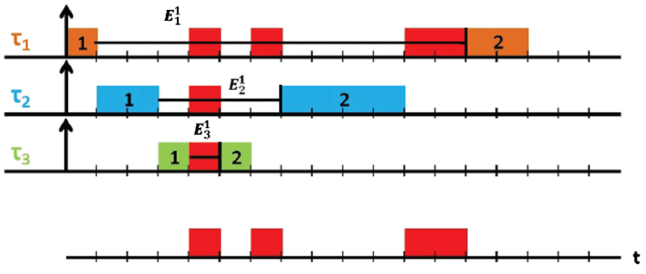


Fig. 4 Processor idle time due to E_1^1 is shown at the bottom.

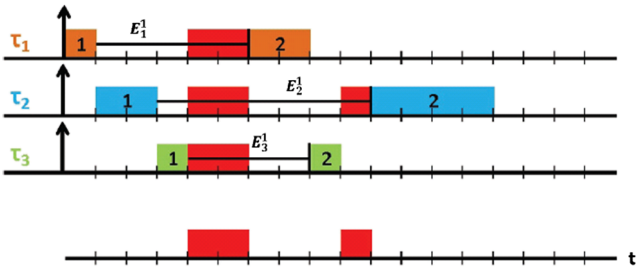


Fig. 5 Processor idle time due to E_2^1 is shown at the bottom.

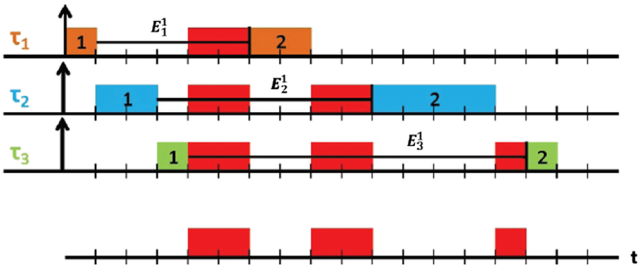


Fig. 6 Processor idle time due to E_3^1 is shown at the bottom.

The quantity W_ϕ^τ computed in step 2 is summed with W^1 [e.g., Eq. (8)], computed in step 1 to conservatively bound the contributions of the first self-suspensions and the first phase offsets to the processor idle time. This summation allows us to relax the assumption in step 1 that there is no processor idle time during the interval $t = [0, f_n^1]$:

$$W_\phi^\tau = \max_i \phi_i \tag{10}$$

For an example for Eq. (10), we extend example 2 from Fig. 5 to consider nonzero phase offsets. The new task set parameters are shown in Table 5, Fig. 7:

The upper bound on the processor idle time due to phase offsets is $W_\phi = 3$, as shown in Eq. (11):

$$W_\phi^\tau = \max_i \phi_i = \max\{0, 2, 3\} = 3 \tag{11}$$

C. Step 3: General Number of Subtasks Per Task

The next step in formulating our schedulability test is incorporating general numbers of subtasks in each task. As in step 1, our goal is to determine an upper bound on the processor idle time that results from the worst-case interleaving of the jth and $(j+1)$th subtask costs during the jth self-suspensions. Again, we recall that our formulation for upper-bounding idle time due to the first self-suspensions, in actuality, was an upper bound for the idle time during the interval $t = [f_n^1, \max_i(f_i^2)]$.

In step 2, we upper bounded the idle time resulting from phase offsets. To do this, we determined an upper bound on the idle time between the release of the first instance of each task at $t = 0$ and the finish of τ_n^1. Equivalently, this duration is $t = [0, \max_i(f_i^1)]$.

It follows then that, for each of the jth self-suspensions, we can apply Eq. (6) to determine an upper bound on the processor idle time during the interval $t = [\max_i(f_i^j), \max_i(f_i^{j+1})]$. The upper bound on the total processor idle time for all free self-suspensions in the task set is computed by summing over the contribution of each of the jth self-suspensions, as shown in Eq. (12):

$$W_{\text{free}}^\tau = \sum_j W^j \quad = \sum_j \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^j) \quad = \sum_j \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} \left(\max\left(\left(E_i^j - \sum_{k=1}^{\eta_i^j} B_i^j(k) \right), 0 \right) \right) \tag{12}$$

However, we need to be careful in the application of this equation for general task sets with unequal numbers of subtasks per task. Let us consider a scenario where one task, τ_i, has m_i subtasks, and τ_x has only $m_x = m_i - 1$ subtasks. When we upper bound the idle time due to the $(m_i - 1)$th self-suspensions, there is no corresponding subtask $\tau_x^{m_i}$ that could execute during $E_i^{m_i-1}$. We note that $\tau_x^{m_i-1}$ does exist and might execute during $E_i^{m_i-1}$, but we cannot guarantee that it does. Thus, when computing the set of subtasks, B_i^j, that may execute during a given self-suspension E_i^j, we only add a pair of subtasks τ_x^j, τ_x^{j+1} if both τ_x^j, τ_x^{j+1} exist, as described by Eq. (3). We note that, by inspection, if τ_x^j were to execute during E_i^j, it would only reduce processor idle time.

Table 5 Task set with three tasks and phase offsets

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	B_i^1	η_i^1
$i=1$	0	1	5	2	$\{C_3^1, C_3^2, C_2^1, C_2^2\}$	2
$i=2$	2	2	7	4	$\{C_3^1, C_3^2, C_1^1, C_1^2\}$	2
$i=3$	3	1	4	1	$\{C_1^1, C_1^2, C_2^1, C_2^2\}$	2

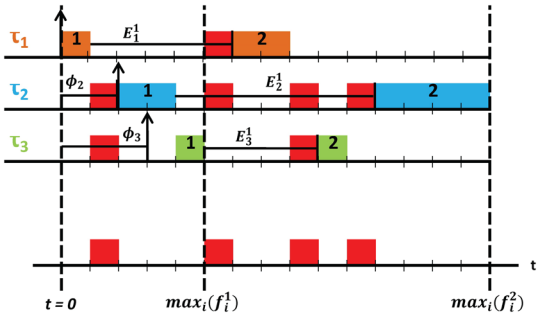


Fig. 7 Example schedule for three tasks with phase offsets.

For an example for Eq. (6), we extend our example from Fig. 7 to include multiple self-suspensions in each task. The new task set is shown in Table 6, Fig. 8.

To upper bound the processor idle time due to all self-suspensions, we first upper bound the processor idle time W^j for each of the jth self-suspensions $\{E_i^j | 1 \leq i \leq n\}$ using Eq. (6), as shown in Eqs. (13) and (14). Second, we apply Eq. (12) to the set of W^j terms to compute the total upper bound $W_{\text{free}}^{\boldsymbol{\tau}}$. For this example, $W_{\text{free}}^{\boldsymbol{\tau}} = 7$ [Eq. (15)]:

$$W^1 = \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_1^1, W_2^1, W_3^1) \quad = \max(3, 5, 1) \quad = 5 \tag{13}$$

$$W^2 = \max_{i|E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_1^2, W_2^2, W_3^2) \quad = \max(2, 2, 0) \quad = 2 \tag{14}$$

$$W_{\text{free}}^{\boldsymbol{\tau}} = \sum_j W^j = W^1 + W^2 = 5 + 2 = 7 \tag{15}$$

D. Step 4: Subtask-to-Subtask Deadline Constraints

In steps 1 and 3, we provided a lower bound for the number of free subtasks that will execute during a free self-suspension, if that self-suspension produces processor idle time. We then upper bounded the processor idle time due to the set of free self-suspensions by computing the least amount of free task costs that will execute during a given self-suspension. However, our proof assumed no subtask-to-subtask deadline constraints. Now, we relax this assumption and calculate an upper bound on the processor idle time due to embedded self-suspensions $W_{\text{embedded}}^{\boldsymbol{\tau}}$.

Recall that, under the JSF priority scheduling policy, an embedded subtask τ_i^{j+1} may execute before all jth subtasks are executed, contingent on a temporal consistency check for subtask-to-subtask deadlines. The implication is that we cannot guarantee that embedded tasks (e.g., τ_i^j or τ_i^{j+1}) will be interleaved during their associated self-suspensions (e.g., E_x^j, $x \in N \setminus i$).

To account for this lack of certainty, we conservatively treat embedded self-suspensions as task costs, as shown in Eqs. (16) and (17). Equation (16) requires that, if a self-suspension E_i^j is free, then $E_i^j(1 - x_i^{j+1}) = 0$. The formula $(1 - x_i^{j+1})$ is used to restrict our sum to only include embedded self-suspensions. Recall that a self-suspension E_i^j is embedded if τ_i^{j+1} is an embedded subtask.

Second, we restrict B_i^j such that the jth and $(j + 1)$th subtasks must be free subtasks if either is to be added. We specified this constraint in step 1, but this restriction did not have an effect because we were considering task sets without subtask-to-subtask deadlines.

Third, we now must consider cases where $\eta_i^j < n - 1$, as described in Eq. (4). We recall that $\eta_i^j = n - 1$ if there are no subtask-to-subtask deadlines; however, with the introduction of these deadline constraints, we can only guarantee that at least $\eta_i^j = |B_i^j|/2$ subtasks will execute during a given E_i^j if E_i^j results in processor idle time:

$$W_{\text{embedded}}^{\boldsymbol{\tau}} = \sum_{i=1}^{n} \left(\sum_{j=1}^{m_i - 1} E_i^j (1 - x_i^{j+1}) \right) \tag{16}$$

$$x_i^j = \begin{cases} 1, & \text{if } \tau_i^j \in \boldsymbol{\tau}_{\text{free}} \\ 0, & \text{if } \tau_i^j \in \boldsymbol{\tau}_{\text{embedded}} \end{cases} \tag{17}$$

Having bounded the amount of processor idle time due to free and embedded self-suspensions and phase offsets, we now provide an upper bound on the time H_{UB}^{τ} the processor will take to complete all instances of each task in the hyperperiod [Eq. (18)]. H denotes the hyperperiod of the task set, and H_{LB}^{τ} is defined as the sum over all task costs released during the hyperperiod. Recall that we are still assuming that $T_i = D_i = T_j = D_j$,

Table 6 Task set with phase offsets and multiple self-suspensions per task

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	E_i^2	C_i^3	E_i^3	C_i^4
$i = 1$	0	1	5	2	5	2	1	1
$i = 2$	2	2	7	4	5	2	0	0
$i = 3$	3	1	4	1	2	2	0	0

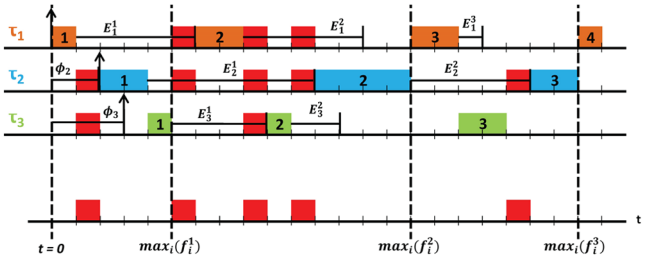


Fig. 8 Example schedule for three tasks with phase offsets and multiple self-suspensions.

$\forall\ i, j \in N$; thus, there is only one instance of each task in the hyperperiod. Under this assumption, the task set is schedulable under JSF if $H^{\boldsymbol{\tau}}_{\text{UB}}/H \leq 1$:

$$H^{\boldsymbol{\tau}}_{\text{UB}} = H^{\boldsymbol{\tau}}_{\text{LB}} + W^{\boldsymbol{\tau}}_{\text{phase}} + W^{\boldsymbol{\tau}}_{\text{free}} + W^{\boldsymbol{\tau}}_{\text{embedded}} \tag{18}$$

$$H^{\tau}_{LB} = \sum_{i=1}^{n} \frac{H}{T_i} \sum_{j=1}^{m_i} C_i^j \tag{19}$$

For an example for subtask-to-subtask deadline constraints, consider our example from Fig. 8, which is now augmented to include a subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle} = 9$$

as shown in Table 7, Fig. 9.

We apply Eq. (16) to our example to upper bound the processor idle time due to all embedded self-suspensions. In this case, there is only one embedded self-suspension, E_1^2; thus, the upper bound on processor idle time due to embedded self-suspensions is $W^{\tau}_{\text{embedded}} = 5$ [Eq. (20)]:

$$W^{\boldsymbol{\tau}}_{\text{embedded}} = \sum_{i=1}^{n} \left(\sum_{j=1}^{m_i - 1} E_i^j (1 - x_i^{j+1}) \right) \quad = E_1^2 \quad = 5 \tag{20}$$

Because of the addition of this subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$$

the upper bound for $W^{\boldsymbol{\tau}}_{\text{free}}$ must be recomputed. The deadline

$$D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$$

embeds just one of the second self-suspensions $\{E_i^2 | 1 \leq i \leq n\}$, so we only need to recompute W^2; W^1 is unchanged.

Recall that W^j is the maximum over all $\{W_i^j | 1 \leq i \leq n\}$, where each associated self-suspension E_i^j is a free self-suspension. Because E_1^2 is embedded, we only need to calculate W_2^2 [Eq. (21)] and W_3^2 [Eq. (22)]:

$$W_2^2 = \max\left(\left(E_2^2 - \sum_{k=1}^{\eta_2^2} B_2^2(k) \right), 0 \right) \quad = \max((5 - (1))), 0) \quad = 4 \tag{21}$$

$$W_3^2 = \max\left(\left(E_3^2 - \sum_{k=1}^{\eta_3^2} B_3^2(k) \right), 0 \right) \quad = \max((2 - (1))), 0) \quad = 1 \tag{22}$$

The new upper bound for the idle time due to free self-suspensions is now calculated as shown in Eqs. (23) and (24):

$$W^2 = \max_{i | E_i^j \in \boldsymbol{E}_{\text{free}}} (W_i^1) \quad = \max(W_2^2, W_3^2) \quad = \max(4, 1) \quad = 4 \tag{23}$$

$$W^{\boldsymbol{\tau}}_{\text{free}} = \sum_j W^j = W^1 + W^2 = 5 + 4 = 9 \tag{24}$$

Finally, the upper bound $H^{\boldsymbol{\tau}}_{\text{UB}}$ on the time required to process $\boldsymbol{\tau}$ can be computed via Eq. (18). For our example, $H^{\boldsymbol{\tau}}_{\text{UB}} = 35$ [Eq. (25)]. This upper bound guarantees that this task set can be processed if the hyperperiod $H = T_i = T_j$ of the task set is greater than or equal to $H^{\boldsymbol{\tau}}_{\text{UB}} = 35$:

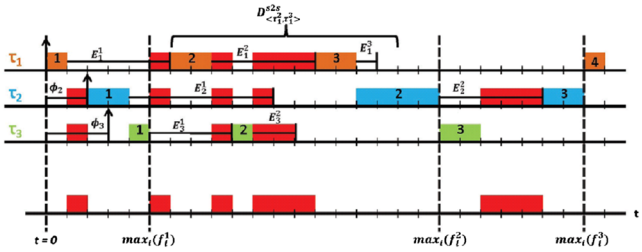


Fig. 9 Example schedule for three tasks with a subtask-to-subtask deadline constraint $D^{s2s}_{\langle\tau_1^2,\tau_1^3\rangle}$.

$$H^{\boldsymbol{\tau}}_{\text{UB}} = H^{\boldsymbol{\tau}}_{\text{LB}} + W^{\boldsymbol{\tau}}_{\text{phase}} + W^{\boldsymbol{\tau}}_{\text{free}} + W^{\boldsymbol{\tau}}_{\text{embedded}} \quad = 18 + 3 + 9 + 5 \quad = 35 \tag{25}$$

E. Step 5: Deadlines Less Than or Equal to Periods

Next, we allow for tasks to have deadlines less than or equal to the period. We recall that we still restrict the periods such that $T_i = T_a, \forall\ i, a \in N$ for this step. When we formulated our schedulability test of a self-suspending task set in Eq. (18), we calculated an upper bound on the time the processor needs to execute the task set $H^{\boldsymbol{\tau}}_{\text{UB}}$. Now, we seek to upper bound the amount of time required to execute the final subtask $\tau_i^{m_i}$ for task τ_i, and we can use the methods already developed to upper bound this time. We consider the largest subset of subtasks $\boldsymbol{\tau}|_j \subset \boldsymbol{\tau}$ for $j = m_i$ that might execute before the deadline for τ_i. If we find that $H^{\boldsymbol{\tau}|_j}_{\text{UB}} \leq d$, where $d = \phi_i + D_i$ and $j = m_i$, then we know that a processor scheduling under JSF will satisfy the task deadline for τ_i.

We present an algorithm named testDeadline($\boldsymbol{\tau}$, d, j) to perform this test. The pseudocode for testDeadline($\boldsymbol{\tau}$, d, j) is shown in Fig. 10. This algorithm requires as input a task set $\boldsymbol{\tau}$, a deadline $d = D_i + \phi$ for τ_i, and the subtask index $j = m_i$ corresponding to the final subtask in τ_i. The algorithm returns true if a guarantee can be provided that the processor will satisfy D_i under the JSF, and it returns false otherwise.

In lines 1–14, the algorithm computes $\boldsymbol{\tau}|_j$, which is the set of subtasks that may execute before d. In the absence of subtask-to-subtask deadline constraints, $\boldsymbol{\tau}|_j$ includes all subtasks τ_x^y where $x \in N$ and $y \in \{1, 2, \ldots, m_i\}$. In the case where a subtask-to-subtask deadline

$$D^{s2s}_{\langle\tau_x^a,\tau_x^b\rangle}$$

spans subtask $\tau_x^{m_i}$, where $a \leq m_i < b$, the processor may be required to execute all embedded subtasks $\tau_x^y | a < y \leq b$ associated with the deadline before executing the final subtask for task τ_i. Therefore, the embedded subtasks $\tau_x^y | a < y \leq b$ of

$$D^{s2s}_{\langle\tau_x^a,\tau_x^b\rangle}$$

are also added to the set $\boldsymbol{\tau}|_j$. In line 15, the algorithm tests the schedulability of $\boldsymbol{\tau}|_j$ using Eq. (18).

Next, we walk through the pseudocode for testDeadline($\boldsymbol{\tau}$, d, j) in detail. Line 1 initializes $\tau|_j$. Line 2 iterates over each task τ_x in $\boldsymbol{\tau}$. Line 3 initializes the index of the last subtask from τ_x that may need to execute before τ_i^j as $z \leftarrow j$, assuming no subtask-to-subtask constraints.

Lines 5–11 search for additional subtasks that may need to execute before τ_i^j due to subtask-to-subtask deadlines. If the next subtask, τ_x^{z+1}, does not exist, then τ_x^z is the last subtask that may need to execute before τ_i^j (lines 5–6). The same is true if $\tau_x^{z+1} \in \boldsymbol{\tau}_{\text{free}}$, because τ_x^{z+1} will not execute before $\tau_i^{m_i}$ under JSF if $z + 1 > j$ (lines 7–8). If τ_x^{z+1} is an embedded subtask, then it may be executed before τ_i^j, so we increment z, the index of the last subtask, by one (lines 9–10). Finally, line 13 adds the subtasks collected for τ_x, denoted $\tau_x|_j$, to the task subset $\tau|_j$.

After constructing our subset $\tau|_j$, we compute an upper bound on the fraction of time required by the processor to satisfy the deadline d (line 15). If this fraction is less than or equal to one, then we can guarantee that the deadline will be satisfied by a processor scheduling under JSF (line 16). Otherwise, we cannot guarantee the deadline will be satisfied and return false (line 18). To determine if all task deadlines are satisfied, we call testDeadline($\boldsymbol{\tau}$, d, j) once for each task deadline.

F. Step 6: General Periods

Thus far, we have established a mechanism for testing the schedulability of a self-suspending task set with general task deadlines less than or equal to the period, general numbers of subtasks in each task, nonzero phase offsets, and subtask-to-subtask deadlines. We now relax the restriction that $T_i = T_j, \forall\ i, j$. The principle challenge of relaxing this restriction is there will be any number of task instances in a hyperperiod, whereas before, each task only had one instance.

To determine the schedulability of the task set, we first start by defining a task superset $\boldsymbol{\tau}^*$, where $\boldsymbol{\tau}^* \boldsymbol{\tau}$. This superset has the same number of tasks as $\boldsymbol{\tau}$ (i.e., n), but each task $\tau_i^* \in \boldsymbol{\tau}^*$ is composed of H/T_i instances of $\tau_i \in \boldsymbol{\tau}$. A formal definition is shown in Eq. (26), where $C^j_{i,k}$ and $E^j_{i,k}$ are the kth instance of the jth subtask cost and self-suspension of τ_i^*:

$$\tau_i^*\text{: } (\phi_i, (C^1_{i,1}, E^1_{i,1}, \ldots, C^{m_i}_{i,1}, C^1_{i,2}, E^1_{i,2}, \ldots, C^{m_i}_{i,2}, \ldots, C^1_{i,k}, E^1_{i,k}, \ldots, C^{m_i}_{i,k}), D^*_i = H, T^*_i = H) \tag{26}$$

We aim to devise a test where τ_i^* is schedulable if $H^{\boldsymbol{\tau}^*}_{\text{UB}}/D^*_i \leq 1$ and if the task deadline D_i of each instance of τ_i is satisfied. This requires three steps. First, we must perform a mapping of subtasks from $\boldsymbol{\tau}$ to $\boldsymbol{\tau}^*$ that guarantees that τ_i^{j+1*} will be released by the completion time of the set of all

Table 7 Task set with phase offsets, multiple self-suspension per task, and a subtask-to-subtask deadline

τ_i	ϕ_i	C_i^1	E_i^1	C_i^2	E_i^2	C_i^3	E_i^3	C_i^4
$i = 1$	0	1	5	2	5	2	1	1
$i = 2$	2	2	7	4	5	2	0	0
$i = 3$	3	1	4	1	2	2	0	0

```
testDeadline(τ,d,j)
 1: τ|_j ← NULL
 2: for x = 1 to |τ|, do
 3:     z ← m_i
 4:     while TRUE, do
 5:         if τ_x^{z+1} ∉ (τ_free ∪ τ_embedded), then
 6:             break
 7:         else if τ_x^{z+1} ∈ τ_free, then
 8:             break
 9:         else if τ_x^{z+1} ∈ τ_embedded, then
10:             z ← z + 1
11:         end if
12:     end while
13:     τ_x|_j ← (φ_x, (C_x^1, E_x^1, C_x^2, ..., C_x^z), D_x, T_x)
14: end for
15: if H_UB^{τ|_j}/d ≤ 1 //Using Eq. (18), then
16:     return TRUE
17: else
18:     return FALSE
19: end if
```

Fig. 10 Pseudocode for testDeadline(τ,d,j), which tests whether a processor scheduling under JSF is guaranteed to satisfy a task deadline D_i.

other jth subtasks $\{\tau_a^{j*} | a \in N, \tau_a^{j*} \in \boldsymbol{\tau}^*\}$. JSF requires that all jth subtasks be executed before starting any $(j+1)$th free subtask. Consider the case where a jth subtask in $\boldsymbol{\tau}^*$ corresponds to the final subtask of a task instance. The processor may have to idle until the release of the next instance of that task before it can begin any $(j+1)$th subtask in $\boldsymbol{\tau}^*$. Thus, we would like to shift the index of each subtask in subsequent instances to some $j' \geq j$ such that we can guarantee the subtask τ_i^{j+1*} will be released by the completion time of all τ_i^{j*}, $\forall\ i$ subtasks.

Second, we need to check that each task deadline D_i for each instance k of each task τ_i released during the hyperperiod will be satisfied. To perform this check, we compose a paired list $D[i][k]$ to keep track of the subtasks in $\boldsymbol{\tau}^*$ that correspond to the last subtasks of each instance k of a task τ_i. $D[i][k]$ returns the subtask index j in $\boldsymbol{\tau}^*$ of instance k of τ_i. Finally, we must determine an upper bound $H_{\text{UB}}^{\boldsymbol{\tau}^*}$ on the temporal resources required to execute $\boldsymbol{\tau}^*$ using Eq. (18). If $H_{\text{UB}}^{\boldsymbol{\tau}^*}/H \leq 1$, where H is the hyperperiod of $\boldsymbol{\tau}$, then the task set is schedulable under JSF.

We use an algorithm called constructTaskSuperSet($\boldsymbol{\tau}$), presented in Fig. 11, to construct our task superset $\boldsymbol{\tau}^*$. The function constructTaskSuperSet($\boldsymbol{\tau}$) takes as input a self-suspending task set $\boldsymbol{\tau}$ and returns either the superset $\boldsymbol{\tau}^*$ if we can construct the superset or null if we cannot guarantee that the deadlines for all task instances released during the hyperperiod will be satisfied.

In line 1, we initialize our task superset $\boldsymbol{\tau}^*$ to include the subtask costs, self-suspensions, phase offsets, and subtask-to-subtask deadlines of the first instance of each task τ_i in $\boldsymbol{\tau}$. In line 2, we initialize a vector I, where $I[i]$ corresponds to the instance number of the last instance of τ_i that we have added to $\boldsymbol{\tau}^*$. Note that, after initialization, $I[i] = 1$ for all i. In line 3, we initialize a vector J, where $J[i]$ corresponds to the j subtask index of τ_i^{j*}: for instance $I[i]$, the last task instance added to τ_i^*. The mapping to new subtask indices is constructed in J to ensure that the $(j+1)$th subtasks in $\boldsymbol{\tau}^*$ will be released by the time the processor finishes executing the set of jth subtasks. In line 4, $D[i][k]$ is initialized to the subtask indices associated with the first instance of each task.

In line 5, we initialize counter, which we use to iterate through each j subtask index in $\boldsymbol{\tau}^*$. In line 6 we initialize H_{LB} to zero. H_{LB} will be used to determine whether we can guarantee that a task instance in $\boldsymbol{\tau}$ has been released by the time the processor finishes executing the set of $j = \text{counter} - 1$ subtasks in $\boldsymbol{\tau}^*$.

Next, we compute the mapping of subtask indices for each of the remaining task instances released during the hyperperiod (lines 7–29). In line 11, we increment H_{LB} by the sum of the costs of the set of the $j = \text{counter} - 1$ subtasks. In line 12, we iterate over each task τ_i^*. First, we check if there is a remaining instance of τ_i to add to τ_i^* (line 13). If so, we then check whether counter $> J[i]$ (i.e., the current jth subtask index, where $j = \text{counter}$, is greater than the index of the last subtask we added to τ_i^*) (line 14).

If the two conditions in lines 13 and 14 are satisfied, we test whether we can guarantee the first subtask of the next instance of τ_i will be released by the completion of the set of the $j = \text{counter} - 1$ subtasks in $\boldsymbol{\tau}^*$ (line 15). We recall that, under JSF, the processor executes all $j-1$ subtasks before executing a jth free subtask and, by definition, the first subtask in any task instance is always free. The release time of the next instance of τ_i is given by $T_i * [i] + \phi_i$. Therefore, if the sum of the costs of all subtasks with index $j \in \{1, 2, \ldots, \text{counter} - 1\}$ is greater than the release time of the next task instance, then we can guarantee the next task instance will be released by the time the processor finishes executing the set of $j = \text{counter} - 1$ subtasks in $\boldsymbol{\tau}^*$.

We can therefore map the indices of the subtasks of the next instance of τ_i to subtask indices in τ_i^* with $j = \text{counter} + y - 1$, where y is the subtask index of τ_i^y in τ_i. Thus, we increment $I[i]$ to indicate that we are considering the next instance of τ_i (line 16) and add the next instance of τ_i, including subtask costs, self-suspensions, and subtask-to-subtask deadlines, to τ_i^* (line 17). Next, we set $J[i]$ and $D[i][k]$ to the j subtask index of the subtask we last added to τ_i^* (lines 18–19). We will use $D[i][k]$ later to test the task deadlines of the task instances we add to τ_i^*.

In the case where all subtasks of all task instances up to instance $I[i]$, $\forall\ i$ are guaranteed to complete before the next scheduled release of any task in $\boldsymbol{\tau}$ (i.e., there are no subtasks to execute at $j = \text{counter}$), then counter is not incremented and H_{LB} is set to the earliest next release time of any task instance (lines 24 and 25). Otherwise, the counter used to iterate through each j subtask index in $\boldsymbol{\tau}^*$ is incremented. The mapping of subtasks from $\boldsymbol{\tau}$ to $\boldsymbol{\tau}^*$ continues until all remaining task instances released during the hyperperiod are processed. Finally, lines 31–39 ensure that the superset exists if each task deadline $D_{i,k}$ for each instance k of each task τ_i released during the hyperperiod is guaranteed to be satisfied.

G. Schedulability Test Summary

To determine the schedulability of task set $\boldsymbol{\tau}$ we call constructTaskSuperSet($\boldsymbol{\tau}$) on $\boldsymbol{\tau}$. This function tests the schedulability of $\boldsymbol{\tau}$ by computing an upper bound $H_{\text{UB}}^{\boldsymbol{\tau}}$ on the time required to process the task (or subtask) using Eq. (18).

$H_{\text{UB}}^{\boldsymbol{\tau}}$ comprises four terms. The first term $H_{\text{LB}}^{\boldsymbol{\tau}}$ is simply the sum over the cost of the tasks [Eq. (19)]. The next three terms upper bound the amount of the processor idle time due to phase offsets, as well as free and embedded self-suspensions. $W_{\phi}^{\boldsymbol{\tau}}$ [Eq. (10)] accounts for processor idle

```
constructTaskSuperSet(τ)
 1: τ* ← Initialize to τ
 2: I[i] ← 1, ∀i ∈ N
 3: J[i] ← m_i, ∀i ∈ N
 4: D[i][k] ← m_i, ∀i ∈ N, k = 1
 5: counter ← 2
 6: H_LB ← 0
 7: while TRUE, do
 8:     if I[i] = H/T_i, ∀i ∈ N, then
 9:         break
10:     end if
11:     H_LB ← H_LB + Σ_{i=1}^{n} C_i^{*(counter−1)}
12:     for i = 1 to n, do
13:         if I[i] < H/T_i, then
14:             if counter > J[i], then
15:                 if H_LB ≥ T_i*I[i]+φ_i, then
16:                     I[i] ←I[i]+1
17:                     τ_i^{*(counter+y−1)} ← τ_i^y, ∀y ∈ {1, 2, ..., m_i}
18:                     J[i] = counter + m_i − 1
19:                     D[i][I[i]] ← J[i]
20:                 end if
21:             end if
22:         end if
23:     end for
24:     if counter > max_i J[i], then
25:         H_LB = min_i (T_i * I[i] + φ_i)
26:     else
27:         counter ← counter +1
28:     end if
29: end while
30: //Test Task Deadlines for Each Instance
31: for i = 1 to n, do
32:     for k = 1 to H/T_i, do
33:         D_{i,k} ← D_i + T_i(k − 1) + φ_i
34:         j ← D[i][k]
35:         if testDeadline(τ*, D_{i,k}, j) = FALSE, then
36:             return NULL
37:         end if
38:     end for
39: end for
40: return τ*
```

Fig. 11 Pseudocode for constructTaskSuperSet(τ), which constructs a task superset τ^* for τ.

time due to phase offsets and equals the maximum over all phase offsets. W^{τ}_{free} upper bounds the processor idle time due to free self-suspensions by considering the worst-case interleaving of subtasks during free self-suspensions [Eq. (12)]. Lastly, $W^{\tau}_{\text{embedded}}$ upper bounds the processor idle time due to self-suspensions that are constrained by subtask-to-subtask deadlines [Eq. (16)].

If the schedulability test determines that the processor can schedule $\boldsymbol{\tau}$ under JSF, then we process $\boldsymbol{\tau}$. In addition to testing the schedulability of $\boldsymbol{\tau}$, constructTaskSuperSet($\boldsymbol{\tau}$) returns a super task set $\boldsymbol{\tau}$ consisting of all instances of tasks in $\boldsymbol{\tau}$ released during the hyperperiod. Then, constructTaskSuperSet($\boldsymbol{\tau}$) constructs $\boldsymbol{\tau}^*$ in a careful way, such that the processor will schedule $\boldsymbol{\tau}$ according to JSF using jth indices of subtasks, as specified in $\boldsymbol{\tau}^*$.

VII. Uniprocessor Scheduling Algorithm for Self-Suspending Task Sets

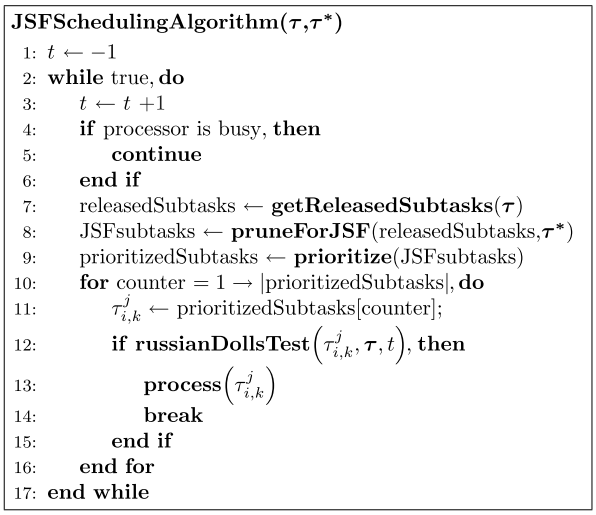
In Sec. VI, we developed a uniprocessor schedulability test for hard nonpreemptive self-suspending task sets. This schedulability test relies on a processor operating using the jth-subtask-first scheduling priority. JSF requires that all jth subtasks are processed before any $(j+1)$th free subtasks, where a subtask τ_i^{j+1} is free if it does not share a deadline constraint with subtask τ_i^j. In computing the analytical schedulability test, we assume that the processor idles during the embedded self-suspensions. We now describe our JSF scheduling algorithm, which uses an online schedulability test to execute subtasks during embedded self-suspensions, and thus better uses the processor.

A. Scheduling Algorithm Pseudocode

The JSF scheduling algorithm takes as input a self-suspending task set $\boldsymbol{\tau}$ and the super set $\boldsymbol{\tau}^*$ generated by constructTaskSuperSet($\boldsymbol{\tau}$). The algorithm processes instances of $\boldsymbol{\tau}$ until terminated by the system. Recall that $\boldsymbol{\tau}^*$ is a special task set that contains H/T_i instances of each task τ_i, where H is the hyperperiod of task set $\boldsymbol{\tau}$. JSF prioritizes subtask τ_i^j according to its j index in $\boldsymbol{\tau}^*$.

The pseudocode for the JSF scheduling algorithm is shown in Fig. 12. In line 1, we initialize our clock. Line 2 sets the algorithm up to indefinitely process released subtasks. In line 3, we increment our clock. In line 4, we check if the processor is busy processing a subtask. If so, we wait until the next clock step (line 5). If our processor is available to process a new subtask, we first collect all released subtasks (line 7).

Next, the scheduling algorithm prunes this list of subtasks according to JSF. As an example, consider two released subtasks, $\tau_{i,a}^j$ and $\tau_{x,b}^y$, for instances a and b of τ_i and τ_x, respectively, at time t. There are corresponding subtasks τ_i^{k*} and τ_x^{z*} in $\boldsymbol{\tau}^*$ such that $j \le k$ and $y \le z$. Recall that, if $j < y$ and τ_x^y is a free subtask, then τ_x^y is not considered for execution at time t according to JSF prioritization. Line 8 prunes all such released subtasks $\tau_{x,b}^y$.



JSFSchedulingAlgorithm($\boldsymbol{\tau}$,$\boldsymbol{\tau}^*$)

1: $t \leftarrow -1$
2: **while** true, **do**
3: $t \leftarrow t + 1$
4: **if** processor is busy, **then**
5: **continue**
6: **end if**
7: releasedSubtasks $\leftarrow$ **getReleasedSubtasks**($\boldsymbol{\tau}$)
8: JSFsubtasks $\leftarrow$ **pruneForJSF**(releasedSubtasks,$\boldsymbol{\tau}^*$)
9: prioritizedSubtasks $\leftarrow$ **prioritize**(JSFsubtasks)
10: **for** counter $= 1 \rightarrow$ |prioritizedSubtasks|, **do**
11: $\tau_{i,k}^j \leftarrow$ prioritizedSubtasks[counter];
12: **if russianDollsTest**$\left(\tau_{i,k}^j, \boldsymbol{\tau}, t\right)$, **then**
13: **process**$\left(\tau_{i,k}^j\right)$
14: **break**
15: **end if**
16: **end for**
17: **end while**

Fig. 12 Pseudocode for JSFSchedulingAlgorithm(τ, τ^*). This algorithm schedules self-suspending task sets on a uniprocessor.

Line 9 prioritizes the remaining, released subtasks according to an application-specific priority. Because JSF sets the same priority for subtasks $\tau_{i,a}^{j*}$ and $\tau_{\alpha,\beta}^{\gamma*}$ if $j = \gamma$, then there is room to further prioritize within JSF. For now, we assume that such subtasks are prioritized according to the earliest deadline first algorithm.

Line 10 iterates over all the released, prioritized subtasks allowed by JSF to be processed at time t. In line 11, the algorithm stores the next subtask to consider processing $\tau_{i,k}^j$. In line 12, a novel online consistency test, called the Russian dolls test, determines whether scheduling $\tau_{i,k}^j$ at time t may result in a subtask missing a deadline. We describe this test in Sec. VII.B. If our online consistency test guarantees that processing $\tau_{i,k}^j$ at time t will not result in a subtask missing its deadline, then the algorithm schedules $\tau_{i,k}^j$ on the processor.

B. Online Schedulability Test

The uniprocessor Russian dolls test is a schedulability test for ensuring feasibility while scheduling tasks against subtask-to-subtask deadline constraints. The test is a variant of the resource edge-finding algorithm [35,36], the purpose of which is to determine whether an event must or may execute before or after a set of activities [37]. Our analytical polynomial-time approach determines whether a subtask τ_i^j can feasibly execute before a set of other subtasks given the set of subtask-to-subtask deadline constraints. To our knowledge, our approach is the first to leverage the structure of the self-suspending task model to perform fast edge checking.

To describe our test, we first define an active subtask-to-subtask deadline (Definition 6) and an active subtask (Definition 7).

Definition 6 (active subtask-to-subtask deadline): A subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}$$

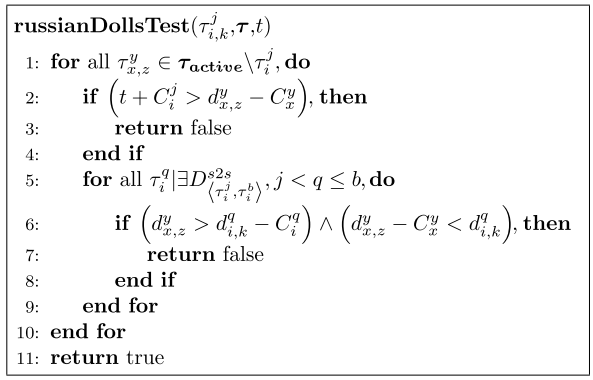
is considered active at $s_i^j \le t \le f_i^b$.

Definition 7 (active subtask): A subtask is active at time t if it has been released, is yet unprocessed at time t, and is directly constrained by an active subtask-to-subtask deadline.

1. *Walkthrough of Pseudocode*

The pseudocode describing the uniprocessor Russian dolls test is shown in Fig. 13. The Russian dolls test takes as input a subtask τ_i^j, the task set $\boldsymbol{\tau}$, and the current time t. The Russian dolls test returns whether we can guarantee that processing τ_i^j at time t will not result in another subtask violating its subtask-to-subtask deadline constraint.

To determine the feasibility of scheduling $\tau_{i,k}^j$ at time t, we must consider two scenarios. First, if processing $\tau_{i,k}^j$ does not activate a subtask-to-subtask deadline, then we merely need to guarantee that processing $\tau_{i,k}^j$ leaves enough time for the processor to finish executing the set of active subtasks. Second, if processing $\tau_{i,k}^j$ does activate a subtask-to-subtask deadline



russianDollsTest($\tau_{i,k}^j$,$\boldsymbol{\tau}$,t)

1: **for** all $\tau_{x,z}^y \in \boldsymbol{\tau}_{\mathbf{active}} \backslash \tau_i^j$, **do**
2: **if** $\left(t + C_i^j > d_{x,z}^y - C_x^y\right)$, **then**
3: **return** false
4: **end if**
5: **for** all $\tau_i^q | \exists D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}, j < q \le b$, **do**
6: **if** $\left(d_{x,z}^y > d_{i,k}^q - C_i^q\right) \wedge \left(d_{x,z}^y - C_x^y < d_{i,k}^q\right)$, **then**
7: **return** false
8: **end if**
9: **end for**
10: **end for**
11: **return** true

Fig. 13 Pseudocode describing the uniprocessor Russian dolls test.

$$D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}$$

then we must also consider whether the processor will have enough time to attend to subtasks $\{\tau_{i,k}^q | j < q \leq b\}$ in addition to the other active subtasks.

In line 1, the test iterates over all active subtasks $\tau_{x,z}^y$ (Definition 7), not including $\tau_{i,k}^j$. In lines 2–4, the test considers the direct effect of processing $\tau_{i,k}^j$ at time t. Line 2 tests whether the processor can nest the execution of $\tau_{i,k}^j$ within the laxity of the $\tau_{x,z}^y$ implicit deadline $d_{x,z}^y$. The implicit deadline $d_{x,z}^y$ is the latest-allowable finish time for $\tau_{x,z}^y$, assuming that $\tau_{i,k}^j$ will be executed at time t. The laxity of $\tau_{x,z}^y$ is the difference between $d_{x,z}^y$ and $C_{x,z}^y$ as defined by Dertouzos and Mok [38]. If no such nesting is possible, then the test returns false, thus prohibiting the processing of $\tau_{i,k}^j$ at time t (line 3).

If scheduling $\tau_{i,k}^j$ at time t would activate a subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_i^j, \tau_a^b \rangle}$$

(Definition 6), then we must consider the indirect effects of this activation on the other subtasks constrained by this deadline constraint. If this activation would occur, the test iterates over all subtasks $\tau_{i,k}^q | j < q \leq b$ constrained by any

$$D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}$$

(line 5), except for $\tau_{i,k}^j$, which is accounted for in line 2.

We then determine whether the processor can nest the execution of $\tau_{i,k}^q$ within the laxity of the $\tau_{x,z}^y$ implicit deadline or vice versa (line 6). Here again, the implicit deadline is computed as the latest-allowable finish time for $\tau_{x,z}^y$, assuming that $\tau_{i,k}^j$ will be executed at time t. If the nesting is not feasible, then the test returns false, indicating that there is no guarantee that the processor will satisfy all subtask-to-subtask deadline constraints if $\tau_{i,k}^j$ is processed at time t (line 7). If this nesting can be performed for all such pairs of subtasks, then the test returns true, indicating that $\tau_{i,k}^j$ can safely be processed at time t (line 10).

2. *Proof of Correctness of the Uniprocessor Russian Dolls Test*

The problem of sequencing a self-suspending task set with upper- and lower-bound temporal constraints is known to be NP hard [8,21,22]. The uniprocessor Russian dolls test is polynomial in time complexity, since it only performs pairwise comparisons between active subtasks and the subtasks that share a subtask-to-subtask deadline with $\tau_{i,k}^j$, and it is therefore not a complete schedulability test. In this section, we prove that the algorithm is nonetheless correct, in that only pairwise comparisons of subtasks are necessary to ensure schedule feasibility.

The test leverages the problem structure inherent in the augmented self-suspending task model to perform efficient computation. Recall that the self-suspending task model requires that subtasks within a task are executed in order, meaning that a subtask τ_i^{j+1} must execute after subtask τ_i^j and before subtask τ_i^{j+2}. This problem structure restricts the interaction of deadlines in the task set.

Theorem VII.1: The Russian dolls test is correct, in that it requires only a pairwise comparison between active subtasks $\boldsymbol{\tau}_{\text{active}}$ and subtasks that share a subtask-to-subtask deadline with τ_i^j (i.e., $\{\tau_i^b | \exists D^{s2s}_{\langle \tau_i^j, \tau_i^z \rangle}, j \leq b \leq z\}$) when testing the schedulability of $\tau_{i,k}^j$.

Proof 3 (proof by deduction for Theorem VII.1): The Russian dolls test works by determining whether the implicit deadline (i.e., latest-allowable finish time) of a subtask can be nested within the laxity of another subtask's implicit deadline or vice versa. When τ_i^j is scheduled, the implicit deadlines for active subtasks and

$$\tau_i^q | \exists D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}, \qquad j < q \leq b$$

may be tightened. If the problem is correctly structured as an augmented self-suspending task model, then the execution of τ_i^j only affects the implicit deadline of another subtask τ_x^y if there exists a subtask-to-subtask deadline

$$D^{s2s}_{\langle \tau_i^j, \tau_i^z \rangle}$$

such that $x = i$ and $j \leq y \leq z$. If it is feasible to nest an active subtask τ_x^y within the slack of every other

$$\tau_i^q | \exists D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}, \qquad j < q \leq b$$

or vice versa; then, this schedule commitment will reduce the schedule slack available for future scheduling commitments but will not invalidate previous commitments. In other words, for a new scheduling commitment to satisfy the Russian dolls test, there must be no cascading effects on implicit deadlines for scheduled subtasks. As such, a pairwise comparison between these sets of subtasks is sufficient to guarantee the production of a feasible schedule.

VIII. Results and Discussion

In this section, we empirically evaluate the tightness and computational complexity of our schedulability test and scheduling algorithm. We perform our empirical analysis using randomly generated task sets. The number of subtasks m_i of a task τ_i is drawn from a uniform distribution in the range $[1, 2n]$, $m_i \sim U(1, 2n)$, with n being the number of tasks. If $m_i = 1$, then that task does not have a self-suspension. Subtask costs, self-suspension durations, and phase offsets are drawn from uniform distributions $C_i^j \sim U(1, 10)$, $E_i^j \sim U(1, 10)$, and $\phi \sim U(1, 10)$, respectively. Task periods are drawn from a uniform distribution such that

$$T_i \sim U\left(\sum_{i,j} C_i^j, 2\sum_{i,j} C_i^j\right)$$

Task deadlines are drawn from a uniform distribution such that

$$D_i \sim U\left(\sum_{i,j} C_i^j, T_i\right)$$

The durations of subtask-to-subtask deadlines

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle}$$

are drawn from a uniform distribution:

$$\sim U\left(\left(C_i^b + \sum_{j=a}^{b-1} C_a^b + E_a^b\right), 2\left(C_i^b + \sum_{j=a}^{b-1} C_a^b + E_a^b\right)\right)$$

The number of subtasks m_i within each task τ_i is drawn from a uniform distribution such that $m_i \sim U(1, 2n)$, where n is the number of tasks. For example, a task set with 10 tasks may have anywhere from 1 to 20 subtasks per task. All parameters are integers.

We evaluate the performance of our methods as a function of problem size, and we consider task sets with 2 to 23 tasks. We note that the average number of subtasks in the task set is equal to the square of the number of tasks; for 23 tasks, there are, on average, 529 subtasks in the task set. Each data point and associated error bar represent, respectively, the median and quartiles for 50 randomly generated task sets.

We benchmark our method against the naive approach that treats all self-suspensions as task costs. To our knowledge, our method is the first polynomial-time test for hard, periodic, nonpreemptive, self-suspending task systems with any number of self-suspensions per task.

A. Metrics for Tightness of the Schedulability Test and Scheduling Algorithm

We use three metrics to evaluate the tightness of our schedulability test. First, we consider the percentage of self-suspension time our method treats as a task cost, as calculated in Eq. (27):

$$\hat{E} = \frac{W^{\boldsymbol{\tau}}_{\text{free}} + W^{\boldsymbol{\tau}}_{\text{embedded}}}{\sum_{i,j} E_i^j} * 100\% \tag{27}$$

This metric provides a comparison between our method and the naive worst-case analysis that treats all self-suspensions as idle time. We similarly evaluate the tightness of our scheduling algorithm using the percentage of self-suspension time that the processor is idle. Second, we evaluate the proportion of randomly generated problems for which the schedulability test does guarantee feasibility when the scheduling algorithm also produces a feasible schedule. Third, we report the deadline miss ratio for problems where the test does not guarantee schedulability. This is the proportion of the deadlines that the schedulability test cannot guarantee will be satisfied. This ratio is a commonly used metric to evaluate the tightness of schedulability tests and scheduling algorithms [39,40].

B. Evaluation of Test and Scheduling Algorithm

First, we evaluate tightness of the JSF schedulability test and scheduling algorithm for the traditional self-suspending task model and the model augmented with subtask-to-subtask deadline constraints. We use a metric $\hat{D}$ to classify the degree to which subtask-to-subtask deadlines constrain the task set. The quantity $\hat{D}$ is computed as the number of subtasks constrained by subtask-to-subtask deadlines, normalized by the total number of subtasks released during the hyperperiod. We show the empirical tightness of our schedulability test and scheduling algorithm for task sets where zero (Fig. 14), one-quarter (Fig. 15), and one-half (Fig. 16) of the subtasks released during the hyperperiod are constrained by subtask-to-subtask deadlines. The case where no subtasks are constrained by subtask-to-subtask deadlines corresponds to the traditional self-suspending model presented in Eq. (1).

For small problem sizes, the schedulability test largely overestimates the amount of time the processor will idle due to self-suspensions. However, the schedulability test and scheduling algorithm quickly converge for the traditional self-suspending task model as the task size increases. The amount of idle time the processor experiences due to self-suspensions approaches approximately 10% for both the schedulability

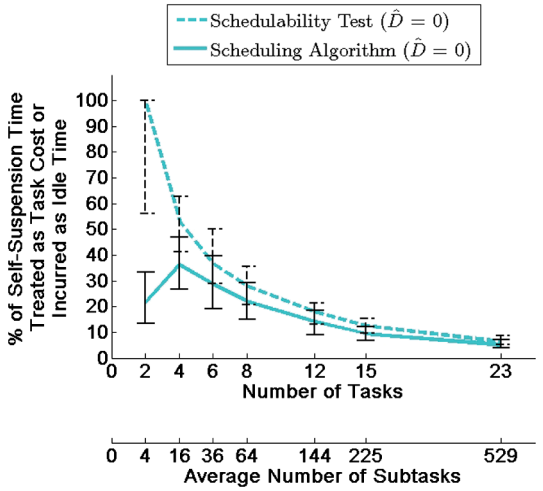


Fig. 14 This plot shows the amount of self-suspension time treated as task cost.

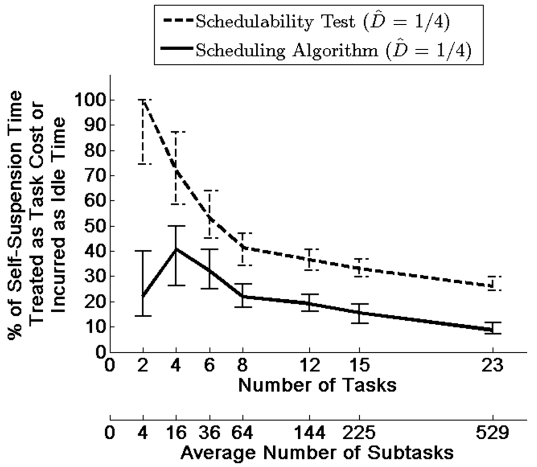


Fig. 15 This plot shows the amount of self-suspension time treated as task cost.

test and scheduling algorithm as task size grows (Fig. 14). We do not see that same behavior for task sets with subtask-to-subtask deadlines (Figs. 15 and 16). Recall that our schedulability test treats all self-suspensions constrained by subtask-to-subtask deadlines (embedded self-suspensions) as a task cost (or processor idle time). Online, our scheduling algorithm uses the Russian dolls test to correctly interleave subtasks during these embedded self-suspensions to reduce the processor idle time.

Figure 17 shows the proportion of problems for which the schedulability test does guarantee feasibility when the scheduling algorithm also produces a feasible schedule. The test success on schedulable problems remains high, between 70 and 100% as a function of problem size, for the traditional model without subtask-to-subtask deadlines. Test success is approximately 50% for $\hat{D} = \frac{1}{4}$ and drops off to 10% for $\hat{D} = \frac{1}{2}$ as problem size grows. Nonetheless, our methods are tight for task sets that have a relatively low number of subtasks constrained by subtask-to-subtask deadlines. Lastly, the deadline miss ratio remains low, under 10% for all large problems tested, as shown in Fig. 18. To our knowledge, this is the first polynomial-time schedulability test and scheduling algorithm that handles self-suspending task models with subtask-to-subtask deadlines.

Figure 19 shows the proportion of problems for which our JSF schedulability test guarantees feasibility when the JSF scheduling algorithm also produces a feasible schedule, versus the proportion of problems for which the naive approach, which treats self-suspension durations as a task cost, guarantees feasibility. We show this comparison for the traditional task model where there are no subtask-to-subtask deadlines. The naive approach is unable to guarantee the feasibility of many task sets, whereas our technique provides guarantees for nearly all randomly generated problems.

C. Computational Complexity

1. *JSF Schedulability Test*

The JSF schedulability test is computed in polynomial time. We bound the time complexity as follows, noting that $m_{\max}$ is the largest number of subtasks in any task in $\boldsymbol{\tau}$ and $T_{\min}$ is the shortest period of any task in $\boldsymbol{\tau}$.

The complexity of evaluating Eq. (18) for $\boldsymbol{\tau}^*$ is upper bounded by

$$O\left(n^2 m_{\max} \frac{H}{T_{\min}}\right)$$

where

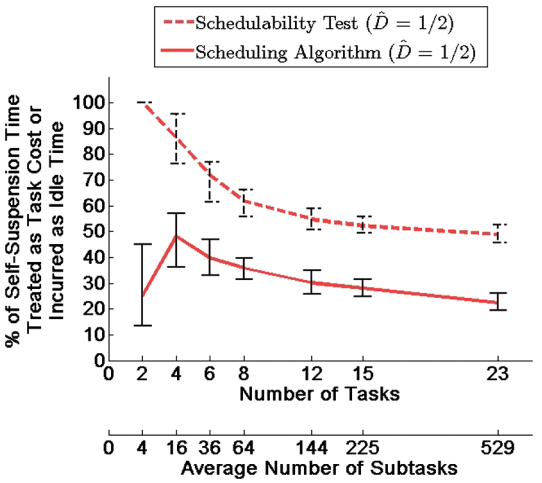


Fig. 16 This plot shows the amount of self-suspension time treated as task cost.

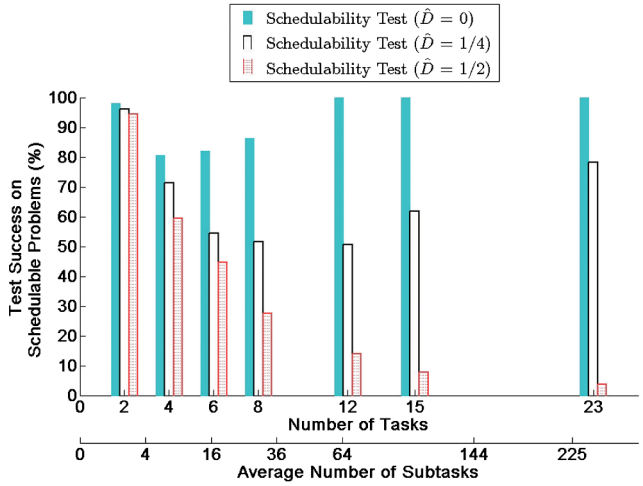


Fig. 17 Percentage of problems for which the schedulability test guarantees feasibility when the scheduling algorithm also produces a feasible schedule.

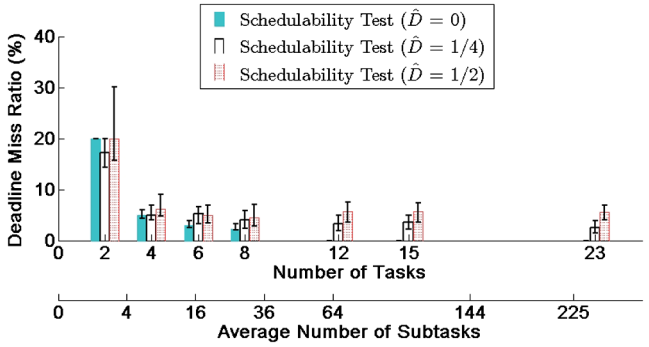


Fig. 18 Deadline miss ratio, which is the percentage of deadlines that the schedulability test could not guarantee to satisfy.

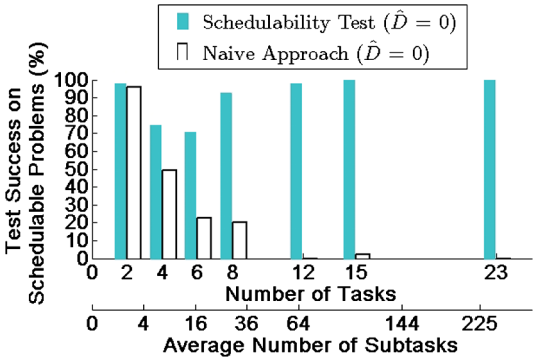


Fig. 19 Percentage of problems passing our schedulability test versus the naive approach.

$$O\left(nm_{\max}\frac{H}{T_{\min}}\right)$$

bounds the number of self-suspensions in $\boldsymbol{\tau}^*$. The complexity of testDeadline$(\boldsymbol{\tau}, d, j)$ is dominated by evaluating Eq. (18). In turn, constructTaskSuperset$(\boldsymbol{\tau})$ is dominated by

$$O\left(n\frac{H}{T_{\min}}\right)$$

calls to testDeadline$(\boldsymbol{\tau}, d, j)$. Thus, for the algorithm we have presented in Figs. 10 and 11, the computational complexity is

$$O\left(n^3 m_{\max}\left(\frac{H}{T_{\min}}\right)^2\right)$$

However, we note our implementation of the algorithm is more efficient. We reduce the complexity to

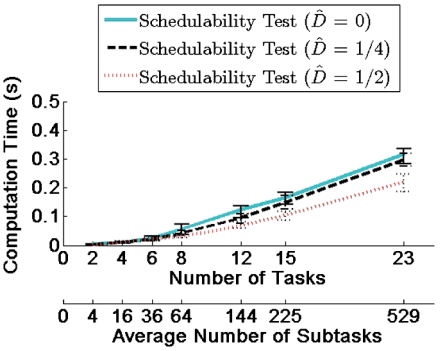


Fig. 20 Schedulability test computation time for task sets with and without subtask-to-subtask deadlines.

$$O\left(n^2 m_{\max} \frac{H}{T_{\min}}\right)$$

by caching the result of the intermediate steps in evaluating Eq. (18). In constructTaskSuperset($\boldsymbol{\tau}$), we make

$$O\left(n \frac{H}{T_{\min}}\right)$$

calls to testDeadline($\boldsymbol{\tau}, d, j$). We arrange the calls to testDeadline($\boldsymbol{\tau}, d, j$) such that the subtask index (e.g., j') is always greater than or equal to that of the previous call (e.g., j), and then we simply build on the upper bound $H_{\text{UB}}^{\tau|_j}$ through consideration of the contribution of the additional subtasks (e.g., $\tau_a^b | j < b \leq j', i \in N$). Thus, constructTaskSuperset($\boldsymbol{\tau}$) only evaluates Eq. (18) once for the entire super task set $\boldsymbol{\tau}^*$.

We provide empirical validation of the computational time of the JSF schedulability test in Fig. 20 with this more efficient implementation. This figure shows the computation time of the JSF schedulability test as a function of problem size and the proportion of subtasks constrained by subtask-to-subtask deadline constraints $\hat{D}$. These results were generated using a MATLAB implementation of the schedulability test and run on a commercial off-the-shelf laptop with an Intel Core i7-2820QM CPU with 2.30 GHz and 8 GB of RAM. With a more efficient implementation, we expect the computation time to significantly decrease.

2. *JSF Scheduling Algorithm*

Our scheduling algorithm is also computed in polynomial time. We bound the time complexity for each time step of the algorithm. The largest number of released subtasks at any point in time is n. The algorithm attempts to schedule, at worst, all n of the released subtasks. For each attempt to schedule a subtask, the algorithm calls the Russian dolls test to determine temporal feasibility. The Russian dolls test must perform a pairwise comparison of all active subtasks. In the worst case, there are

$$O\left(n \sum_i m_i\right)$$

active subtasks. Thus, the complexity of our scheduling algorithm is

$$O(n^2 m_{\max})$$

per time step.

IX. Future Work

The extension of our schedulability test to preemptive task sets is not trivial and remains an active research topic. There are cases where a nonpreemptive JSF scheduler can successfully produce a feasible schedule, whereas a preemptive scheduler cannot. Figure 21 shows one such example. The processor interleaves subtasks within self-suspension durations to reduce idle time. However, if a subtask is preempted, the ensuing

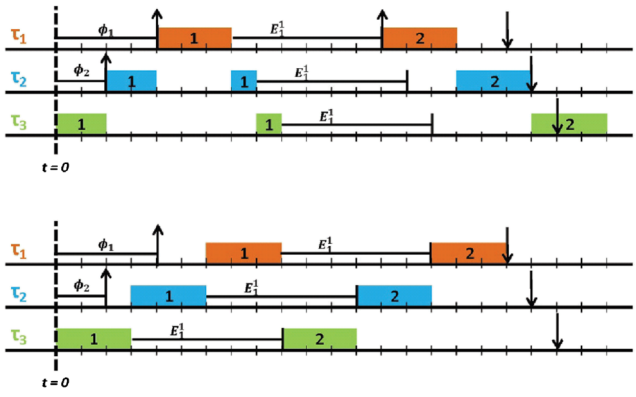


Fig. 21 Example task set that is made infeasible under EDF by allowing preemption.

self-suspension is delayed until the subtask finishes. Such a preemption can result in less efficient use of the self-suspension duration and longer completion time for the task set, thereby producing a schedule that does not meet deadlines.

Generalization of the schedulability test and algorithm to multiprocessor systems also poses challenges. One approach is to decompose task allocation and sequencing. Once tasks are allocated to processors, the schedulability test may be applied to each processor. However, environments with shared memory resources preclude this approach. Recent work proposes scheduling these task sets through simulation [13], but generalization of an analytical schedulability test for self-suspending task sets with shared resources remains an open problem.

X. Conclusions

In this paper, a polynomial-time solution to the open problem of determining the feasibility of hard, periodic, nonpreemptive, self-suspending task sets with any number of self-suspensions in each task, phase offsets, and deadlines less than or equal to periods was presented. The self-suspending task model and schedulability test to handle task sets with subtask-to-subtask deadlines was also generalized, which constrained the upper-bound temporal difference between the start and finish of two subtasks within the same task. These constraints are commonly included in AI and operations research scheduling models.

The current schedulability test worked by leveraging a novel priority scheduling policy for self-suspending task sets, called the *j*th subtask first, that restricted the behavior of a self-suspending task set so as to provide an analytical basis for an informative schedulability test. The correctness of the schedulability test was proved.

Furthermore, an online consistency test was also introduced, called the Russian dolls test, that ensured temporal feasibility during runtime when scheduling against subtask-to-subtask deadlines. The tightness and computational complexity of the current methods were empirically evaluated. For the standard self-suspending task model, the current method enabled the processor to effectively use 95% of the self-suspension time to process tasks. The current test provides a substantial improvement compared to the approach that treats all self-suspension time as a task cost, which is unable to guarantee feasibility as the size of the task set increases.

Acknowledgments

Funding for this project was provided by Boeing Research and Technology and the National Science Foundation Graduate Research Fellowship Program under grant number 2388357.

References

[1] Liu, J., *Real-Time Systems*, Prentice–Hall, Upper Saddle River, NJ, 2000, pp. 146–186.

[2] Stankovic, J., Supri, M., Natale, M. D., and Buttazzo, G., "Implications of Classical Scheduling Results for Real-Time Systems," *Computer*, Vol. 28, No. 6, 1995, pp. 16–25.

[3] Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J., "Real-Time Scheduling: The Deadline-Monotonic Approach," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, IEEE, Piscataway, NJ, 1991, pp. 133–137.

[4] Kim, I.-G., Choi, K.-H., Park, S.-K., Kim, D.-Y., and Hong, M.-P., "Real-Time Scheduling of Tasks That Contain the External Blocking Intervals," *Proceedings of the Conference on Real-Time Computing Systems and Applications*, IEEE, New York, 1995, pp. 54–59.

[5] Liu, C. L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, 1973, pp. 46–61.

[6] Lakshmanan, K., Kato, S., and Rajkumar, R. R., "Open Problems in Scheduling Self-Suspending Tasks," *Proceedings of the Real-Time Scheduling Open Problems Seminar (RTSOPS)*, IEEE, New York, July 2010.

[7] Liu, C., and Anderson, J. H., "An O(m) Analysis Technique for Supporting Real-Time Self-Suspending Task Systems," *Proceedings of the Real-Time Systems Symposium (RTSS)*, IEEE, New York, 2012, pp. 373–382.

[8] Richard, P., "On the Complexity of Scheduling Real-Time Tasks with Self-Suspensions on One Processor," *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE, New York, July 2003, pp. 187–194.

[9] Ridouard, F., Richard, P., Cottet, F., and Traoré, K., "Some Results on Scheduling Task with Self-Suspensions," *Journal of Embedded Computing*, Vol. 2, Nos. 3–4, 2006, pp. 213–301.

[10] Lakshmanan, K., and Rajkumar, R. R., "Scheduling Self-Suspending Real-Time Tasks with Rate-Monotonic Priorities," *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, New York, April 2010, pp. 3–12.

[11] Brunet, L., Choi, H.-L., and How, J. P., "Consensus-Based Auction Approaches for Decentralized Task Assignment," *Proceedings of the AIAA Guidance, Navigation, and Control Conference (GNC)*, AIAA Paper 2008-6839, 2008.

[12] Han-Lim, C., Luc, B., and Jonathan, H., "Consensus-Based Decentralized Auctions for Robust Task Allocation," *IEEE Transactions on Robotics*, Vol. 25, No. 4, Aug. 2004, pp. 912–926.

[13] Gombolay, M. C., Wilcox, R. J., and Shah, J. A., "Fast Scheduling of Multi-Robot Teams with Temporospatial Constraints," *Proceedings of the Robots: Science and Systems (RSS)*, MIT Press, Berlin, June 2013.

[14] Hooker, J. N., "Principles and Practice of Constraint Programming," *A Hybrid Method for Planning and Scheduling*, edited by Wallace, M., Lecture Notes in Computer Science, Springer, Berlin Heidelberg, Vol. 3258, 2004, pp. 305–316.

[15] Cordeau, J.-F., Stojković, G., Soumis, F., and Desrosiers, J., "Benders Decomposition for Simultaneous Aircraft Routing and Crew Scheduling," *Transportation Science*, Vol. 35, No. 4, 2001, pp. 375–388. doi:10.1287/trsc.35.4.375.10432

[16] Castro, E., and Petrovic, S., "Combined Mathematical Programming and Heuristics for a Radiotherapy Pre-Treatment Scheduling Problem," *Journal of Scheduling*, Vol. 15, No. 3, 2012, pp. 333–346. doi:10.1007/s10951-011-0239-8

[17] Chen, J., and Askin, R. G., "Project Selection, Scheduling and Resource Allocation with Time Dependent Returns," *European Journal of Operational Research*, Vol. 193, No. 1, 2009, pp. 23–34. doi:10.1016/j.ejor.2007.10.040

[18] Ridouard, F., and Richard, P., "Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions," *Proceedings of the Real-Time and Network Systems (RTNS)*, IEEE, New York, May 2004, pp. 47–56.

[19] Bertsimas, D., and Weismantel, R., *Optimization Over Integers*, Dynamic Ideas, Belmont, MA, 2005, pp. 1–33.

[20] Dechter, R., Meiri, I., and Pearl, J., "Temporal Constraint Networks," *Artificial Intelligence*, Vol. 49, No. 1, 1991, pp. 61–95.

[21] Muscettola, N., Morris, P., and Tsamardinos, I., "Reformulating Temporal Plans for Efficient Execution," *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR&R)*, Morgan Kaufmann, San Francisco, June 1998, pp. 444–452.

[22] Harbour, M. G., and Palencia, J. C., "Response Time Analysis for Tasks Scheduled Under EDF Within Fixed Priorities," *Proceedings of the Real-Time Systems Symposium (RTSS)*, IEEE, New York, 2003, pp. 200–209.

[23] Rajkumar, R. R., "Dealing with Self-Suspending Period Tasks," IBM, Thomas J. Watson Research Center TR, Armonk, NY, 1991.

[24] Abdeddaïm, Y., and Masson, D., "Scheduling Self-Suspending Periodic Real-Time Tasks Using Model Checking," *Proceedings of the Real-Time Systems Symposium (RTSS)*, ACM, New York, 2011, pp. 211–220.
[25] Armando, A., Castellini, C., Giunchiglia, E., Idini, M., and Maratea, M., "TSAT++: An Open Platform for Satisfiability Modulo Theories," *Electronic Notes in Theoretical Computer Science*, Elsevier, Amsterdam, Netherlands, Vol. 125, No. 3, 2005, pp. 25–36.
[26] Nelson, B., and Kumar, T. K., "CircuitTSAT: A Solver for Large Instances of the Disjunctive Temporal Problem," *Proceedings of the ICAPS*, edited by Rintanen, J., Nebel, B., Beck, J. C., and Hansen, E. A., AAAI, Palo Alto, CA, 2008, pp. 232–239.
[27] Audsley, N. C., Burns, A., Richardson, M. F., Tindell, K., and Wellings, A. J., "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling," *Software Engineering Journal*, Vol. 8, No. 5, Sept. 1993, pp. 284–292.
[28] Ming, L., "Scheduling of the Inter-Dependent Messages in Real-Time Communication," *Proceedings of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.
[29] Devi, U. C., "An Improved Schedulability Test for Uniprocessor Periodic Task Systems," *Proceedings of the 16th Euromicro Technical Committee on Real-Time Systems*, IEEE, New York, 2003, pp. 23–30.
[30] Liu, C., and Anderson, J. H., "Suspension-Aware Analysis for Hard Real-Time Multiprocessor Scheduling," *25th Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE, New York, July 2013, pp. 271–281.
[31] Liu, C., and Anderson, J. H., "Task Scheduling with Self-Suspensions in Soft Real-Time Multiprocessor Systems," *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, IEEE, New York, Dec. 2009, pp. 425–436.
[32] Liu, C., and Anderson, J. H., "Improving the Schedulability of Sporadic Self-Suspending Soft Real-Time Multiprocessor Task Systems," *Proceedings of the 16th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, IEEE, New York, Aug. 2010, pp. 13–22.
[33] Liu, C., and Anderson, J. H., "Scheduling Suspendable, Pipelined Tasks with Non-Preemptive Sections in Soft Real-Time Multiprocessor Systems," *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, Piscataway, NJ, April 2010, pp. 23–32.
[34] Gombolay, M. C., and Shah, J. A., "Multiprocessor Scheduler for Task Sets with Well-Formed Precedence Relations, Temporal Deadlines, and Wait Constraints," *AIAA Infotech@Aerospace*, 2012.
[35] Laborie, P., "Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results," *Artificial Intelligence*, Vol. 143, No. 2, 2003, pp. 151–188.
doi:10.1016/S0004-3702(02)00362-4
[36] Vilm, P., Barták, R., and Čepek, O., "Extension of o(nlogn) Filtering Algorithms for the Unary Resource Constraint to Optional Activities," *Constraints*, Vol. 10, No. 4, 2005, pp. 403–425.
doi:10.1007/s10601-005-2814-0
[37] Baptiste, P., and Pape, C. L., "Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling," *U.K. Planning and Scheduling Special Interest Group*, Liverpool, England, U.K., Nov. 1996.
[38] Dertouzos, M. L., and Mok, A. K.-L., "Multiprocessor Online Scheduling of Hard-Real-Time Tasks," *IEEE Transactions on Software Engineering*, Vol. 15, Dec. 1989, pp. 1497–1506.
[39] Lu, C., Stankovic, J. A., Abdelzaher, T. F., Tao, G., Son, S. H., and Marley, M., "Performance Specifications and Metrics for Adaptive Real-Time Systems," *Proceedings of the 21st IEEE Real-Time Systems Symposium*, IEEE, Piscataway, NJ, 2000, pp. 13–23.
[40] Manolache, S., Eles, P., and Pen, Z., "Optimization of Soft Real-Time Systems with Deadline Miss Ratio Constraints," *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, Piscataway, NJ, May 2004, pp. 562–570.

S. Ferrari
Associate Editor