Schedulability Analysis of Task Sets with Upper and Lowerbound Temporal Constraints

Matthew C. Gombolay* and Julie A. Shah

Massachusetts Institute of Technology

Cambridge, MA 02139, USA

gombolay@csail.mit.edu, julie_a_shah@csail.mit.edu

Increasingly real-time systems must handle the self-suspension of tasks, i.e. lowerbound 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, we present a polynomial-time test that, to our knowledge, is the first to handle non-preemptive, self-suspending tasks sets with hard deadlines, where each task has any number of self-suspensions. To construct our test, we leverage a novel priority scheduling policy, j^{th} Subtask First (JSF), 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 upperbound and lowerbound temporal constraints requires an idling scheduling policy and is known to be NP-Hard. However we empirically validate the tightness of our schedulability test and scheduling algorithm, and show that the processor is able to effectively utilize 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 whose timing properties can be predicted and mathematically proven correct.[1,2] 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.[1,3–6]

*Corresponding Author

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.[7–10] In processor scheduling, self-suspensions (i.e. lowerbound "wait times" between subtasks), can result both due to hardware and software architecture. At the hardware level, the addition of multi-core processors, dedicated cards (e.g., GPUs, PPUs, etc.), and various I/O devices such as external memory drives, can necessitate task self-suspensions. Furthermore, the software that utilizes these hardware systems can employ synchronization points and other algorithmic techniques that also result in self-suspensions.[11] 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 upperbound and lowerbound (self-suspension) temporal constraints is a challenging problem with important applications outside of processor scheduling, as well. Other examples include autonomous tasking of unmanned aerial and under-water vehicles,[12,13] scheduling of factory operations,[14,15] and scheduling of aircraft and flight crews.[16] 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.[14] Solutions to these applications typically draw from methods in AI and OR,[15–18] which provide complete search algorithms that require exponential time to compute a solution in the worst case. These methods cannot provide fast re-computation 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 underutilized in these applications.

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

Our schedulability test and scheduling algorithm utilize a novel scheduling policy to create problem structure in self-suspending task networks. Restricting the behavior of the scheduler sacrifices completeness for this NP-Hard problem, 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 Section 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 Section 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 Section 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 Section 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 derive the computational complexity of our test and scheduling algorithm.

II. Self-Suspending Task Model

The basic model for the self-suspending task set[7] is shown in Equation 1.

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

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. C_i^j is the worst-case duration of the j^{th} subtask of τ_i, and E_i^j is the worst-case duration of the j^{th} self-suspension interval of τ_i.

Subtasks within a task are dependent, meaning that a subtask τ_i^{j+1} must start after the finish times of the subtask τ_i^j and the self-suspension E_i^j. T_i and D_i are the period and deadline of τ_i, respectively, where $D_i \leq T_i$. Lastly, a phase offset delays the release of a task, τ_i, by the duration, ϕ_i, after the start of a new period.

The self-suspending task model shown in Equation 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 upperbound 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 Equation 2.

$$D^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle} : \left(f_i^b - s_i^a \leq d^{s2s}_{\langle \tau_i^a, \tau_i^b \rangle} \right) \tag{2}$$

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 $d_{\langle \tau_i^a, \tau_i^b \rangle}$ is the upperbound 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. lowerbound wait times between subtasks) may arise due to, for example, "cure times" involved in the assembly process. Upperbound 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. Upperbound subtask-to-subtask deadlines encode the amount of time an aircraft can remain in a holding pattern based on fuel considerations. While each domain has its own nuances in problem formulation, there is sufficient underlying commonality in problem structure to investigate the application of real-time scheduling techniques to these problems.

In the remainder of this paper, we present a schedulability test and complementary scheduling algorithm that handles periodic, self-suspending task sets. We develop the test for non-preemptable subtasks, meaning the interruption of a subtask significantly degrades its quality. However, we note that a schedulability test for non-preemptive subtasks conservatively bounds the temporal resources necessary to execute a preemptable system. 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.

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.

III.A. Challenge Posed by Task Self-Suspension

The problem of scheduling, or testing the schedulability of a self-suspending task set, is NP-Hard as can be shown through an analysis of the interaction of self-suspensions and task deadlines.[9,21,22] Many uniprocessor, priority-based scheduling algorithms, such as Earliest Deadline First (EDF) or Rate-Monotonic (RM) introduce scheduling anomalies since they do not account for this interaction.[7,23]

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 non-preemptive task sets. Lakshmanan *et al.*[7] report that finding an anomaly-free scheduling priority for self-suspending task sets remains an open problem.

We provide illustrations to exemplify different types of scheduling anomalies in Figures 1-2. Each figure depicts a feasible schedule (top) and an infeasible schedule resulting from a scheduling anomaly (bottom). Upward arrows indicate the release of a task, and downward arrows indicate a task's deadline. Self-suspensions are represented by a horizontal bar with a corresponding label. Blocks correspond to the execution cost of each subtask and are numbered according to the subtask index. For example, a block labeled "2" on a row labeled "τ_3" corresponds to τ_3^2.

III.A.1. Scheduling Anomalies Produced by Reducing Task Cost

The first type of scheduling anomaly occurs when a reduction in the computation time of a subtask causes the processor to violate a deadline constraint. This type of scheduling anomaly was first described by Ridouard *et al.*[23] Figure 1 shows a scenario where execution of three tasks under the Earliest Deadline First (EDF) 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.

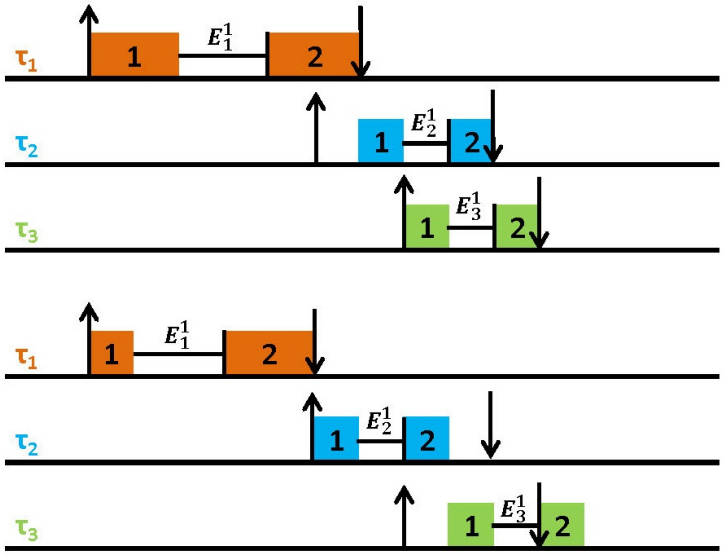

Figure 1: A scheduling anomaly occurs when the reduction in the computation time of a subtask causes the processor to violate a deadline constraint. The top graph shows a feasible schedule and the bottom graph shows how the same task set is rendered infeasible due to a reduction in the cost of one subtask C_1^1.

III.A.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 2 shows a scenario where the execution of two tasks under the Earliest Deadline First (EDF) 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.

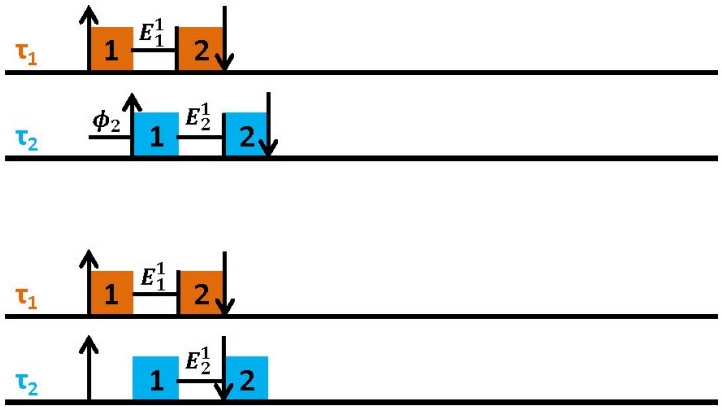

Figure 2: Another scheduling anomaly occurs when the reduction of a phase offset duration allows a task to release earlier, and thus prevents the processor from satisfying all deadline constraints. The top graph shows a feasible schedule and the bottom graph shows how the same task set is rendered infeasible due to a decreased phase offset ϕ_2.

III.B. Schedulability Testing

Given sufficient computational resources, 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, as is necessary when the task set changes. To gain computational speed, many real-time systems utilize 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. The naive method for testing the schedulability of these task sets is to treat self-suspensions as task costs; however, this can result in significant under-utilization of the processor if the duration of self-suspensions is large relative to task cost.[8]

Fast polynomial times schedulability tests have been studied for restrictions of the self-suspending task model. Kim *et al.*[4] presents two methods for testing task sets where each task has exactly one self-suspension.

Their first method builds on work by Wellings *et al.*[3] 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[3] 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[6] to more tightly bound the amount of self-suspension time that must be considered as 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[27] develop analyses for another restricted form of the 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 analyzes the schedulability of the task set when it is executed under the fixed-priority RM scheduling policy, and treats delays of 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[27] developed a similar method for testing the schedulability of self-suspending task sets operating under the EDF dynamic-priority scheduling algorithm.

Recently, Abdeddaïm and Masson introduced an approach for testing self-suspending task sets using model checking with Computational Tree Logic (CTL).[24] While 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 *et al.*[11] also increase generality by developing a pseudo-polynomial-time test to determine the worst-case interference imposed on a lower priority self-suspending tasks by higher priority *non-suspending* tasks. However, Lakshmanan *et al.* report that an exact-case test for multiple self-suspensions per task remains an open problem.

Finally, recent works by C. Liu and Anderson[8,28] analyze preemptive task sets with multiple self-suspensions per task for soft real-time requirements. We have not yet seen a schedulability test for hard, non-preemptive task sets with multiple self-suspensions per task. Our approach seeks to fill this gap by providing the first such analytical schedulability test.

III.C. Scheduling Algorithms

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

Rajkumar[29] presents an algorithm called *Period Enforcer* for preemptive, self-suspending task sets scheduled with the RM scheduling algorithm. Period Enforcer works by adding pre-conditions 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 $\tau_i', \tau_i'', \tau_i'''$, each with the same deadline as τ_i. However, their approach does not handle non-preemptive task sets, nor is there a complementary, analytical schedulability test.

Lakshmanan *et al.*[11] build on previous approaches to develop a *static slack enforcement algorithm* that delays the release times of subtasks to improve the schedulability of 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.

While there exist scheduling algorithms that can handle non-preemptive, self-suspending tasks sets with multiple suspensions per task, we have not yet seen a such an algorithm that is accompanied by an 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 free subtask, $\tau_i^j \in \boldsymbol{\tau_{free}}$, is a subtask that does not share a deadline constraint with τ_i^{j-1}. In other words, a subtask τ_i^j is free* iff *for any deadline $D_{\langle \tau_i^a, \tau_i^b \rangle}$ associated with that task, $(j \leq a) \vee (b < j)$. We define τ_i^1 as free since there does not exist a preceding subtask.*

Definition 2 *An embedded subtask, $\tau_i^{j+1} \in \boldsymbol{\tau_{embedded}}$, is a subtask shares a deadline constraint with τ_i^j (i.e., $\tau_i^{j+1} \notin \boldsymbol{\tau_{free}}$). $\boldsymbol{\tau_{free}} \cap \boldsymbol{\tau_{embedded}} = \emptyset$.*

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 free self-suspension, $E_i^j \in \boldsymbol{E_{free}}$, is a self-suspension that suspends two subtasks, τ_i^j and τ_i^{j+1}, where $\tau_i^{j+1} \in \boldsymbol{\tau_{free}}$.*

Definition 4 *An embedded self-suspension, $E_i^j \in \boldsymbol{E_{embedded}}$, is a self-suspension that suspends the execution of two subtasks, τ_i^j and τ_i^{j+1}, where $\tau_i^{j+1} \in \boldsymbol{\tau_{embedded}}$. $\boldsymbol{E_{free}} \cap \boldsymbol{E_{embedded}} = \emptyset$.*

In Section VI, we describe how we can use $\boldsymbol{\tau_{free}}$ to reduce processor idle time due to $\boldsymbol{E_{free}}$, and, in turn, analytically upperbound the duration of the self-suspensions that needs to be treated as task cost. We will also derive an upperbound on processor idle time due to $\boldsymbol{E_{embedded}}$.

V. Motivating our j^{th} Subtask First (JSF) 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 processor idle time, without unnecessarily sacrificing processor efficiency.

We restrict behavior using a novel scheduling priority, which we call j^{th} Subtask First (JSF). We formally define the j^{th} Subtask First priority scheduling policy in Definition 5.

Definition 5 *j^{th} Subtask First (JSF). We use j to correspond to the subtask index in τ_i^j. A processor executing a set of self-suspending tasks under JSF must execute the j^{th} subtask (free or embedded) of every task before any $(j+1)^{th}$* free *subtask. 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. The JSF priority scheduling policy offers 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. $\tau_i^j \in \boldsymbol{\tau_{free}}, \tau_i^{j+1} \in \boldsymbol{\tau_{embedded}}$), 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 utilize processor time more efficiently and operate on this structured task model exist.[30–32]

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.A 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., $T_i = D_i = T_j = D_j, \forall i, j \in \{1, 2, \ldots, n\}$). Thus, the hyperperiod of the task set is equal to the period of each task. Here we will introduce our formula for upperbounding the amount of self-suspension time that we treat as task cost, $W^{\boldsymbol{\tau}}_{free}$.

2. Next, we allow for general task release times (i.e., $\phi_i \geq 0, \forall i$). In this step, we upperbound processor idle time due to phase offsets, $W^{\boldsymbol{\tau}}_{\phi}$.

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 upperbound on processor idle time due to embedded self-suspensions $W^{\boldsymbol{\tau}}_{embedded}$.

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 $T_i \neq T_j, \forall i, j \in \{1, 2, \ldots, n\}$.

Step 1) Two Subtasks Per Task, No Deadlines, and Zero Phase Offsets

In step one, 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, we say that task deadlines are equal to task periods, and that all tasks have equal periods (i.e., $T_i = D_i = T_j = D_j, \forall i, j \in \{1, 2, \ldots, n\}$). We assert that one can upperbound the idle time due to the set of all of the E^1_i self-suspensions by analyzing the difference between the duration of the self-suspensions and the duration of the subtasks costs that will be interleaved during the self-suspensions.

We say that the set of the cost of all subtasks that *might* be interleaved during a self-suspension, E^1_i, is B^1_i. As described by Equation 3, B^j_i is the set of all of the j^{th} and $(j+1)^{th}$ subtask costs less the subtasks costs for τ^j_i and τ^{j+1}_i. Note, by definition, τ^j_i and τ^{j+1}_i cannot execute during E^j_i. We further define an operator $B^j_i(k)$ that provides the k^{th} smallest subtask cost from B^j_i. We also restrict B^j_i such that the j^{th} and $(j+1)^{th}$ subtasks must both be free subtasks if either is to be added. Because we are currently considering task sets with no deadlines, this restriction does not affect the subtasks in B^1_i during this step. In Step 4 (Section VI), we will explain why we make this restriction on the subtasks in B^j_i.

For convenience in notation, we say that N is the set of all task indices (i.e., $N = \{i | i \in \{1, 2, \ldots, n\}\}$, where n is the number of tasks in the task set, $\boldsymbol{\tau}$). Without loss of generality, we assume that the first

subtasks τ_i^1 execute in the order $i = \{1, 2, \ldots, n\}$.

$$B_i^j = \{C_x^y | x \in N \backslash i, y \in \{j, j+1\}, \tau_x^j \in \boldsymbol{\tau_{free}}, \tau_x^{j+1} \in \boldsymbol{\tau_{free}}\} \tag{3}$$

To upperbound the idle time due to the set of E_i^1 self-suspensions, we consider a worst-case interleaving of subtask costs and self-suspension durations, as shown in Equation 5 and Equation 6, where W^j is an upperbound on processor idle time due to the set of E_i^j self-suspensions, and W_i^j is an upperbound on processor idle time due to E_i^j.

To determine W^j, we first calculate the amount of processor idle time W_i^j due to each of the E_i^j self-suspensions. We calculate W_i^j based on the cost of the fewest number of subtasks (Equation 4) that will be processed during E_i^j *iff* E_i^j is the *dominant contributer* to processor idle time from the set of $E_i^j, \forall j$. We define the conditions for a self-suspension to be the dominant contributor to processor idle time in Definition 6. By then taking the maximum over all i of W_i^j, we arrive at our upperbound on processor idle time W^j due to set of j^{th} self-suspensions $\{E_i^j | i \in N\}$.

Definition 6 *A self-suspension E_i^j is the dominant contributor to processor idle time from the set of j^{th} self-suspensions $\{E_i^j | i \in N\}$ if it subsumes all idle time contributed by other self-suspensions in the set. If multiple self-suspensions subsume all idle time contributed by other self-suspensions, then they are co-dominant contributors.*

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

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

$$W^j = \max_{i | E_i^j \in \boldsymbol{E_{free}}} \left(W_i^j\right) \tag{6}$$

To prove that our method is correct, we first show that Equation 4 lowerbounds the number of free subtasks that execute during a self-suspension E_i^1, if E_i^1 is the dominant contributor to processor idle time. We perform this analysis for three cases: for $i = 1$, $1 < i = x < n$, and $i = n$. Second, we will show that, if at least $\eta_i^j = \frac{|B_i^1|}{2}$ subtasks execute during E_i^1, then Equation 5 correctly upperbounds idle time due to E_i^1. Lastly, we show that if an E_i^1 is the dominant contributor to idle time then Equation 6 holds, meaning W^j is an upperbound on processor idle time due to the set of E_i^1 self-suspensions. (In Step 3 we will show that these three equations also hold for all E_i^j.)

Proof of Correctness for *Equation* 4, where $j = 1$.

Proof 1 (Proof by Deduction for $i = 1$) *We currently assume that all subtasks are free (i.e., there are no subtask-to-subtask deadline constraints), thus* $\eta_i^j = \frac{|B_i^1|}{2} = n - 1$. *We recall that a processor executing under JSF will execute all* j^{th} *subtasks before any free* $(j+1)^{th}$ *subtask. Thus, after executing the first subtask,* τ_1^1, *there are* $n - 1$ *other subtasks that must execute before the processor can execute* τ_1^2. *Thus, Equation 4 holds for* E_1^1 *irrespective of whether or not* E_1^1 *results in processor idle time.*

Corollary 1 *From our proof for* $i = 1$, *any first subtask,* τ_x^1, *will have at least* $n - x$ *subtasks that execute during* E_x^1 *if* E_x^1 *causes processor idle time, (i.e., the remaining* $n - x$ *first subtasks in* $\boldsymbol{\tau}$*).*

Example for Equation 4, where $i = 1$

Figure 3 illustrates the proof for $i = 1$ with an example task set. Actual processor idle time is shown in red and projected onto the timeline below. The task set has three tasks as defined here:

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

At $t = 0$, all three tasks are released. We can see by inspection the duration of E_i^1 must exceed the processing time of subtasks τ_2^1 and τ_3^1 for E_i^1 to possibly cause processor idle time. We can calculate the lowerbound on the fewest subtasks that will execute during a dominant contributor E_1^1 as shown in Equation 7.

$$\eta_1^1 = \frac{|B_1^1|}{2} = \frac{4}{2} = 2 \tag{7}$$

Proof 2 (Proof by Contradiction for $1 < i = x < n$) *We assume for contradiction that fewer than* $n-1$ *subtasks execute during* E_x^1 *and* E_x^1 *is the dominant contributor to processor idle time from the set of first self-suspensions* E_i^1. *We apply Corollary 1 to further constrain our assumption that fewer than* $x - 1$ *second subtasks execute during* E_x^1. *We consider two cases: 1) fewer than* $x - 1$ *subtasks are released before* τ_x^2 *and 2) at least* $x - 1$ *subtasks are released before* τ_x^2.

First, if fewer than $x - 1$ *subtasks are released before* r_x^2 *(with release time of* τ_x^j *is denoted* r_x^j*), then at least one of the* $x - 1$ *second subtasks,* τ_a^2, *is released at or after* r_x^2. *We recall that there is no idle time during* $t = [0, f_n^1]$. *Thus,* E_a^1 *subsumes any and all processor idle time due to* E_x^1. *In turn,* E_x^1 *cannot be the dominant contributor to processor idle time.*

Second, we consider the case where at least $x - 1$ *second subtasks are released before* r_x^2. *If we complete* $x - 1$ *of these subtasks before* r_x^2, *then at least* $n - 1$ *subtasks execute during* E_x^1, *which is a contradiction.*

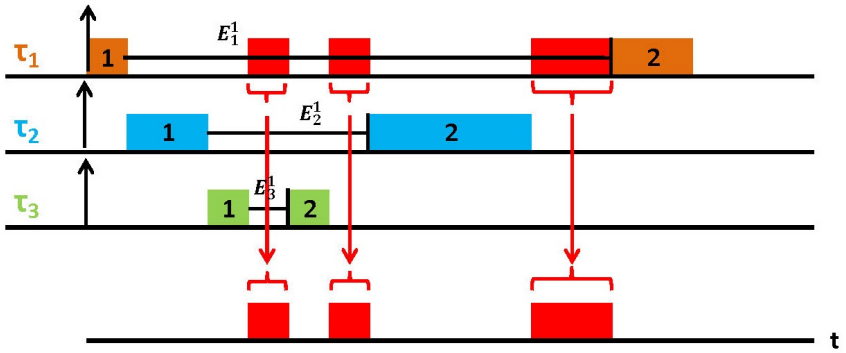

Figure 3: An example schedule is shown with three tasks where the self-suspension E_1^1 is the dominant contributor to processor idle time. Processor idle time is shown in red and projected onto the timeline below.

If fewer than $x-1$ of these subtasks execute before r_x^2, then there must exist a continuous non-idle duration between the release of one of the $x-1$ subtasks, τ_a^2 and the release of r_x^2, such that the processor does not have time to finish all of the $x-1$ released subtasks before r_x^2. Therefore, the self-suspension that defines the release of that second subtask, E_a^2, subsumes any and all idle time due to E_x^1. E_x^1 then is not the dominant contributor to processor idle time, which is a contradiction.

Example for Equation 4, where $1 < i = x < n$

Consider the example shown in Figure 4 where the dominant contributor to processor idle time is E_2^1. We calculate the lowerbound on the fewest subtasks that will execute during E_2^1 in Equation 8. The parameters of the task set for this example are:

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

$$\eta_2^1 = \frac{|B_2^1|}{2} = \frac{4}{2} = 2 \tag{8}$$

Proof 3 (Proof by Contradiction for $i = n$) *We show that if fewer than $n-1$ subtask execute during E_n^1, then E_n^1 cannot be the dominant contributor to processor idle time. As in* Case 2: i = x*, if r_n^2 is less than or equal to the release of some other task, τ_z^1, then any idle time due to E_n^1 is subsumed by E_z^1, thus E_n^1 cannot be the dominant contributor to processor idle time. If τ_n^2 is released after any other second subtask*

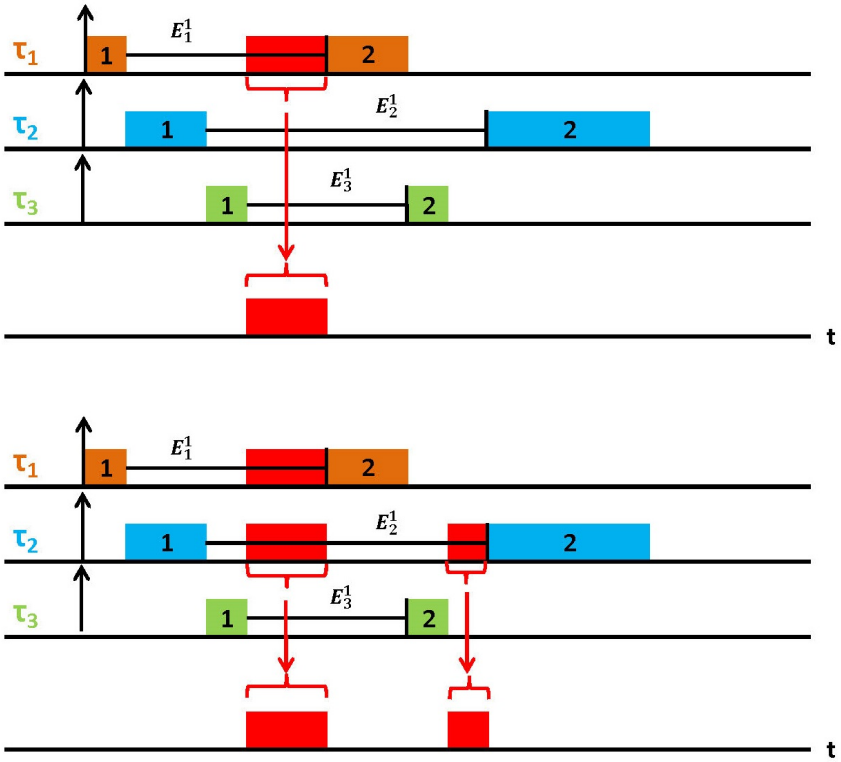

Figure 4: An example schedule is shown with three tasks where the self-suspension E_2^1 is the dominant contributor to processor idle time. Processor idle time is shown in red and projected onto the timeline below.

and fewer than $n-1$ *subtasks then at least one subtask finishes executing after* r_n^2. *Then, for the same reasoning as in* Case 2: i = x, *any idle time due to* E_n^1 *must be subsumed by another self-suspension. Thus,* E_x^1 *cannot be the dominant contributor to processor idle time if fewer than* $n-1$ *subtasks execute during* E_i^1, *where* $i = n$.

Example for Equation 4, where $i = n$

We now consider an example for the final case, where $i = n$. As shown in Figure 5, the dominant contributor to processor idle time is E_3^1. We calculate the lowerbound on the fewest subtasks that will execute during E_3^1 in Equation 8. The parameters of the task set in this example are shown here:

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

$$\eta_3^1 = \frac{|B_3^1|}{2} = \frac{4}{2} = 2 \tag{9}$$

Proof of Correctness for Equation 5, where $j = 1$.

Proof 4 (Proof by Deduction) *If* $n-1$ *subtasks execute during* E_i^j, *then the amount of idle time that results from* E_i^j *is greater than or equal to the duration of* E_i^j *less the cost of the* $n-1$ *subtasks that execute during that self-suspension. We also note that the sum of the costs of the* $n-1$ *subtasks that execute during* E_i^j *must be greater than or equal to the sum of the costs of the* $n-1$ *smallest-cost subtasks that* could possibly *execute during* E_i^j. *We can therefore upperbound the idle time due to* E_i^j *by subtracting the* $n-1$ *smallest-cost subtasks. Next we compute* W_i^1 *as the maximum of zero and* E_i^1 *less the sum of the smallest* $n-1$ *smallest-cost subtasks. If* W_i^1 *is equal to zero, then* E_i^1 *is not the dominant contributor to processor idle time, since this would mean that fewer than* $n-1$ *subtasks execute during* E_i^1 *(see proof for Equation 4). If* W_i^j *is greater than zero, then* E_i^1 *may be the dominant contributor to processor idle time, and this idle time due to* E_i^j *is upperbounded by* W_i^j.

Example for Equation 5, where $j = 1$

Returning to our example shown in Figure 3, the dominant contributor to processor idle time is E_1^1. The upperbound on processor idle time due to this self-suspension is shown in Equation 10. If either of the other self-suspensions E_2^1 or E_3^1 were the dominant contributor to processor idle time, the upperbound on processor idle time due to those self-suspensions is shown in Equations 11 and 12, respectively.

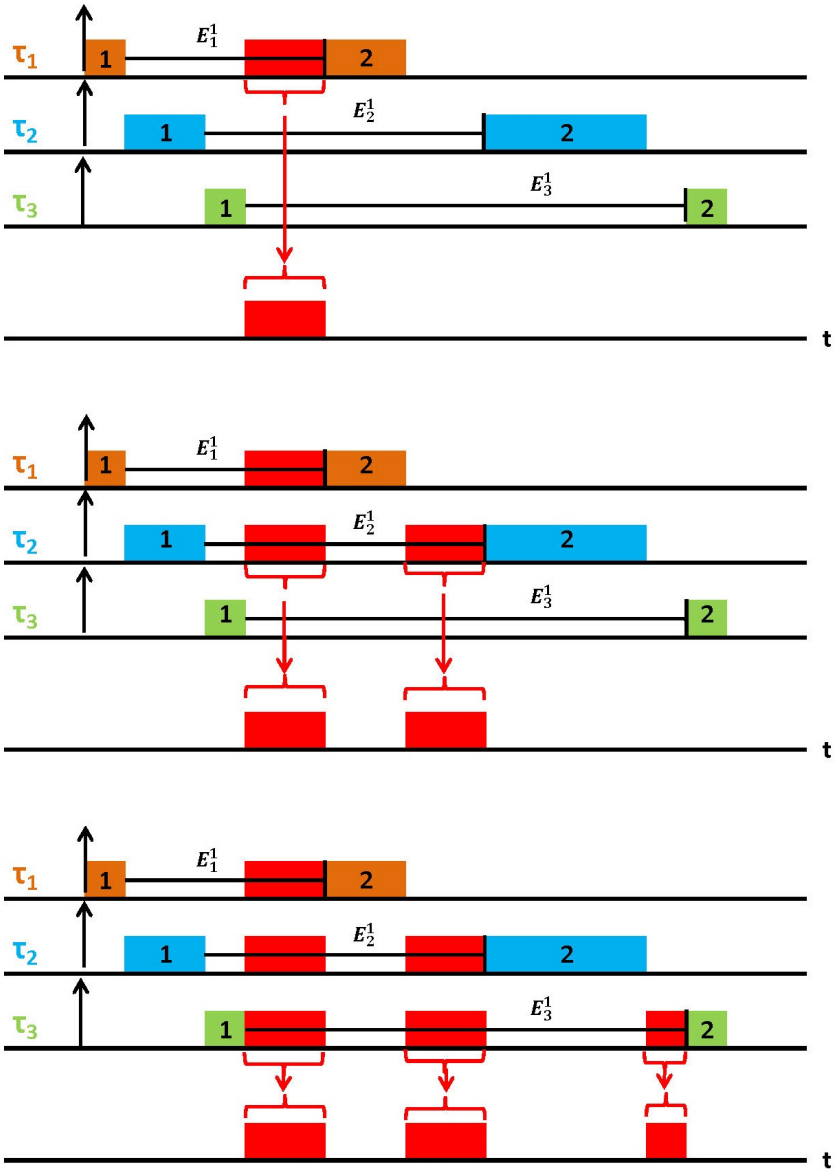

Figure 5: An example schedule is shown with three tasks where the self-suspension E_3^1 is the dominant contributor to processor idle time. Processor idle time is shown in red and projected onto the timeline below.

$$
\begin{aligned}
W_1^1 &= \max\left(\left(E_1^1 - \sum_{k=1}^{\eta_1^1} B_1^1(k)\right), 0\right) \\
&= \max\left((12 - (1+1))\right), 0) \\
&= 10
\end{aligned}
\tag{10}
$$

$$
\begin{aligned}
W_2^1 &= \max\left(\left(E_2^1 - \sum_{k=1}^{\eta_2^1} B_2^1(k)\right), 0\right) \\
&= \max\left((4 - (1+1))\right), 0) \\
&= 2
\end{aligned}
\tag{11}
$$

$$
\begin{aligned}
W_3^1 &= \max\left(\left(E_3^1 - \sum_{k=1}^{\eta_3^1} B_3^1(k)\right), 0\right) \\
&= \max\left((1 - (1+2))\right), 0) \\
&= 0
\end{aligned}
\tag{12}
$$

Figure 4 shows an example where the dominant contributor to processor idle time is E_2^1. The upperbound on processor idle time due to this self-suspension is shown in Equation 14. If either of the other self-suspensions E_1^1 or E_3^1 were the dominant contributor to processor idle time, the upperbound on processor idle time due to those self-suspensions is shown in Equations 13 and 15, respectively.

$$
\begin{aligned}
W_1^1 &= \max\left(\left(E_1^1 - \sum_{k=1}^{\eta_1^1} B_1^1(k)\right), 0\right) \\
&= \max\left((5 - (1+1))\right), 0) \\
&= 3
\end{aligned}
\tag{13}
$$

$$
\begin{aligned}
W_2^1 &= \max\left(\left(E_2^1 - \sum_{k=1}^{\eta_2^1} B_2^1(k)\right), 0\right) \\
&= \max\left((7 - (1+1))\right), 0) \\
&= 5
\end{aligned}
\tag{14}
$$

$$
\begin{aligned}
W_3^1 &= \max\left(\left(E_3^1 - \sum_{k=1}^{\eta_3^1} B_3^1(k)\right), 0\right) \\
&= \max\left((4 - (1+2))\right), 0) \\
&= 1
\end{aligned}
\tag{15}
$$

The dominant contributor to processor idle time in our third example (Figure 5) is E_3^1. The upperbound on processor idle time due to this self-suspension is shown in Equation 18. If either of the other self-suspensions E_1^1 or E_2^1 were the dominant contributor to processor idle time, the upperbound on processor idle time due to those self-suspensions is shown in Equations 16 and 17, respectively.

$$\begin{aligned} W_1^1 &= \max\left(\left(E_1^1 - \sum_{k=1}^{\eta_1^1} B_1^1(k)\right), 0\right) \\ &= \max\left((5 - (1+1))\right), 0) \\ &= 3 \end{aligned} \tag{16}$$

$$\begin{aligned} W_2^1 &= \max\left(\left(E_2^1 - \sum_{k=1}^{\eta_2^1} B_2^1(k)\right), 0\right) \\ &= \max\left((7 - (1+1))\right), 0) \\ &= 5 \end{aligned} \tag{17}$$

$$\begin{aligned} W_3^1 &= \max\left(\left(E_3^1 - \sum_{k=1}^{\eta_3^1} B_3^1(k)\right), 0\right) \\ &= \max\left((11 - (1+2))\right), 0) \\ &= 8 \end{aligned} \tag{18}$$

Proof of Correctness for Equation 6, where $j = 1$.

Proof 5 (Proof by Deduction) *Here we show that by taking the maximum over all i of W_i^1, we upperbound the idle time due to the set of E_i^1 self-suspensions. We know from the proof of correctness for Equation 4 that if fewer than $n-1$ subtasks execute during a self-suspension, E_i^1, then that self-suspension cannot be the dominant contributor to idle time. Furthermore, the dominant self-suspension subsumes the idle time due to any other self-suspension. We recall that Equation 5 bounds processor idle time caused by the dominant self-suspension, say E_q^j. Thus, we note in Equation 6 that the maximum of the upperbound processor idle time due to any other self-suspension and the upperbound for E_q^j is still an upperbound on processor idle time due to the dominant self-suspension.*

Example for Equation 6

For the example schedules shown in Figures 3, 4, and 5, the actual processor idle times are 4, 3, and 5, respectively. We upperbound the processor idle time for our three examples in Equations 19, 20, and 21.

Example 1 in Figure 3:

$$\begin{aligned} W^1 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\ &= \max(W_1^1, W_2^1, W_3^1) \\ &= \max(10, 2, 0) \\ &= 10 \end{aligned} \tag{19}$$

Example 2 in Figure 4:

$$\begin{aligned} W^1 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\ &= \max(W_1^1, W_2^1, W_3^1) \\ &= \max(3, 5, 1) \\ &= 5 \end{aligned} \tag{20}$$

Example 3 in Figure 5:

$$\begin{aligned} W^1 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\ &= \max(W_1^1, W_2^1, W_3^1) \\ &= \max(3, 5, 8) \\ &= 8 \end{aligned} \tag{21}$$

In all three examples, we can see that Equation 6 correctly upperbounds the processor idle time due to the set of first self-suspensions $\{E_i^1 | 1 \leq i \leq n\}$. Specifically, $4 \leq W^1 = 10$ (Figure 3), $3 \leq W^1 = 5$ (Figure 4), and $5 \leq W^1 = 8$ (Figure 5).

Step 2) General Phase Offsets

Next we allow for general task release times (i.e., $\phi_i \geq 0, \forall i$). 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., $s_i^1 \leq s_{i+1}^1$). We can therefore conservatively upperbound 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 Equation 22.

The quantity $W_\phi^{\boldsymbol{\tau}}$ computed in Step 2 is summed with W^1 (e.g., Equation 20) computed in Step 1 to conservatively bound the contributions of first self-suspensions and first phase offsets to processor idle time. This summation allows us to relax the assumption in Step 1 that there is no processor idle time during the interval $t = [0, f_n^1]$.

$$W_{\phi}^{\tau} = \max_{i} \phi_i \tag{22}$$

Example for Equation 22

We extend Example 2 from Figure 4 to consider non-zero phase offsets. The new task set parameters are shown in the table below:

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

The upperbound on processor idle time due to phase offsets is $W_{\phi} = 3$, as shown in Equation 23.

$$W_{\phi}^{\tau} = \max_{i} \phi_i = \max\{0, 2, 3\} = 3 \tag{23}$$

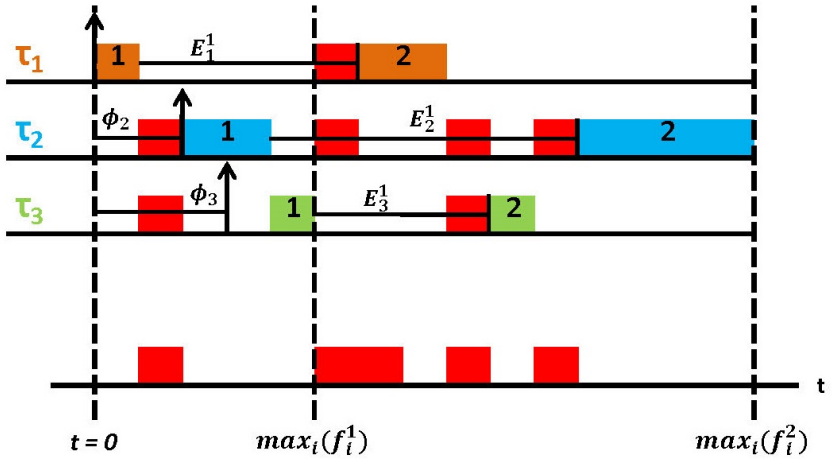

Figure 6: An example schedule is shown for three tasks with phase offsets. Processor idle time is shown in red and projected onto the timeline below. This plot includes dashed, vertical lines separate the timeline. W_{phi}^{τ} upperbounds idle time between $t = [0, max_i(f_i^1)]$, and W^1 upperbounds processor idle time during the domain of the first self-suspension $t = [max_i(f_i^1), max_i(f_i^2)]$.

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 upperbound on processor idle time that results from the worst-case interleaving of the j^{th} and $(j+1)^{th}$ subtask costs during the j^{th} self-suspensions. Again, we

recall that our formulation for upperbounding idle time due to the 1st self-suspensions in actuality was an upperbound for idle time during the interval $t = [f_n^1, max_i(f_i^2)]$.

In Step 2, we upperbounded idle time resulting from phase offsets. To do this we determined an upperbound 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 j^{th} self-suspensions, we can apply Equation 6 to determine an upperbound on processor idle time during the interval $t = \left[\max_i\left(f_i^j\right), \max_i\left(f_i^{j+1}\right)\right]$. The upperbound on total processor idle time for all free self-suspensions in the task set is computed by summing over the contribution of each of the j^{th} self-suspensions as shown in Equation 24.

$$\begin{aligned} W_{free}^{\boldsymbol{\tau}} &= \sum_j W^j \\ &= \sum_j \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^j\right) \\ &= \sum_j \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(\max\left(\left(E_i^j - \sum_{k=1}^{\eta_i^j} B_i^j(k)\right), 0\right)\right) \end{aligned} \tag{24}$$

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 were one task, τ_i, has m_i subtasks, and τ_x has only $m_x = m_i - 1$ subtasks. When we upperbound 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 Equation 3. We note that, by inspection, if τ_x^j were to execute during E_i^j, it would only reduce processor idle time.

Example for Equation 6

We extend our example from Figure 6 to include multiple self-suspensions in each task. The new task set is shown here:

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

To upperbound the processor idle time due to all self-suspensions, we first upperbound processor idle time W^j for each of the j^{th} self-suspensions $\{E_i^j | 1 \leq i \leq n\}$ using Equation 6, as shown in Equations 25 and 26. Second, we apply Equation 24 to the set of W^j terms to compute the total upperbound W_{free}^{τ}. For this example, $W_{free}^{\tau} = 7$ (Equation 27).

$$\begin{aligned} W^1 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\ &= \max\left(W_1^1, W_2^1, W_3^1\right) \\ &= \max(3, 5, 1) \\ &= 5 \end{aligned} \tag{25}$$

$$\begin{aligned} W^2 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\ &= \max(W_1^2, W_2^2, W_3^2) \\ &= \max(2, 2, 0) \\ &= 2 \end{aligned} \tag{26}$$

$$W_{free}^{\tau} = \sum_j W^j = W^1 + W^2 = 5 + 2 = 7 \tag{27}$$

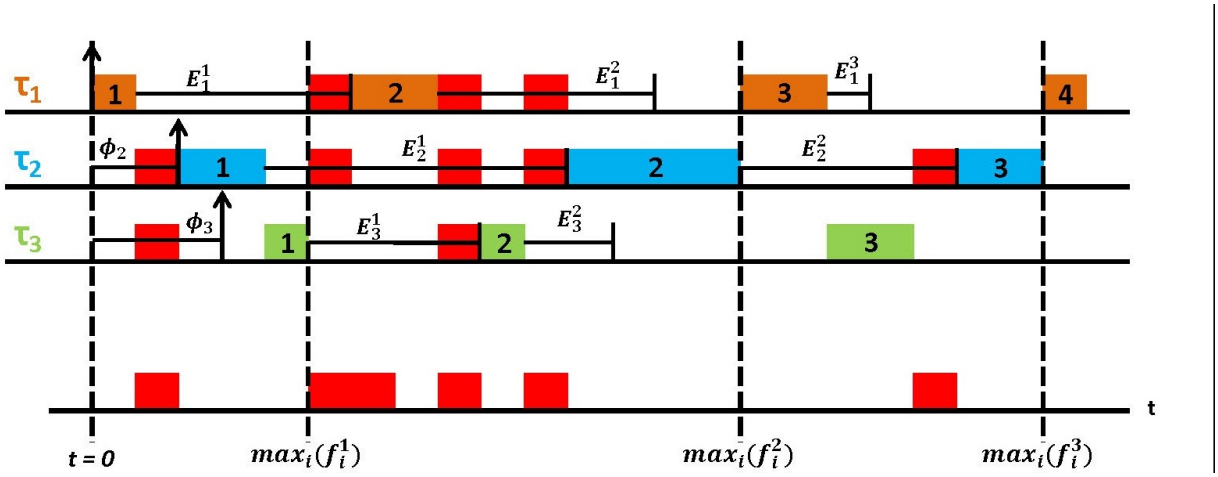

Figure 7: An example schedule is shown for three tasks with phase offsets. Processor idle time is shown in red and projected onto the timeline below. This plot includes dashed, vertical lines separate the timeline. W_{ϕ}^{τ} upperbounds idle time between $t = [0, max_i\left(f_i^1\right)]$, and W^1 upperbounds processor idle time during the domain of the first self-suspension $t = [max_i\left(f_i^1\right), max_i\left(f_i^2\right)]$.

Step 4) Subtask-to-Subtask Deadline Constraints

In Steps 1 and 3, we provided a lowerbound for the number of free subtasks that will execute during a free self-suspension, if that self-suspension produces processor idle time. We then upperbounded the processor idle time due to the set of free self-suspensions by computing the least amount of free task cost that will execute during a given self-suspension. However, our proof assumed no subtask-to-subtask deadline constraints. Now, we relax this constraint and calculate an upperbound on processor idle time due to embedded self-suspensions $W^{\boldsymbol{\tau}}_{embedded}$.

Recall under the JSF priority scheduling policy, 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., $E_x^j, x \in N \backslash i$).

To account for this lack of certainty, we conservatively treat embedded self-suspensions as task cost, as shown in Equations 28 and 29. Equation 28 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 *iff* τ_i^{j+1} is an embedded subtask.

Second, we restrict B_i^j such that the j^{th} 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 Equation 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 $\eta_i^j = \frac{|B_i^j|}{2}$ subtasks will execute during a given E_i^j, if E_i^j results in processor idle time.

$$W^{\boldsymbol{\tau}}_{embedded} = \sum_{i=1}^{n} \left(\sum_{j=1}^{m_i - 1} E_i^j \left(1 - x_i^{j+1}\right) \right) \tag{28}$$

$$x_i^j = \begin{cases} 1, & \text{if } \tau_i^j \in \boldsymbol{\tau_{free}} \\ 0, & \text{if } \tau_i^j \in \boldsymbol{\tau_{embedded}} \end{cases} \tag{29}$$

Having bounded the amount of processor idle time due to free and embedded self-suspensions and phase offsets, we now provide an upperbound on the time H^{τ}_{UB} the processor will take to complete all instances of each task in the hyperperiod (Equation 30). H denotes the hyperperiod of the task set, and H^{τ}_{LB} is defined as the sum over all task costs released during the hyperperiod. Recall that we are still assuming that $T_i = D_i = T_j = D_j, \forall i, j \in N$; thus, there is only one instance of each task in the hyperperiod. Under this

assumption, the task set is schedulable under JSF if $\frac{H^{\boldsymbol{\tau}}_{UB}}{H} \leq 1$.

$$H^{\boldsymbol{\tau}}_{UB} = H^{\boldsymbol{\tau}}_{LB} + W^{\boldsymbol{\tau}}_{phase} + W^{\boldsymbol{\tau}}_{free} + W^{\boldsymbol{\tau}}_{embedded} \tag{30}$$

$$H^{\tau}_{LB} = \sum_{i=1}^{n} \frac{H}{T_i} \sum_{j=1}^{m_i} C_i^j \tag{31}$$

Example for Subtask-to-Subtask Deadline Constraints

Consider our example from Figure 7, which is now augmented to include a subtask-to-subtask deadline $D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle} = 9$. The parameters of the task set are repeated here:

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

We apply Equation 28 to our example to upperbound 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 upperbound on processor idle time due to embedded self-suspensions is $\mathrm{W}^{\boldsymbol{\tau}}_{embedded} = 5$ (Equation 32).

$$\begin{aligned} W^{\boldsymbol{\tau}}_{embedded} &= \sum_{i=1}^{n} \left(\sum_{j=1}^{m_i - 1} E_i^j \left(1 - x_i^{j+1}\right) \right) \\ &= E_1^2 \\ &= 5 \end{aligned} \tag{32}$$

Because of the addition of this subtask-to-subtask deadline $D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$, the upperbound for $W^{\boldsymbol{\tau}}_{free}$ must be recomputed. Deadline $D^{s2s}_{\langle \tau_1^2, \tau_1^3 \rangle}$ embeds just one of the 2^{nd} 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 max 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 (Equation 33) and W_3^2 (Equation 34).

$$\begin{aligned} W_2^2 &= \max\left(\left(E_2^2 - \sum_{k=1}^{\eta_2^2} B_2^2(k) \right), 0 \right) \\ &= \max\left((5 - (1)) \right), 0) \\ &= 4 \end{aligned} \tag{33}$$

$$
\begin{aligned}
W_3^2 &= \max\left(\left(E_3^2 - \sum_{k=1}^{\eta_3^2} B_3^2(k)\right), 0\right) \\
&= \max\left((2 - (1))\right), 0) \\
&= 1
\end{aligned}
\tag{34}
$$

The new upperbound for idle time due to free self-suspensions is now calculated as shown in Equations 35 and 36.

$$
\begin{aligned}
W^2 &= \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^1\right) \\
&= \max(W_2^2, W_3^2) \\
&= \max(4, 1) \\
&= 4
\end{aligned}
\tag{35}
$$

$$
W_{free}^{\boldsymbol{\tau}} = \sum_j W^j = W^1 + W^2 = 5 + 4 = 9 \tag{36}
$$

Finally, the upperbound $H_{UB}^{\boldsymbol{\tau}}$ on the time required to process $\boldsymbol{\tau}$ can be computed via Equation 30. For our example, $H_{UB}^{\boldsymbol{\tau}} = 35$ (Equation 37). This upperbound 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 $H_{UB}^{\boldsymbol{\tau}} = 35$.

$$
\begin{aligned}
H_{UB}^{\boldsymbol{\tau}} &= H_{LB}^{\boldsymbol{\tau}} + W_{phase}^{\boldsymbol{\tau}} + W_{free}^{\boldsymbol{\tau}} + W_{embedded}^{\boldsymbol{\tau}} \\
&= 18 + 3 + 9 + 5 \\
&= 35
\end{aligned}
\tag{37}
$$

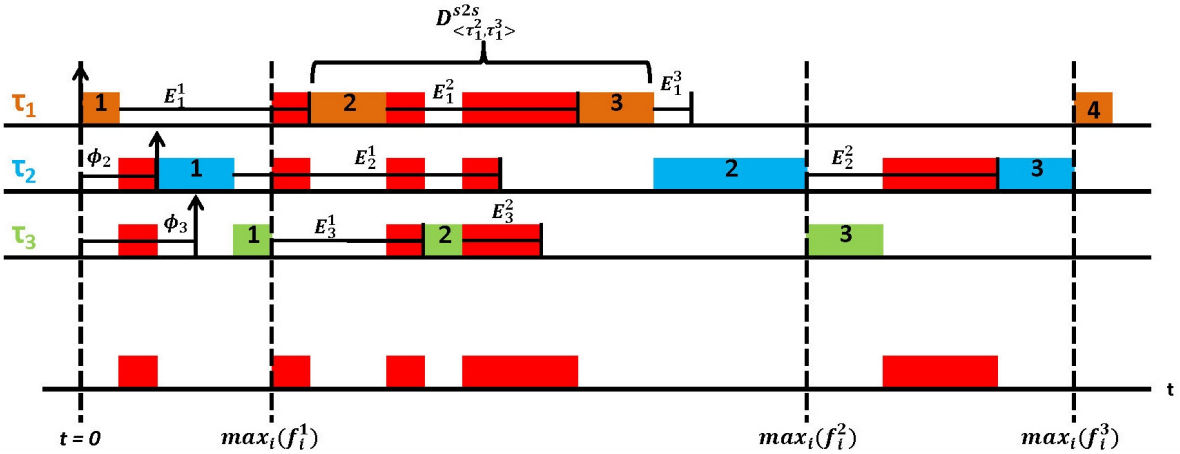

Figure 8: An example schedule is shown for three tasks with a subtask-to-subtask deadline constraint $D^{s2s}_{\langle\tau_1^2,\tau_1^3\rangle}$.

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 $T_i = T_j, \forall i,j \in N$ for this step. When we formulated our schedulability test of a self-suspending task set in Equation 30, we calculated an upperbound on the time the processor needs to execute the task set, H^{τ}_{UB}. Now we seek to upperbound the amount of time required to execute the final subtask τ_i^j for task τ_i, and we can utilize the methods already developed to upperbound this time.

To compute this bound we consider the largest subset of subtasks in $\boldsymbol{\tau}$, which we define as $\boldsymbol{\tau|_j} \subset \boldsymbol{\tau}$, that might execute before the task deadline for τ_i. If we find that $H^{\tau|_j}_{UB} \leq D^{abs}$, where D^{abs} is the absolute task deadline for τ_i, then we know that a processor scheduling under JSF will satisfy the task deadline for τ_i. We recall that, for Step 5, we have restricted the periods such that there is only one instance of each task in the hyperperiod. Thus, we have $D^{abs}_{i,1} = D_i + \phi_i$. In Step 6, we consider the more general case where each task may have multiple instances within the hyperperiod. For this scenario, the absolute deadline of the k^{th} instance of τ_i is $D^{abs}_{i,k} = D_i + T_i(k-1) + \phi_i$.

We present an algorithm named **testDeadline($\boldsymbol{\tau}$,D^{abs},j)** to perform this test. Pseudocode for **testDeadline($\boldsymbol{\tau}$,D^{abs},j)** is shown in Figure 9. This algorithm requires as input a task set τ, an absolute deadline D^{abs} for task deadline D_i, and the subtask index (i.e., index j in τ_i^j) of the last subtask associated with D_i (e.g., $j = m_i$ associated with D_i for $\tau_i \in \boldsymbol{\tau}$). The algorithm returns true if a guarantee can be provided that the processor will satisfy D_i under the JSF, and returns false otherwise.

In Lines 1-14, the algorithm computes $\boldsymbol{\tau|_j}$, the set of subtasks that may execute before D_i. In the absence of subtask-to-subtask deadline constraints, $\boldsymbol{\tau|_j}$ includes all subtasks $\tau_i^{j'}$ where $i \in N$ and $j' \in \{1, 2, \ldots, j\}$. In the case an subtask-to-subtask deadline spans subtask τ_x^j (in other words, a deadline $D_{\langle \tau_x^a, \tau_x^b \rangle}$ exists where $a \leq j$ and $b > j$), then the processor may be required to execute all embedded subtasks associated with the deadline before executing the final subtask for task τ_i. Therefore the embedded subtasks of $D_{\langle \tau_x^a, \tau_x^b \rangle}$ are also added to the set $\boldsymbol{\tau|_j}$. In Line 15, the algorithm tests the schedulability of $\boldsymbol{\tau|_j}$ using Equation 30.

Next we walk through the pseudocode for **testDeadline($\boldsymbol{\tau}$,D^{abs},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 $z = j$, 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 $\tau_x^{z+1} \in \boldsymbol{\tau_{free}}$, because τ_x^{z+1} will not execute before τ_i^j 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 (Line 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 upperbound on the fraction of time required by the processor to satisfy some subtask $\tau_{i,k}^j$ constrained by D^{abs} (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^{abs},j)** once for each task deadline.

testDeadline($\boldsymbol{\tau}$,D^{abs},j)

```
1:  τ|_j ← NULL
2:  for x = 1 to |τ| do
3:      z ← j
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^abs ≤ 1 //Using Eq. 30 then
16:     return TRUE
17: else
18:     return FALSE
19: end if
```

Figure 9: Pseudo-code for testDeadline($\boldsymbol{\tau}, \boldsymbol{D_i}, \boldsymbol{j}$), which tests whether a processor scheduling under JSF is guaranteed to satisfy a task deadline, D_i.

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, non-zero 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 $\boldsymbol{\tau^*} \supset \boldsymbol{\tau}$. 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 $\frac{H}{T_i}$ instances of $\tau_i \in \boldsymbol{\tau}$. A formal definition is shown in Equation 38, where $C_{i,k}^j$ and $E_{i,k}^j$ are the kth instance of the jth subtask cost and self-suspension of τ_i^*.

$$\begin{aligned}\tau_i^* :(\phi_i, (&C_{i,1}^1, E_{i,1}^1, \ldots, C_{i,1}^{m_i}, C_{i,2}^1, E_{i,2}^1, \ldots, C_{i,2}^{m_i}, \\ &\ldots, C_{i,k}^1, E_{i,k}^1, \ldots, C_{i,k}^{m_i}), D_i^* = H, T_i^* = H)\end{aligned} \tag{38}$$

We aim to devise a test where τ_i^* is schedulable if $\frac{H_{UB}^{\boldsymbol{\tau}^*}}{D_i^*} \leq 1$ and if the task deadline D_i for each release of τ_i is satisfied for all tasks and releases. 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 all other j^{th} subtasks in $\boldsymbol{\tau^*}$. Consider a scenario where we have just completed the last subtask $\tau_{i,k}^j$ of the k^{th} instance of τ_i. We do not know if the first subtask of the $k+1^{th}$ instance of τ_i will be released by the time the processor finishes executing the other j^{th} subtasks from $\boldsymbol{\tau^*}$. We would like to shift the index of each subtask in the new instance to some $j' \geq j$ such that we can guarantee the subtask will be released by the completion time of all other $(j'-1)^{th}$ subtasks.

Second, we need to check that each task deadline $D_{i,k}$ for each instance k of each task τ_i released during the hyperperiod will be satisfied. To do this check, we compose a paired list of the subtask indices j in $\boldsymbol{\tau^*}$ that correspond to the last subtasks for each task instance, and their associated deadlines. We then apply **testDeadline**($\boldsymbol{\tau}$,D_i,j) for each pair of deadlines and subtask indices in our list. Finally, we must determine an upperbound, $H_{UB}^{\boldsymbol{\tau}^*}$, on the temporal resources required to execute $\boldsymbol{\tau^*}$ using Equation 30. If $\frac{H_{UB}^{\boldsymbol{\tau}^*}}{H} \leq 1$, 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 Figure 10, 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 $\mathbf{I}$, where $\mathbf{I}$[i] corresponds to the instance number of the last instance of τ_i that we have added to $\boldsymbol{\tau^*}$. Note that after initialization, $\mathbf{I}$[i]$= 1$ for all i. In Line 3, we initialize a vector $\mathbf{J}$, where $\mathbf{J}$[i] corresponds to the j subtask index of τ_i^{*j} for instance $\mathbf{I}$[i], the last task instance added to τ_i^*. The mapping to new subtask indices is constructed in $\mathbf{J}$ to ensure that the $(j+1)^{th}$ subtasks in $\boldsymbol{\tau^*}$ will be released by the time the processor finishes executing the set of j^{th} subtasks.

We use $\mathbf{D}$[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. $\mathbf{D}$[i][k] returns the subtask index j in $\boldsymbol{\tau^*}$ of instance k of τ_i. In Line 4, $\mathbf{D}$[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 (Line 7-31). 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 $j = \text{counter}$ subtask index is greater than the index of the last subtask we added to τ_i^*) (Line 14).

If the two conditions in Line 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 j^{th} 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 $T_i * \mathbf{I}[i] + \phi_i$. Therefore, if the sum of the cost 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, counter is incremented (Line 27). 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 *iff* each task deadline $D_{i,k}$ for each instance k of each task τ_i released during the hyperperiod is guaranteed to be satisfied.

VI.A. 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 upperbound $H_{UB}^{\boldsymbol{\tau}}$ on the time required to process task (or subtask) using Equation 30.

$H_{UB}^{\boldsymbol{\tau}}$ is comprised of four terms. The first term $H_{LB}^{\boldsymbol{\tau}}$ is simply the sum over the cost of the tasks (Equation 31). The next three terms upperbound the amount of processor idle time due to phase offsets, and free and embedded self-suspensions. $W_{\phi}^{\boldsymbol{\tau}}$ (Equation 22) accounts for processor idle time due to phase offsets and equals the maximum over all phase offsets. $W_{free}^{\boldsymbol{\tau}}$ upperbounds processor idle time due to free

constructTaskSuperSet($\boldsymbol{\tau}$)

1: $\boldsymbol{\tau}^*$ ← Initialize to $\boldsymbol{\tau}$
2: **I**[i] ← $1, \forall i \in N$
3: **J**[i] ← $m_i, \forall i \in N$
4: **D**[i][k] ← $m_i, \forall i \in N, k = 1$
5: counter ← 2
6: $H_{LB} \leftarrow 0$
7: **while** TRUE **do**
8: **if** **I**[i] = $\frac{H}{T_i}, \forall i \in N$ **then**
9: break
10: **end if**
11: $H_{LB} \leftarrow H_{LB} + \sum_{i=1}^{n} C_i^{*(\text{counter}-1)}$
12: **for** i = 1 to n **do**
13: **if** **I**[i] < $\frac{H}{T_i}$ **then**
14: **if** counter > **J**[i] **then**
15: **if** $H_{LB} \geq T_i * \mathbf{I}[\text{i}] + \phi_i$ **then**
16: **I**[i] ←**I**[i]+1
17: $\tau_i^{*(\text{counter}+y-1)} \leftarrow \tau_i^y, \forall y \in \{1, 2, \ldots, m_i\}$
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: $H_{LB} = \min_i (T_i * \boldsymbol{I}[i] + \phi_i)$
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 $\frac{H}{T_i}$ **do**
33: $D_{i,k} \leftarrow D_i + T_i(k-1) + \phi_i$
34: j ← **D**[i][k]
35: **if** **testDeadline**($\boldsymbol{\tau}^*$,$D_{i,k}$,j) = FALSE **then**
36: **return** NULL
37: **end if**
38: **end for**
39: **end for**
40: **return** $\boldsymbol{\tau}^*$

Figure 10: Pseudo-code for constructTaskSuperSet($\boldsymbol{\tau}$), which constructs a task superset, $\boldsymbol{\tau}^*$ for $\boldsymbol{\tau}$.

self-suspensions by considering the worst-case interleaving of subtasks during free self-suspensions (Equation 24). Lastly, $W^{\boldsymbol{\tau}}_{embedded}$ upperbounds processor idle time due to self-suspensions that are constrained by subtask-to-subtask deadlines (Equation 28).

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. **constructTaskSuperSet**($\boldsymbol{\tau}$) constructs $\boldsymbol{\tau}^*$ in a careful way such that the processor will schedule $\boldsymbol{\tau}$ according to JSF using jth indeces of subtasks as specified in $\boldsymbol{\tau}^*$.

VII. Uniprocessor Scheduling Algorithm for Self-Suspending Task Sets

In Section VI, we developed a uniprocessor schedulability test for hard, non-preemptive, self-suspending task sets. This schedulability test relies on a processor operating using the jth Subtask First scheduling priority. JSF requires that all j^{th} subtasks are processed before any $(j+1)^{th}$ *free* subtasks, where a subtask τ_i^{j+1} is free *iff* 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 utilizes the processor.

VII.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 $\boldsymbol{\tau}^*$.

Pseudo-code for the JSF Scheduling Algorithm is shown in Figure 11. 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 an instance a and b of τ_i and τ_x, respectively. There are corresponding subtasks τ_i^{k*} and τ_x^{z*} in $\boldsymbol{\tau}^*$ such that $j \leq k$ and $y \leq z$. If both τ_i^j and τ_x^y are free subtasks, and $j < y$, then the processor does consider τ_i^j for execution at time t, but does not consider τ_x^y for execution, according to the JSF prioritization. Line 8 prunes all such released subtasks $\tau_{x,b}^y$.

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 (EDF).

Line 10 iterates over all 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 Section 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.

```
JSFSchedulingAlgorithm(τ,τ*)
 1: t ← −1
 2: while true do
 3:     t ← t +1
 4:     if processor is busy then
 5:         continue
 6:     end if
 7:     releasedSubtasks ← getReleasedSubtasks(τ)
 8:     JSFsubtasks ← pruneForJSF(releasedSubtasks,τ*)
 9:     prioritizedSubtasks ← prioritize(JSFsubtasks)
10:     for counter = 1 → |prioritizedSubtasks| do
11:         τ^j_{i,k} ← prioritizedSubtasks[counter];
12:         if russianDollsTest(τ^j_{i,k}) then
13:             process(τ^j_{i,k})
14:             break
15:         end if
16:     end for
17: end while
```

Figure 11: This figure provides pseudo-code for JSFSchedulingAlgorithm(τ,τ^*). This algorithm schedules self-suspending task sets on a uniprocessor.

VII.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,[31,32] the purpose of which is to determine whether an event must or may execute before or after a set of activities.[33] 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 7) and an *active subtask* (Definition 8).

Definition 7 Active Subtask-to-Subtask Deadline - *A subtask-to-subtask deadline* $D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}$ *is considered* active *between* $s_i^j \leq t \leq f_i^b$.

Definition 8 Active Subtask - *A subtask is active at time t if it has been released and is yet unprocessed at time t and is directly constrained by an active subtask-to-subtask deadline.*

VII.B.1. Walk-through of Pseudocode

Pseudocode describing the uniprocessor Russian Dolls Test is shown in Figure 12. The Russian Dolls Test takes as input a subtask τ_i^j, the task set $\boldsymbol{\tau}$, 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 τ_i^j at time t, we must consider two scenarios. First, if processing τ_i^j does not activate a subtask-to-subtask deadline, then we merely need to guarantee that processing τ_i^j leaves enough time for the processor to finish executing the set of active subtasks. Second, if processing τ_i^j does activate a subtask-to-subtask deadline $D_{\langle \tau_i^j, \tau_i^b \rangle}$, then we must also consider whether the processor will have enough time to attend to subtasks $\{\tau_i^q | j < q \leq b\}$ in addition to the other active subtasks.

In Line 1, the test iterates over all active subtasks $\tau_{x,z}^y$ (Definition 8) not including τ_i^j. In Lines 2-4, the test considers the direct effect of processing τ_i^j at time t. Line 2 tests whether the processor can nest the execution of τ_i^j within the laxity of $\tau_{x,z}^y$'s deadline. If no such nesting is possible, then the test returns *false* thus prohibiting the processing of τ_i^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 7), then we must consider the indirect effects of this activation on the other subtasks constrained by this deadline constraint. If this activation would occur (Line 5), the test iterate over all subtasks $\tau_i^q | j < q \leq b$ constrained by $D^{s2s}_{\langle \tau_i^j, \tau_i^b \rangle}$ (Line 6) except for $\tau_{i,k}^j$, which is accounted for in Line 2.

We then determine whether the processor can nest the execution of τ_i^q within the laxity of τ_x^y's deadline or vice versa (Line 7). 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 8). 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 13).

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 $m_i \sim U(1, 2n)$, with n being the number of tasks. If

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

for all $\tau_{x,z}^{y} \in \boldsymbol{\tau_{active}} \backslash \tau_i^j$ **do**
  if $\left(t + C_i^j > d_{x,z}^y - C_x^y\right)$ **then**
    return false
  end if
  if $\exists D_{\langle \tau_i^j, \tau_i^b \rangle}$ **then**
    for all $\tau_i^q | j < q \leq b$ **do**
      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**
        return false
      end if
    end for
  end if
end for
return true

Figure 12: Pseudocode describing the uniprocessor Russian Dolls Test.

$m_i = 1$, then that task does not have a self-suspension. The subtask cost and self-suspension durations are drawn from uniform distributions $C_i^j \sim U(1, 10)$ and $E_i^j \sim U(1, 10)$, respectively. Task periods are drawn from a uniform distribution such that $T_i \sim U(\sum_{i,j} C_i^j, 2\sum_{i,j} C_i^j)$. Lastly, task deadlines are drawn from a uniform distribution such that $D_i \sim U(\sum_{i,j} C_i^j, T_i)$.

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

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

VIII.A. Tightness of the Schedulability Test and Scheduling Algorithm

The metric we use to evaluate the tightness of our schedulability test is the percentage of self-suspension time our method treats as task cost, as calculated in Equation 39.

$$\hat{E} = \frac{W_{free}^{\boldsymbol{\tau}} + W_{embedded}^{\boldsymbol{\tau}}}{\sum_{i,j} E_i^j} * 100\% \tag{39}$$

This metric provides a comparison between our method and the naive worst-case analysis that treats all self-suspensions as idle time. Similarly, we evaluate the tightness of our scheduling algorithm using the percentage of self-suspension time that the processor is idle.

VIII.A.1. Traditional Self-Suspending Task Model

In Figure 13, we show the empirical tightness of our schedulability test and scheduling algorithm as a function of the size of the task set. Recall that the traditional model (Equation 1) does not include subtask-to-subtask deadlines.

For small problem sizes, the schedulability test significantly overestimates the amount of time the processor will idle due to self-suspensions while processing the task set. However, the schedulability test and scheduling algorithm quickly converge as task size increases. Both the schedulability test and scheduling algorithm approach approximately 10% idle time.

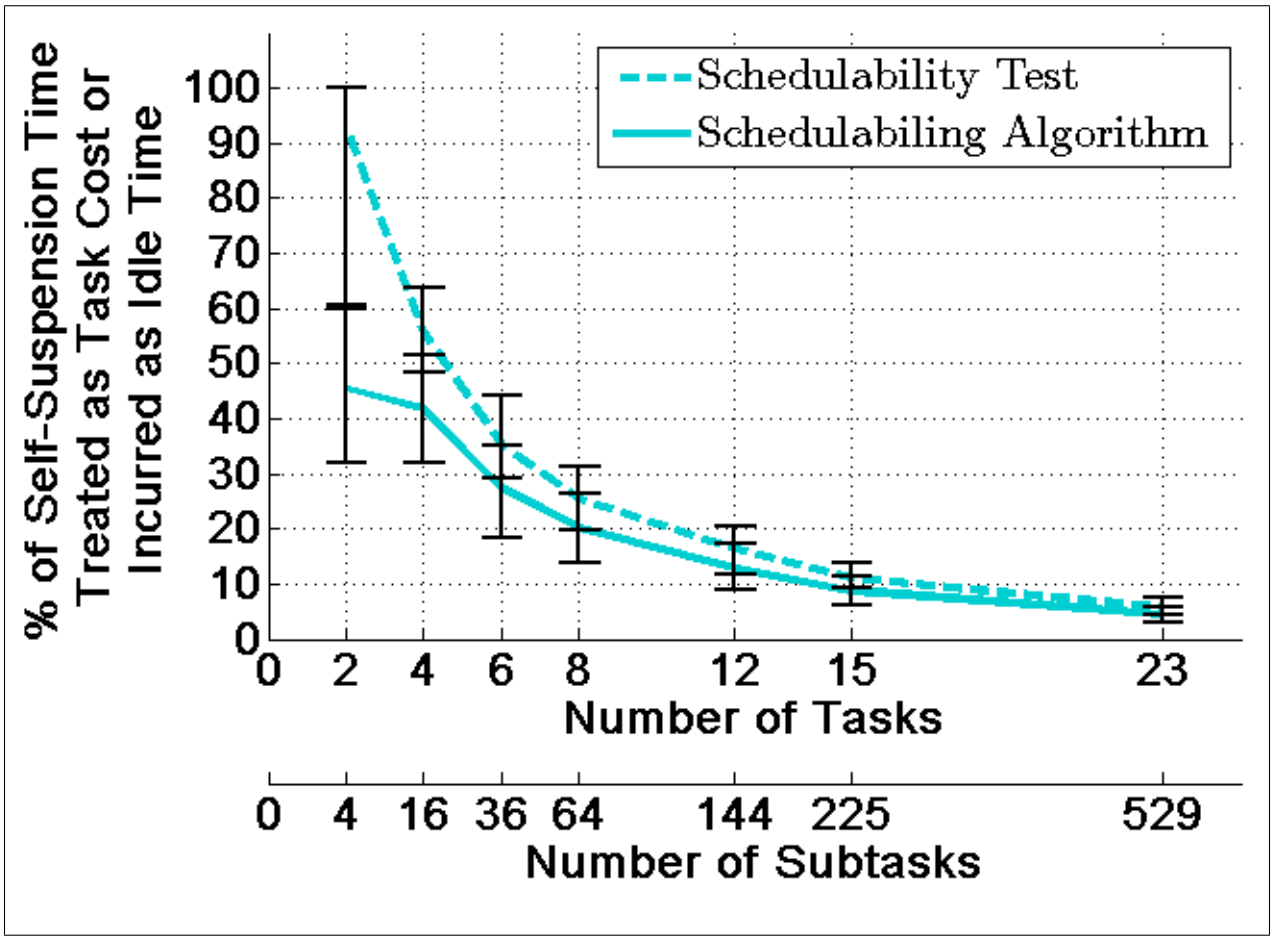

Figure 13: This plot shows the tightness of our schedulability test and scheduling algorithm for the traditional self-suspending task model. For the schedulability test, the plot shows the amount of self-suspension time that is treated as task cost to account for processor idle time. For the scheduling algorithm, the plot shows the actual amount of processor idle time due to self-suspensions. Both measures are normalized to the sum of the duration of all self-suspensions.

VIII.B. Augmented Self-Suspending Task Model

Next, we evaluate tightness of the JSF schedulability test and scheduling algorithm for the self-suspending task 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 one-fourth (Figure 14) and one-half (Figure 15) of the subtasks released during the hyperperiod are constrained by subtask-to-subtask deadlines.

Recall that our schedulability test treats all self-suspensions constrained by subtask-to-subtask deadlines (embedded self-suspensions) as 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 processor idle time.

While the tightness of the schedulability test quickly approaches that of the scheduling algorithm for the traditional model (Figure 13), we do not see that same behavior for task sets with subtask-to-subtask deadlines. The Russian Dolls Test allows the processor to utilize much of the embedded self-suspension time treated as task cost by the schedulability test. Nonetheless, our methods are tight for task sets that have a relatively low number of subtasks constrained by subtask-to-subtask deadlines. 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.

VIII.C. Computational Complexity

VIII.C.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 Equation 30 for $\boldsymbol{\tau}^*$ is upperbounded by $O\left(n^2 m_{max} \frac{H}{T_{min}}\right)$ where $O\left(n m_{max} \frac{H}{T_{min}}\right)$ bounds the number of self-suspensions in $\boldsymbol{\tau}^*$. The complexity of **testDeadline**() is dominated by evaluating Equation 30. In turn, **constructTaskSuperset**() is dominated by $O\left(n \frac{H}{T_{min}}\right)$ calls to **testDeadline**(). Thus, for the algorithm we have presented in Figures 9 and 10, 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 $O\left(n^2 m_{max} \frac{H}{T_{min}}\right)$ by caching the result of intermediate steps in evaluating Equation 30.

We provide empirical validation of the computational time of the JSF schedulability test in Figure 16.

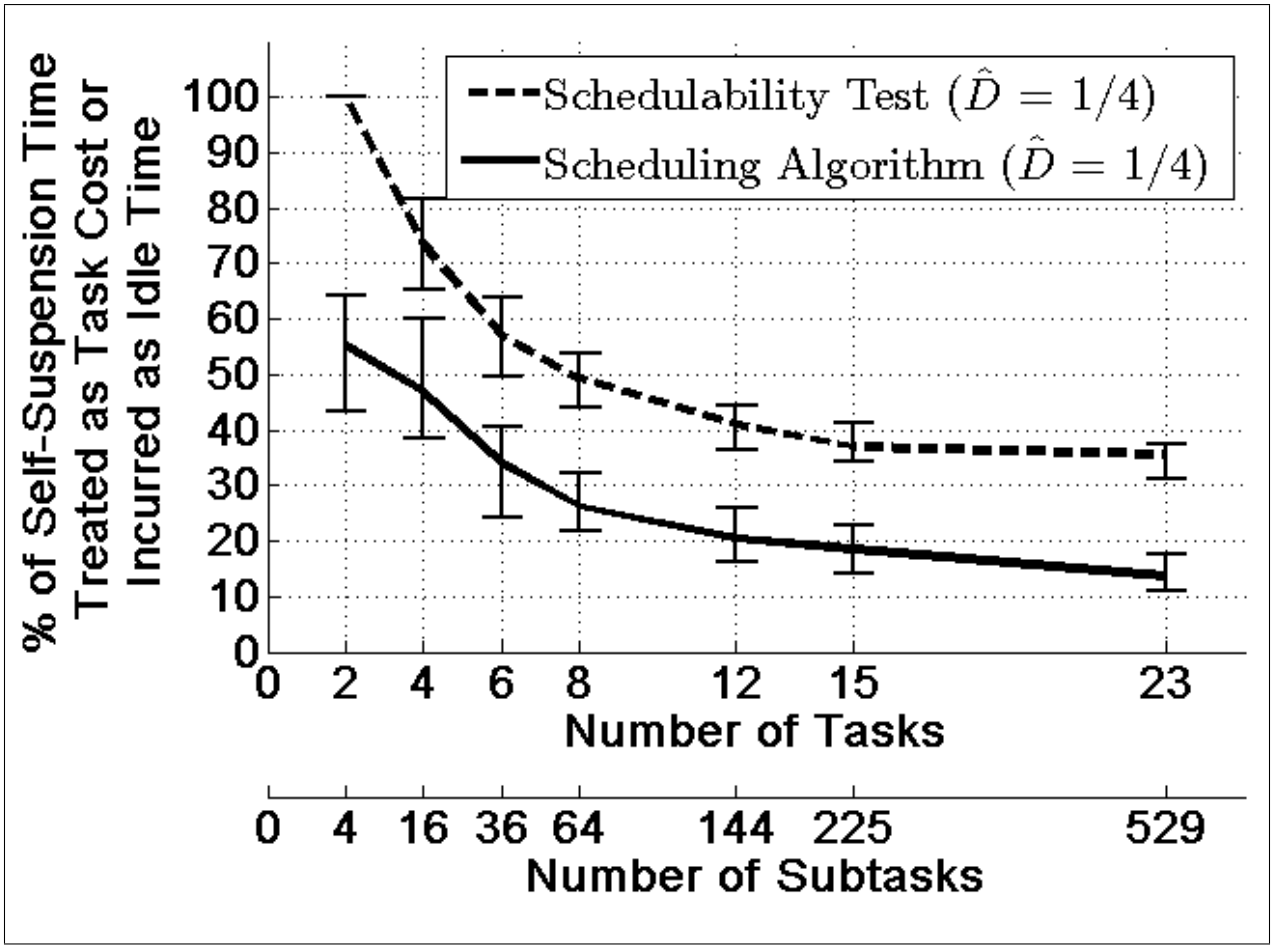

Figure 14: This plot shows the tightness of our schedulability test and scheduling algorithm for the augmented self-suspending task model where one quarter $\left(\hat{D} = \frac{1}{4}\right)$ of the subtasks released during the hyperperiod are constrained by subtask-to-subtask deadlines. For the schedulability test, the plot shows the amount of self-suspension time that is treated as task cost to account for processor idle time. For the scheduling algorithm, the plot shows the actual amount of processor idle time due to self-suspensions. Both measures are normalized to the sum of the duration of all self-suspensions.

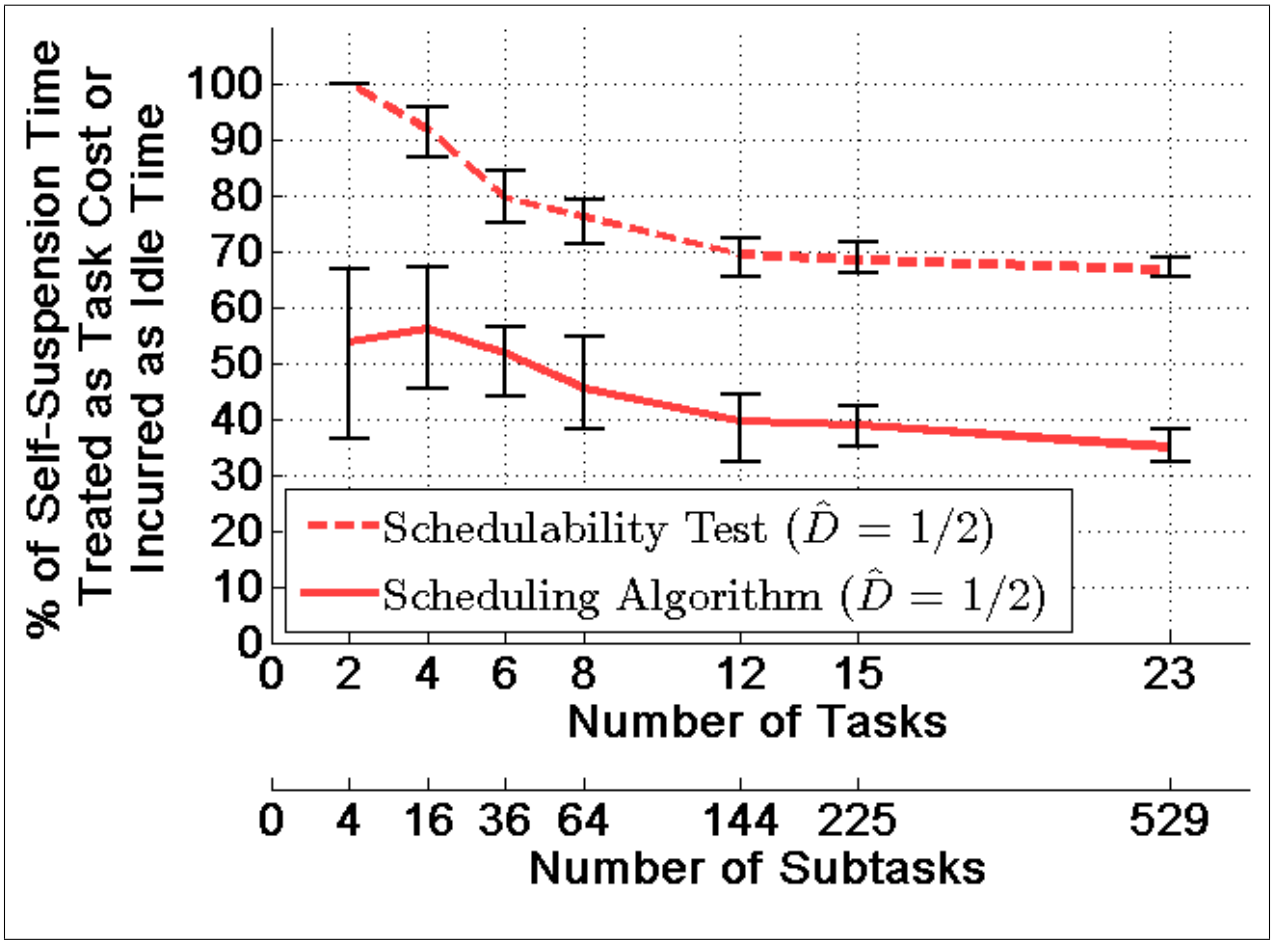

Figure 15: This plot shows the tightness of our schedulability test and scheduling algorithm for the augmented self-suspending task model where one half $\left(\hat{D} = \frac{1}{2}\right)$ of the subtasks released during the hyperperiod are constrained by subtask-to-subtask deadlines. The schedulability test plot shows the amount of self-suspension time that is treated as task cost to account for processor idle time. The scheduling algorithm plot shows the actual amount of processor idle time due to self-suspensions. Both measures are normalized to the sum of the duration of all self-suspensions.

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 2.30GHz and 8 GB of RAM. With a more efficient implementation, we expect the computation time to significantly decrease.

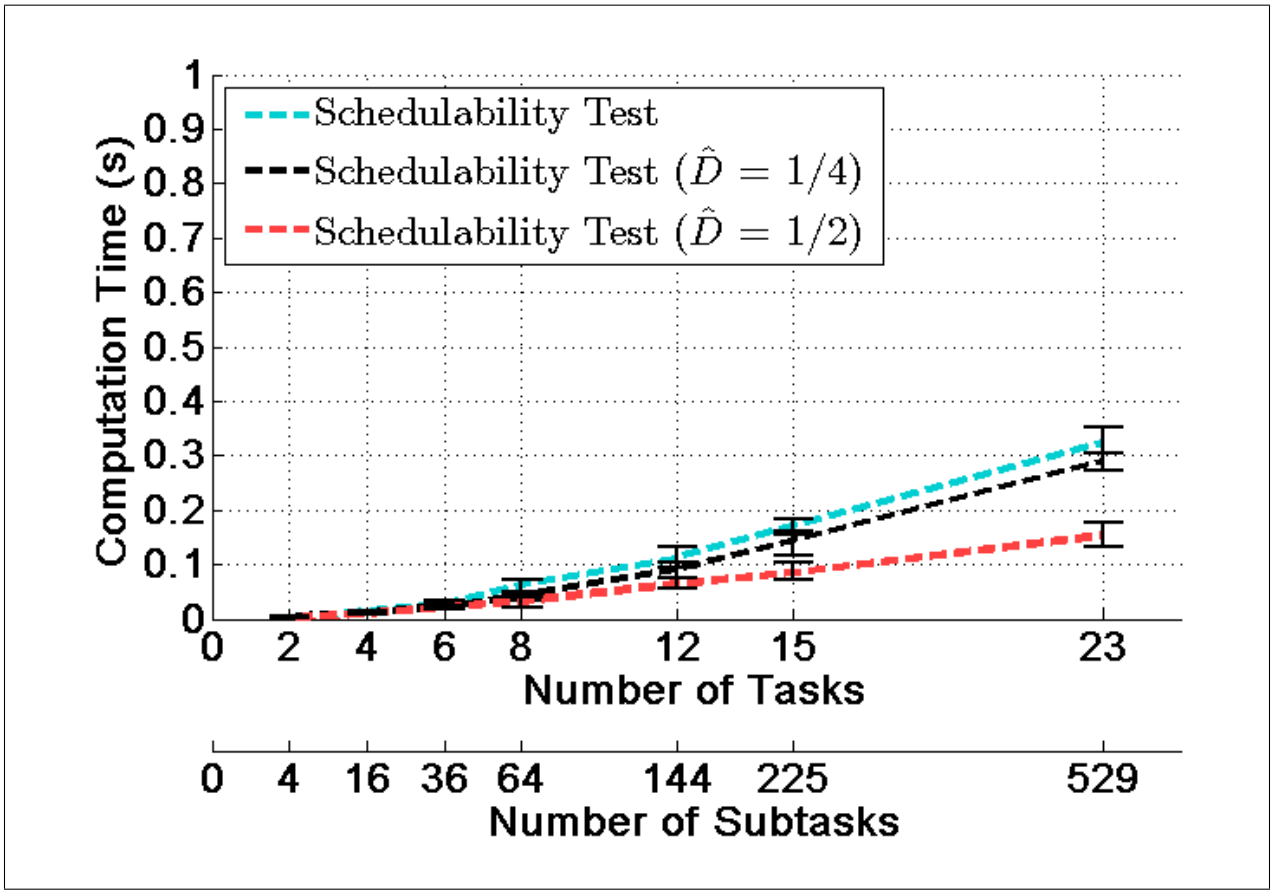

Figure 16: This plot shows the computation time of our polynomial time schedulability test for the traditional model as well as for task sets with subtask-to-subtask deadlines where $\hat{D} \in \{\frac{1}{4}, \frac{1}{2}\}$.

VIII.C.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 pair-wise comparison of all active subtasks. In the worst case, there are $O\left(n \sum_i m_i W\right)$ active subtasks. Thus, the complexity of our scheduling algorithm is $O\left(n^2 m_{max}\right)$ per time step.

IX. Conclusion

In this paper, we present a polynomial time solution to the open problem of determining the feasibility of hard, periodic, non-preemptive, self-suspending task sets with any number of self-suspensions in each task, phase offsets, and deadlines less than or equal to periods. We also generalize the self-suspending task model and our schedulability test to handle task sets with subtask-to-subtask deadlines, which constrain the upperbound 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.

Our schedulability test works by leveraging a novel priority scheduling policy for self-suspending task sets, called j^{th} Subtask First (JSF), that restricts the behavior of a self-suspending task set so as to provide an analytical basis for an informative schedulability test. We prove the correctness of schedulability test.

Furthermore, we also introduce an online consistency test, which we call the Russian Dolls Test, that ensures temporal feasibility during runtime when scheduling against subtask-to-subtask deadlines. We empirically evaluate the tightness and computational complexity of our methods. For the standard self-suspending task model our method enables the processor to effectively use 95% of self-suspension time to process tasks.

Acknowledgment

Funding for this project was provided by Boeing Research and Technology and The National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) under grant number 2388357.

References

[1]J. Liu, *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2000.

[2]J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. Kluwer Academic Publishers, 1998.

[3]N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.

[4]I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, and M.-P. Hong, "Real-time scheduling of tasks that contain the external blocking intervals," in *Proceedings of the Conference on Real-time Computing Systems and Applications*, 1995.

[5]C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, 1973.

[6]L. Ming, "Scheduling of the inter-dependent messages in real-time communication," in *Proceedings of the First International Workshop on Real-Time Computing Systems and Applications*, (Seoul, Korea), December 21-22 1994.

[7]K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Open problems in scheduling self-suspending tasks," in *Proceedings of the Real-Time Scheduling Open Problems Seminar (RTSOPS)*, (Brussels, Belgium), July 6 2010.

[8]C. Liu and J. H. Anderson, "An O(m) analysis technique for supporting real-time self-suspending task systems," in *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2012.

[9]P. Richard, "On the complexity of scheduling real-time tasks with self-suspensions on one processor," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, (Porto, Portugal), Julie 2-4 2003.

[10]F. Ridouard and P. Richard, "Worst-case analysis of feasibility test for self-suspending task sets,"

[11]K. Lakshmanan and R. R. Rajkumar, "Scheduling self-suspending real-time tasks with rate-monotonic priorities," in *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (Stockholm, Sweden), April 12-15 2010.

[12]L. Brunet, H.-L. Choi, and J. P. How, "Consensus-based auction approaches for decentralized task assignment," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference (GNC)*, (Honolulu, HI), 2008.

[13]M. Rekik, J.-F. Cordeau, and F. Soumis, "Consensus-based decentralized auctions for robust task allocation," *IEEE Transactions on Robotics*, vol. 25, pp. 912–926, 2004.

[14]M. C. Gombolay, R. J. Wilcox, and J. A. Shah, "Fast scheduling of multi-robot teams with temporospatial constrints," in *Proceedings of the Robots: Science and Systems (RSS)*, (Berlin, Germany), June 24-28 2013.

[15]J. N. Hooker, "A hybrid method for planning and scheduling," tech. rep., Pittsburgh, Tepper School of Business, Carnegie Mellon University, 2004.

[16]J.-F. Cordeau, G. Stojković, F. Soumis, and J. Desrosiers, "Benders decomposition for simultaneous aicraft routing and crew scheduling," *Transportation Science*, vol. 35, no. 4, pp. 357–388, 2001.

[17]E. Castro and S. Petrovic, "Combined mathematical programming and heuristics for a radiotherapy pre-treatment scheduling problem," *Journal of Scheduling*, vol. 15, no. 3, pp. 333–346, 2012.

[18]J. Chen and R. G. Askin, "Project selection, scheduling and resource allocation with time dependent returns," *European Journal of Operational Research*, vol. 193, pp. 23–34, 2009.

[19]D. Bertsimas and R. Weismantel, *Optimization over Integers*. Belmont: Dynamic Ideas, 2005.

[20]R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *AI*, vol. 49, no. 1, 1991.

[21]N. Muscettola, P. Morris, and I. Tsamardinos, "Reformulating temporal plans for efficient execution," in *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR&R)*, (Trento, Italy), June 2-5 1998.

[22]M. G. Harbour and J. C. Palencia, "Response time analysis for tasks scheduled under EDF within fixed priorities," in *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2003.

[23]F. Ridouard and P. Richard, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Proceedings of the Real-Time and Network Systems (RTNS)*, (Poitiers, France), May 30-31 2006.

[24]Y. Abdeddaïm and D. Masson, "Scheduling self-suspending periodic real-time tasks using model checking," in *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2011.

[25]A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea, "TSAT++: an open platform for satisfiability modulo theories," in *Proceedings of the 2nd workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[26]B. Nelson and T. K. Kumar, "CircuitTSAT: A solver for large instances of the disjunctive temporal problem," in *Proceedings of the ICAPS* (J. Rintanen, B. Nebel, J. C. Beck, and E. A. Hansen, eds.), 2008.

[27]U. C. Devi, "An improved schedulability test for uniprocessor periodic task systems," in *Proceedings of the 16th Euromicro Technical Committee on Real-Time Systems*, (Catania, Italy), June 30 - July 2 2003.

[28]C. Liu and J. H. Anderson, "Task scheduling with self-suspensions in soft real-time multiprocessor systems," in *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, (Washington DC, U.S.A.), December 1-4 2009.

[29]R. R. Rajkumar, "Dealing with self-suspending period tasks," tech. rep., IBM, Thomas J. Watson Research Center, Armonk, 1991.

[30]M. C. Gombolay and J. A. Shah, "Multiprocessor scheduler for task sets with well-formed precedence relations, temporal deadlines, and wait constraints," in *Proceedings of the AIAA Infotech@Aerospace*, 2012.

[31]P. Laborie, "Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results," *Artificial Intelligence*, vol. 143, no. 2, pp. 151–188, 2003.

[32]P. Vilím, R. Barták, and O. Čepek, "Extension of $o(n \log n)$ filtering algorithms for the unary resource constraint to optional activities," *Constraints*, vol. 10, no. 4, pp. 403–425, 2005.

[33]P. Baptiste and C. L. Pape, "Edge-finding consraint propagation algorithms for disjunctive and cumulative scheduling," in *Proceedings of the 15^{th} Workshop of the U.K. Planning and Special Interest Group*, (Liverpool, U.K.), November 21-22 1996.