Fast Methods for Scheduling with Applications to Real-Time Systems and Large-Scale, Robotic Manufacturing of Aerospace Structures

by

Matthew C. Gombolay

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Aeronautics and Astronautics
June 7, 2013

Certified by .....
Julie A. Shah
Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Eytan H. Modiano
Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Fast Methods for Scheduling with Applications to Real-Time Systems and Large-Scale, Robotic Manufacturing of Aerospace Structures

by

Matthew C. Gombolay

Submitted to the Department of Aeronautics and Astronautics
on June 7, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

Across the aerospace and automotive manufacturing industries, there is a push to remove the cage around large, industrial robots and integrate right-sized, safe versions into the human labor force. By integrating robots into the labor force, humans can be freed to focus on value-added tasks (e.g. dexterous assembly) while the robots perform the non-value-added tasks (e.g. fetching parts). For this integration to be successful, the robots need to ability to reschedule their tasks online in response to unanticipated changes in the parameters of the manufacturing process.

The problem of task allocation and scheduling is NP-Hard. To achieve good scalability characteristics, prior approaches to autonomous task allocation and scheduling use decomposition and distributed techniques. These methods work well for domains such as UAV scheduling when the temporospatial constraints can be decoupled or when low network bandwidth makes inter-agent communication difficult. However, the advantages of these methods are mitigated in the factory setting where the temporospatial constraints are tightly inter-coupled from the humans and robots working in close proximity and where there is sufficient network bandwidth.

In this thesis, I present a system, called Tercio, that solves large-scale scheduling problems by combining mixed-integer linear programming to perform the agent allocation and a real-time scheduling simulation to sequence the task set. Tercio generates near optimal schedules for 10 agents and 500 work packages in less than 20 seconds on average and has been demonstrated in a multi-robot hardware test bed. My primary technical contributions are fast, near-optimal, real-time systems methods for scheduling and testing the schedulability of task sets. I also present a pilot study that investigates what level of control the Tercio should give human workers over their robotic teammates to maximize system efficiency and human satisfaction.

Thesis Supervisor: Julie A. Shah
Title: Assistant Professor of Aeronautics and Astronautics

Acknowledgments

Personal Acknowledgments

Over the past two years, I have had the unwavering support of a number of individuals and groups that I would like to personally thank. Assistant Professor Julie Shah, my advisor and mentor has inspired me, constructively criticized my work, and helped to form me into a capable researcher. I thank her for her energetic support. Further, I want to thank Professor Julie Shah's Interactive Robotics Group (IRG) for their support in my research as peer-reviewers, test subjects, and friends.

My family has been a constant support for my entire life. My parents, Craig and Lauren, sister, Alli, and Grandparents, Fred and Vivan, have given me the love and support necessary for me to reach my goals. I treasure their support. I also want to thank Grace, who has been my comfort, companion, and best friend. Lastly, and most importantly, I praise God for the breath he breathes in my lungs, that I might bring the gospel of Jesus Christ to those who do not know Him.

Funding

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.

Contents

List of Figures

Chapter 1

Introduction

Robotic systems are increasingly entering domains previously occupied exclusively by humans. In manufacturing, there is strong economic motivation to enable human and robotic agents to work in concert to perform traditionally manual work. This integration requires a choreography of human and robotic work that meets upper-bound and lowerbound temporal deadlines on task completion (e.g. assigned work must be completed within one shift) and spatial restrictions on agent proximity (e.g. robots must maintain four meter separation from other agents) to support safe and efficient human-robot co-work. Figure 1-1 shows an example scenario where a human quality assurance agent must co-habit work space with robots without delaying the manufacturing process. The multi-agent coordination problem with temporospatial constraints can be readily formulated as a mixed-integer linear program (MILP).

1.1 Formal Problem Description

$$\min(z),\ z = \max_{i,j} (f_j - s_i) + g\left(x, A, s, f, \boldsymbol{\tau}\right) \tag{1.1}$$

subject to

$$\sum_{a \in \boldsymbol{A}} A_{a,i} = 1, \forall i \in \boldsymbol{\tau} \tag{1.2}$$

$$lb_{i,j} \leq f_i - s_j \leq ub_{i,j}, \forall (i,j) \in \boldsymbol{\gamma} \tag{1.3}$$

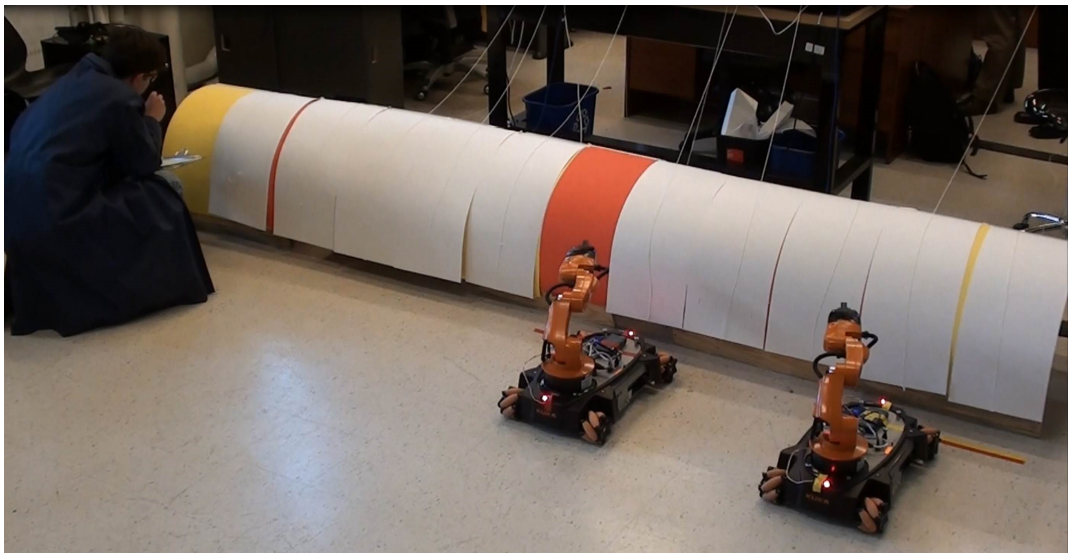

Figure 1-1: Example of a team of robots assigned to tasks on a mock fuselage. These robots must coordinate their efforts as to allow a human quality assurance agent the time and space necessary to inspect progress on the fuselage.

$$f_i - s_i \geq lb_{a,i} - M(1 - A_{i,k}), \forall \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A} \tag{1.4}$$

$$f_i - s_i \leq ub_{a,i} + M(1 - A_{a,i}), \forall \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A} \tag{1.5}$$

$$s_j - f_i \geq M(1 - x_{i,j}), \forall \tau_i, \tau_j \in \boldsymbol{\tau_R} \tag{1.6}$$

$$s_i - f_j \geq Mx_{i,j}, \forall \tau_i, \tau_j \in \boldsymbol{\tau_R} \tag{1.7}$$

$$s_j - f_i \geq M(1 - x_{i,j}) + M(2 - A_{a,i} - A_{a,j}) \forall \tau_i, \tau_j \in \boldsymbol{\tau} \tag{1.8}$$

$$s_i - f_j \geq Mx_{i,j} + M(2 - A_{a,i} - A_{a,j}) \forall \tau_i, \tau_j \in \boldsymbol{\tau} \tag{1.9}$$

In this formulation, $A_{a,i} \in \{0, 1\}$ is a binary decision variable for the assignment of agent a to task τ_i, $x_{i,j} \in \{0, 1\}$ is a binary decision variable specifying whether τ_i comes before or after τ_j, and $s_i, f_i \in [0, \infty)$ are the start and finish times of τ_i. $\boldsymbol{A}$ is the set of all agents a, $\boldsymbol{\tau}$ is the set of all tasks, τ_i, $\boldsymbol{\tau_R}$ is the set of all the set of task pairs (i, j) that are separated by less than the allowable spatial proximity. $\boldsymbol{\gamma}$ is the set of all temporal constraints defined by the task set. M is an artificial variable set to a large positive number, and is used to encode conditional constraints.

The MILP is defined by an objective function (Equation 1.1) that minimizes the

makespan (and other terms that are application specific), and a set of constraints (Equations 1.2-1.9). Equation 1.2 ensures that each task is assigned to one agent. Equation 1.3 ensures that the temporal constraints are satisfied. Equations 1.4 and 1.5 ensure that agents are not required to complete tasks faster or slower than they are capable. Equations 1.6 and 1.7 sequence actions to ensure that agents performing tasks maintain safe buffer distances from one another. Equations 1.8 and 1.9 ensure that each agent only performs one task at a time. Note Equations 1.6 and 1.7 couple the variables relating sequencing constraints, spatial locations, and task start and end times, resulting in tight dependencies among agents' schedules.

While the task allocation and scheduling problem may be readily formulated as a MILP, the complexity of this approach is exponential and leads to computational intractability for problems of interest in large-scale factory operations [5]. The key bottleneck is evaluating the binary decision variables $x_{i,j}$ for the sequencing of tasks, which grows exponentially with the square of the number of tasks (i.e., $2^{|\boldsymbol{\tau}|^2}$). To achieve good scalability characteristics various decentralized or distributed approaches have been proposed [6, 8, 13, 42, 48]. Fast computation is desirable because it provides the capability for on-the-fly replanning in response to schedule disturbances [3, 8, 45]. These works boost computational performance by decomposing plan constraints and contributions to the objective function among agents [6]. However, these methods break down when agents' schedules become tightly intercoupled, as they do when multiple agents are maneuvering in close physical proximity. While distributed approaches to coordination are necessary for field operations where environment and geography affect the communication among agents, factory operations allow for sufficient connectivity and bandwidth for either centralized or distributed approaches to task assignment and scheduling.

The primary goal of the work I present in this thesis is to alleviate the key sequencing bottleneck and provide the capability for fast re-computation of schedules in response to dynamic disturbances under tightly intercoupled temporospatial constraints. The core technical innovation of my work is the development of polynomial-time, near-optimal techniques for sequencing and testing the schedulability of task

sets [19, 20]. My work is inspired by techniques in real-time systems scheduling for scheduling and testing the schedulability of task sets. Methods in real-time systems analysis often sacrifice completeness but gain computational tractability by leveraging problem structure. To model the manufacturing environment as a real-time processor scheduling scenario, I construct a real-time systems scheduling analogy.

1.2 Processor Scheduling Analogy

I represent many real-world manufacturing constraints through real-time processor scheduling models. Specifically, I base the analogy on the self-suspending task model, as shown in Equation 1.10.

$$\tau_i : (\phi_i, (C_i^1, E_i^1, C_i^2, E_i^2, \ldots, E_i^{m_{i-1}}, C_i^{m_i}), T_i, D_i, \boldsymbol{A_i}, \boldsymbol{R_i}) \tag{1.10}$$

In this model, there is a task set, $\boldsymbol{\tau}$, comprised of tasks τ_i. In each task τ_i, there are m_i subtasks τ_i^j 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. The assignment of processors to subtasks is described by A_i, where processor A_i^j is assigned to execute subtask τ_i^j. The shared memory resource constraints for τ_i are described by $\boldsymbol{R_i}$, where $\boldsymbol{R_i^j}$ is the set of shared memory resources required to execute τ_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.

For the manufacturing environment, I develop the following relationships. I model human and robot workers as processor cores on a computer. In the manufacturing environment, there are often sets of tasks related through wait or precedence constraints (e.g., applying several coats of paint, where each task is the application of one coat) and some tasks that are not explicitly ordered (e.g., painting different parts of the fuselage). The self-suspending task model is hierarchically composed of tasks

and subtasks. Subtasks τ_i^j and τ_i^{j+1} are ordered such that τ_i^j must precede τ_i^{j+1} by duration E_i^j. However, there is no explicit precedence constraint between subtasks τ_i^j and τ_x^y for $i \neq x$. Therefore, I translate tasks from the manufacturing environment to the hierarchically composed tasks in the processor scheduling framework.

I model wait or precedence constraints as self-suspensions in the processor scheduling scenario. In manufacturing, tasks may have precedence relations (e.g., assemble one structure before a second) or wait constraints (e.g., the first coat of paint must dry before applying the second). In real-time systems, tasks may self-suspend, meaning the processor must wait a certain duration between the execution of two subtasks. For example, 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. As such, I model wait constraints as self-suspensions and mere precedence constraints as self-suspensions of zero duration.

Manufacturing processes are governed by upperbound temporal relationships, or deadlines. For example, on a "pulse" line, tasks must be completed once every pulse, or else workers at the next assembly location will not be able to progress. The requirement to complete work once every pulse can be expressed as a task deadline. We also must consider spatial constraints. Robot and human workers occupy a specific physical space or location to complete a task when constructing an assembly. While one agent is working, another human or robot cannot enter that space. In processor scheduling, some tasks require access to space in memory to manipulate information. Similarly, multiple processors cannot access the same space in memory without adverse interaction effects. As such, I model spatial locations as shared memory resources.

Augmented Processor Scheduling Task Model

In Chapters 2 and 4, I 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. I call these deadline constraints *intra-task* and *subtask* deadlines. I define an intra-task deadline as shown in

Equation 1.11.

$$D^{rel}_{(i,a),(i,b)} : \left(f^a_i - s^b_i \leq d^{rel}_{(i,a),(i,b)}\right) \tag{1.11}$$

where f^b_i is the finish time of subtask τ^b_i, s^j_i is the start time of subtask τ^j_i, and $d^{rel}_{(i,a),(i,b)}$ is the upperbound temporal constraint between the start and finish times of these two subtasks, such that $b > a$. In addition to intra-task deadlines $\boldsymbol{D^{rel}_i}$ for τ_i, I extend our task model to include subtasks deadlines, where $\boldsymbol{D^{abs}_i}$ is the set of subtask deadlines for subtasks in τ_i. As shown in Equation 1.12, if a subtask τ^j_i is constrained by a subtask deadline constraint, then f^j_i must not exceed $d^{abs}_{i,j}$.

$$D^{abs}_{i,j} : \left(f^j_i \leq d^{abs}_{i,j}\right) \tag{1.12}$$

In our processor scheduling analogy I use intra-task deadlines to represent end-to-end deadlines in manufacturing. For example, only a certain amount of time can pass between the start and finish of applying composite material before it cures. Lastly, I utilize subtask deadlines to upperbound the amount of time that passes since the beginning of the schedule until a certain subtask is complete. For example, if a quality assurance agent needs to inspect a component of the work at a certain time, I apply a subtask deadline to the set of subtasks that need to be completed before the inspection.

1.3 Thesis Contributions

1.3.1 Scheduling and Analysis of Real-Time Systems

Based on this analogy, I present work on scheduling and testing the schedulability of these self-suspending task sets. I begin in Chapter 2 where I present three contributions. First, I provide a 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 [33]. Similar to prior work, I test the schedulability of these task sets by providing an upperbound for the amount of self-suspension time that

needs to be treated as task cost [34, 35, 36, 44]. The test I develop is polynomial in time and, in contrast to prior art, generalizes to non-preemptive task sets with more than one self-suspension per task. Second, I extend our schedulability test to also handle task sets with intra-task deadlines. Third, I introduce a new scheduling policy to accompany the schedulability test. I specifically designed this scheduling policy to restrict the behavior of a self-suspending task set so as to provide an analytical basis for an informative schedulability test.

If the schedulability test I develop in Chapter 2 finds a task set feasible, then we need a method for the online scheduling of the task set. In Chapter 3, I present a near-optimal method for scheduling tasks sets that are returned as feasible from my uniprocessor schedulability test. The main contribution of this work is a polynomial-time, online consistency test, which determines whether we can schedule subtask τ_i^j at time t given the upperbound temporal constraints in the task set. The online consistency test is called the Russian Dolls Test; the name comes from how the test works by determining whether we can "nest" a set of subtasks within the slack of the deadline of another set of subtasks. My scheduling algorithm is not optimal; 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-complete [17, 18]. However, I show through empirical evaluation that schedules resulting from my algorithm are within a few percent of the optimal makespan.

In Chapter 4, I extend both the uniprocessor scheduling algorithm and task model to the multiprocessor case. The multiprocessor task model includes processor-subtask assignments, shared memory resources, intra-task deadlines (i.e., Equation 1.11), and subtask deadlines (i.e., Equation 1.12). The scheduling algorithm utilizes a polynomial-time, online consistency test, which I call the Multiprocessor Russian Dolls Test, to ensure temporal consistency due to the temporal and shared memory resource constraints of the task set.

1.3.2 Tercio: a Task Allocation and Scheduling Algorithm

Based on the techniques I develop in the scheduling of these self-suspending task sets in Chapters 2-4, I designed a multi-agent task allocation and scheduling system, called Tercio[1][22]. The algorithm is made efficient by decomposing task allocation from scheduling and utilizes the techniques I present in Chapter 4 to perform multi-agent sequencing. Results show that the method is able to generate near-optimal task assignments and schedules for up to 10 agents and 500 tasks in less than 20 seconds on average. In this regard, Tercio scales better than previous approaches to hybrid task assignment and scheduling [9, 10, 25, 26, 27, 49]. Although the sequencing algorithm is satisficing, I show that it is *tight*, meaning it produces near-optimal task sequences for real-world, structured problems. An additional feature of Tercio is that it returns flexible time windows for execution [38, 52], which enable the agents to adapt to small disturbances online without a full re-computation of the schedule. I present this work in Chapter 5.

1.3.3 Human-Centered Integration of Centralized Scheduling Algorithms

While fast task assignment and scheduling is an important step to enabling the integration of robots into the manufacturing environment, we also need to consider a human-centered approach when implementing Tercio in the factory. Successful integration of robot systems into human teams requires more than tasking algorithms that are capable of adapting online to the dynamic environment. The mechanisms for coordination must be valued and appreciated by the human workers. Human workers often find identity and security in their roles or jobs in the factory and are used to some autonomy in decision-making. A human worker that is instead tasked by an automated scheduling algorithm may begin to feel that he or she is diminished. Even if the algorithm increases process efficiency at first, there is concern that taking control away from the human workers may alienate them and ultimately damage the

[1] Joint work with Ronald Wilcox

productivity of the human-robot team. The study of human factors can inform the design of effective algorithms for collaborative tasking of humans and robots.

In Chapter 6, I describe a pilot study[2] conducted to gain insight into how to integrate multi-agent task allocation and scheduling algorithms to improve the efficiency of coordinated human and robotic work[21]. In one experimental condition, both the human and robot are tasked by Tercio, the automatic scheduling algorithm. In the second condition, the human worker is provided with a limited set of task allocations from which he/she can choose. I hypothesize that giving the human more control over the decision-making process will increase worker satisfaction, but that doing so will decrease system efficiency in terms of time to complete the task. Analysis of the experimental data ($n = 8$) shows that when workers were given freedom to choose, process efficiency decreased significantly. However, user-satisfaction seems to be confounded by whether or not the subject chose the optimal task allocation. Four subjects were allowed to choose their task allocation. Within that pool, the one subject that chose the optimal allocation rated his/her satisfaction the highest of all subjects tested, and the mean of the satisfaction rating of the three who chose the suboptimal allocation was lower than those subjects who's roles were assigned autonomously.

[2]Joint work with Ronald Wilcox, Ana Diaz Artiles, and Fei Yu

Chapter 2

Uniprocessor Schedulability Test for Hard, Non-Preemptive, Self-Suspending Task Sets with Multiple Self-Suspensions per Task

2.1 Introduction

In this chapter, we present three contributions. First, we provide a 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 [33]. Similar to prior work, we test the schedulability of these task sets by providing an upperbound for the amount of self-suspension time that needs to be treated as task cost [34, 35, 36, 44]. Our test is polynomial in time and, in contrast to prior art, generalizes to non-preemptive task sets with more than one self-suspension per task.

Second, we extend our schedulability test to also handle task sets with deadlines constraining the upperbound temporal difference between the start and finish of two subtasks within the same task. Third, we introduce a new scheduling policy to accompany the schedulability test. We specifically designed this scheduling policy to

restrict the behavior of a self-suspending task set so as to provide an analytical basis for an informative schedulability test.

We begin in Section 2.2 with a brief review or prior work. In Section 2.3, we introduce our augmented self-suspending task model. Next, we introduce new terminology to help describe our schedulability test and the execution behavior of self-suspending tasks in Section 2.4. We then motivate our new scheduling policy, which restricts the behavior of the scheduler to reduce scheduling anomalies 2.5. In Section 2.6, we present our schedulability test, with proof of correctness. Finally, in Section 2.7, we empirically validate that the test is tight, meaning that it does not significantly overestimate the temporal resources needed to execute the task set.

2.2 Background

Increasingly in real-time systems, computer processors must handle the self-suspension of tasks and determine the feasibility of these task sets. Self-suspensions 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 [33]. Thus, a 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.

Unfortunately the problem is NP-Hard, as can be shown through an analysis of the interaction of self-suspensions and task deadlines [24, 38]. In practice, the relaxation of a deadline in a self-suspending task set may result in temporal infeasibility. Many uniprocessor, priority-based scheduling algorithms introduce scheduling anomalies since they do not account for this interaction [32, 44]. The most straightforward, correct approach 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 costs [36, 37].

A number of different approaches have been proposed to test the schedulability of self-suspending task sets. The dominant strategy is to upperbound the duration of self-suspensions that needs to be treated as task cost [34, 36]. Recently, Liu and Anderson *et al.* have demonstrated significant improvements over prior art in testing preemptive task sets with multiple self-suspensions per task, under Global Earliest Deadline First (GEDF) on multiple processor systems [36]. Previously, Devi proposed a test to compute the maximum utilization factor for tasks with single self-suspensions scheduled under Earliest Deadline First (EDF) for uniprocessor systems. The test works by analyzing priorities to determine the number of tasks that may be executed during a self-suspension [15]. Other approaches test schedulability by analyzing the worst case response time of tasks due to external blocking events [24, 29, 50].

The design of scheduling policies for self-suspending task sets also remains a challenge. While EDF has desirable properties for many real-time uniprocessor scheduling problems, certain anomalies arise when scheduling task sets with both self-suspensions and hard deadlines. Ridouard *et al.* note an example where it is possible to schedule a task set under EDF with tight deadlines, while the same task set with looser deadlines fails [44]. Lakshmanan *et al.* report that finding an anomaly-free scheduling priority for self-suspending task sets remains an open problem [32].

While not anomaly-free, various priority-based scheduling policies have been shown to improve the online execution behavior in practice. For example, Rajkumar presents an algorithm called *Period Enforcer* that forces tasks to behave as ideal, periodic tasks to improve scheduling performance and avoid detrimental scheduling anomalies associated with scheduling unrestricted, self-suspending task sets [41]. Similarly, Sun *et al.* presents a set of synchronization protocols and a complementary schedulability test to determine the feasibility of a task set for a scheduler operating under the protocols [47]. Lakshmanan builds on these approaches to develop a *static slack enforcement algorithm* that delays the release times of subtasks to improve the schedulability of task sets [33].

Similar to the approach of Lakshmanan *et al.*, we develop a priority-based schedul-

ing algorithm that reduces anomalies in practice. This policy enables us analytically upperbound the duration of the self-suspensions that needs to be treated as task cost, similar to the approach by Liu *et al.*. To our knowledge, our schedulability test is the first that determines the feasibility of hard, non-preemptive, self-suspending task sets with multiple self-suspensions for each task.

2.3 Our Augmented Task Model

The basic model for self-suspending task sets is shown in Equation 2.1.

$$\tau_i : (\phi_i, (C_i^1, E_i^1, C_i^2, E_i^2, \ldots, E_i^{m_i-1}, C_i^{m_i}), T_i, D_i, \boldsymbol{D_i^{rel}}) \tag{2.1}$$

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.

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 *intra-task* deadlines. We define an intra-task deadline as shown in Equation 2.2.

$$D^{rel}_{(i,a),(i,b)} : \left(f_i^a - s_i^b \leq d^{rel}_{(i,a),(i,b)}\right) \tag{2.2}$$

where f_i^b is the finish time of subtask τ_i^b, s_i^j is the start time of subtask τ_i^j, and $d^{rel}_{(i,a),(i,b)}$ is the upperbound temporal constraint between the start and finish times of these two subtasks, such that $b > a$. $\boldsymbol{D_i^{rel}}$ is the set of intra-task deadlines for τ_i, and $\boldsymbol{D^{rel}}$ is the set of intra-task deadlines for $\boldsymbol{\tau}$. These types of constraints are commonly

included in AI and operations research scheduling models [5, 14, 38, 52].

2.4 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 $D^{rel}_{(i,a)(i,b)}$ 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 intra-task deadlines. On the other hand, the scheduler must take extra consideration to satisfy intra-task 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 2.6, 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 $\boldsymbol{E_{embedded}}$.

2.5 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 $j^{th}+1$* 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 an intra-task deadline constraint.*

Enforcing that all j^{th} subtasks are completed before any $j^{th}+1$ free subtasks allows the processor to execute any embedded k^{th} subtasks where $k > j$ as necessary to ensure that intra-task deadlines are satisfied. The JSF priority scheduling policy offers choice among consistency checking algorithms. A simple algorithm to ensure deadlines are satisfied would require that, if 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 would then be scheduled as early as possible without the processor executing any other subtasks during this duration. Other consistency-check algorithms exist that utilize processor time more efficiently and operate on this structured task model [19, 31, 51].

2.6 Schedulability Test

To describe how our test works and prove its correctness, we will start with a simplified version of the task set and build to the full task model. We follow the following six steps:

1. We restrict $\boldsymbol{\tau}$ such that each task only has two subtasks (i.e., $m_i = 2, \forall i$), there are no intra-task deadlines, and all tasks are released at $t = 0$ (i.e., $\phi = 0, \forall i$). Here we will introduce our formula for upperbounding the amount of self-suspension time that we treat as task cost, W_{free}. 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.

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_ϕ.

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 intra-task deadlines. In this step, we will describe how we calculate an upperbound on processor idle time due to embedded self-suspensions $W_{embedded}$.

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 intra-task 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_i^1 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 all subtasks that *might* be interleaved during a self-suspension, E_i^1, is B_i^1. As described by Equation 2.3, B_i^j is the set of all of the j^{th} and $j^{th} + 1$ subtask costs less the subtasks costs for τ_i^j and τ_i^{j+1}. Note, by definition, τ_i^j and τ_i^{j+1} cannot execute during E_i^j. We further define an operator $B_i^j(k)$ that provides the k^{th} *smallest* subtask cost from B_i^j. We also restrict B_i^j such that the j^{th} and $j^{th} + 1$ 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_i^1 during this step. In Step 4 (Section 2.6), we will explain why we make this restriction on the subtasks in B_i^j.

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 $i = \{1, 2, \ldots, n\}$.

$$B_i^j = \{C_x^y | x \in N \backslash i, y \in \{j, j+1\}, \\ \tau_x^j \in \boldsymbol{\tau_{free}}, \tau_x^{j+1} \in \boldsymbol{\tau_{free}}\} \tag{2.3}$$

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 2.6 and Equation 2.5 where W_i^j is an upperbound on processor idle time due to E_i^j and W^j is an upperbound on processor idle time due to the set of E_i^j self-suspensions. To determine W^j, we first consider the difference between each of the E_i^j self-suspensions and the minimum subtask cost that we can guarantee will execute during E_i^j *iff* E_i^j results in processor idle time. To compute this quantity we provide a minimum bound on the number of free subtasks (Equation 2.4) that will execute during a self-suspension E_i^j. By taking the maximum over all i of W_i^j, we upperbound the idle time due to the set of j^{th} self-suspensions.

$$\eta_i^j = \frac{|B_i^j|}{2} - 1 \tag{2.4}$$

$$W_i^j = \max\left(\left(E_i^j - \sum_{k=1}^{\eta_i^j} B_i^j(k)\right), 0\right) \tag{2.5}$$

$$W^j = \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^j\right) \tag{2.6}$$

To prove that our method is correct, we first show that Equation 2.4 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 will prove this by contradiction, assuming that E_i^1 is the dominant contributor to idle time and fewer than $\frac{|B_i^1|}{2} - 1$ subtasks execute (i.e., are completely interleaved) during E_i^1. 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 $\frac{|B_i^1|}{2} - 1$ subtasks execute during E_i^1, then Equation 2.5 correctly upperbounds idle time due to E_i^1. Lastly, we will show that if an E_i^1 is the dominant contributor to idle time then Equation 2.6 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* 2.4, where $j = 1$.

Proof by Contradiction for $i = 1$. We currently assume that all subtasks are free (i.e., there are no intra-task deadline constraints), thus $\frac{|B_i^1|}{2} = n$. We recall that a processor executing under JSF will execute all j^{th} subtasks before any free $j^{th}+1$ 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 2.4 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 $\boldsymbol{\tau}$).*

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. 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. □

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 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 $i = n$. □

Proof of Correctness for Equation 2.5, where $j = 1$.

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 2.4). 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. □

Proof of Correctness for Equation 2.6, where $j = 1$.

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 2.4 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 2.5 bounds processor idle time caused by the dominant self-suspension, say E_q^j. Thus, we note in Equation 2.6 that the maximum of the upperbound processor idle time due 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. □

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. $i = \{1, 2, \ldots, n\}$). 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 2.7.

The quantity W_ϕ computed in Step 2 is summed with W^1 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 $t = [0, f_n^1]$.

$$W_\phi = \max_i \phi_i \tag{2.7}$$

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 $j^{th} + 1$ subtask costs during the j^{th} self-suspensions. Again, we recall that our formulation for upperbounding idle time due to the 1^{st} 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 used this understanding of Equation 2.6 to upperbound idle time resulting from phase offsets. We said that we needed to determine 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 2.6 to determine an upperbound on processor idle time during the interval $t = [\max_i(f_i^j), \max_i(f_i^{j+1})]$. 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 2.8.

$$\begin{aligned} W_{free} &= \sum_{j} W^{j} \\ &= \sum_{j} \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(W_i^j\right) \\ &= \sum_{j} \max_{i|E_i^j \in \boldsymbol{E_{free}}} \left(\max\left(\left(E_i^j - \sum_{k=1}^{n-1} B_i^j(k)\right), 0\right)\right) \end{aligned} \tag{2.8}$$

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 $m_i^{th} - 1$ 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 2.3. We note that, by inspection, if τ_x^j were to execute during E_i^j, it would only reduce processor idle time.

Step 4) Intra-task 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 intra-task deadline constraints. Now, we relax this constraint and calculate an upperbound on processor idle time due to embedded self-suspensions $W_{embedded}$.

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 intra-task 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 2.9 and 2.10. Equation 2.9 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 $j^{th}+1$ 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 intra-task deadlines)

Third, we now must consider cases where $\eta_i^j < n - 1$, as described in (Equation 2.4). We recall that $\eta_i^j = n - 1$ if there are no intra-task deadlines; however, with the introduction of these deadline constraints, we can only guarantee that at least $\frac{|B_i^j|}{2} - 1$ subtasks will execute during a given E_i^j, if E_i^j results in processor idle time.

$$W_{embedded} = \sum_{i=1}^{n} \left(\sum_{j=1}^{m_i - 1} E_i^j \left(1 - x_i^{j+1}\right) \right) \tag{2.9}$$

$$x_i^j = \begin{cases} 1, & \text{if } \tau_i^j \in \boldsymbol{\tau_{free}} \\ 0, & \text{if } \tau_i^j \in \boldsymbol{\tau_{embedded}} \end{cases} \tag{2.10}$$

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_{UB}^{τ} the processor will take to complete all instances of each task in the hyperperiod (Equation 2.11). H denotes the hyperperiod of the task set, and H_{LB}^{τ} is defined as the sum over all task costs released during the hyperperiod. Recall that we are still assuming that $T_i = D_i = T_j = D_j, \forall i, j \in N$; thus, there is only one instance of each task in the hyperperiod.

$$H_{UB}^{\tau} = H_{LB}^{\tau} + W_{phase} + W_{free} + W_{embedded} \tag{2.11}$$

$$H_{LB}^{\tau} = \sum_{i=1}^{n} \frac{H}{T_i} \sum_{j=1}^{m_i} C_i^j \tag{2.12}$$

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 2.11, we calculated an upperbound on the time the processor needs to execute the task set, H_{UB}^{τ}. 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 $\boldsymbol{\tau}|_{\boldsymbol{j}} \subset \boldsymbol{\tau}$, that might execute before the task deadline for τ_i. If we find that $H_{UB}^{\tau|_j} \leq D^{abs}$, 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 $D_{i,1}^{abs} = D_i + \phi_i$. 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 $D_{i,k}^{abs} = D_i + T_i(k-1) + \phi_i$.

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 2-1. This algorithm requires as input a task set τ, an absolute deadline D^{abs} for task deadline D_i, and the j subtask index of the last subtask τ_i^j 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 $\boldsymbol{\tau}|_{\boldsymbol{j}}$, the set of subtasks that may execute before D_i. In the absence of intra-deadline constraints, $\boldsymbol{\tau}|_{\boldsymbol{j}}$ includes all subtasks $\tau_i^{j'}$ where $i = N$ (recall $N = \{i | i \in \{1, 2, \ldots, n\}\}$) and $j' \in \{1, 2, \ldots, j\}$. In the case

an intra-task deadline spans subtask τ_x^j (in other words, a deadline $D^{rel}_{(x,a),(x,b)}$ 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 $D^{rel}_{(x,a),(x,b)}$ are also added to the set $\boldsymbol{\tau}|_{\boldsymbol{j}}$. In Line 15, the algorithm tests the schedulability of $\boldsymbol{\tau}|_{\boldsymbol{j}}$ using Equation 2.11.

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 intra-task constraints.

Lines 5-11 search for additional subtasks that may need to execute before τ_i^j due to intra-task 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 time the processor needs to complete $\tau|_j$ (Line 15). If this duration is less than or equal to the deadline D^{abs} associated with D_i for τ_i, then we can guarantee that the deadline will be satisfied by a processor scheduling under JSF. satisfy the deadline (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.

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 intra-task 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 hyperpe-

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

$\tau|_j \leftarrow$ NULL
for x = 1 to $|\boldsymbol{\tau}|$ **do**
    $z \leftarrow j$
    while TRUE **do**
        if $\tau_x^{z+1} \notin (\boldsymbol{\tau_{free}} \cup \boldsymbol{\tau_{embedded}})$ **then**
            break
        else if $\tau_x^{z+1} \in \boldsymbol{\tau_{free}}$ **then**
            break
        else if $\tau_x^{z+1} \in \boldsymbol{\tau_{embedded}}$ **then**
            $z \leftarrow z + 1$
        end if
    end while
    $\tau_x|_j \leftarrow (\phi_x, (C_x^1, E_x^1, C_x^2, \ldots, C_x^z), D_x, T_x)$
end for
if $H_{UB}^{\tau|_j} \leq D^{abs}$ //Using Eq. 2.11 **then**
    return TRUE
else
    return FALSE
end if

Figure 2-1: Pseudo-code for **testDeadline($\boldsymbol{\tau}$,D_i,j)**, which tests whether a processor scheduling under JSF is guaranteed to satisfy a task deadline, D_i.

riod, 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 2.13, 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^*.

$$\begin{aligned}\tau_i^* :(\phi_i, (&C_{i,1}^1, E_{i,1}^1, \ldots, C_{i,1}^{m_i}, C_{i,2}^1, E_{i,2}^1, \ldots, C_{i,2}^{m_i}, \\ &\ldots, C_{i,k}^1, E_{i,k}^1, \ldots, C_{i,k}^{m_i}), D_i^* = H, T_i^* = H, \boldsymbol{D_i^{rel*}})\end{aligned} \tag{2.13}$$

We aim to devise a test where τ_i^* is schedulable if $H_{UB}^{\boldsymbol{\tau^*}} \leq D_i^*$ 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 next $k + 1^{th}$ 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 2.11. If $H_{UB}^{\boldsymbol{\tau}^*} \leq H$, 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 2-2, 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 intra-task 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 $j^{th} + 1$ 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 j subtask index 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 $>\mathbf{J}$[i] (i.e., the current j = 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 $\mathbf{I}$[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 intra-task deadlines, to τ_i^* (Line 17). Next, we set $\mathbf{J}$[i] and $\mathbf{D}$[i][k] to the j subtask index of the subtask we last added to τ_i^* (Lines 18-19). We will use $\mathbf{D}$[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 $\mathbf{I}$[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 = 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.

Schedulability Test Summary

To determine the schedulability of a task set, $\boldsymbol{\tau}$, we call **constructTaskSuperSet**($\boldsymbol{\tau}$) on $\boldsymbol{\tau}$. If the function call returns null then we cannot guarantee the feasibility of the task set. If the function call successfully returns a task superset, $\boldsymbol{\tau}^*$, then we determine an upperbound, $H_{UB}^{\boldsymbol{\tau}^*}$, on the temporal resources required to execute $\boldsymbol{\tau}^*$ using Equation 2.11. If $H_{UB}^{\boldsymbol{\tau}^*} \leq H$, where H is the hyperperiod of $\boldsymbol{\tau}$, then the task set is schedulable under JSF. Furthermore the processor executes $\boldsymbol{\tau}$ under JSF according to the j subtask indices of $\boldsymbol{\tau}^*$.

2.7 Results and Discussion

In this section, we empirically evaluate the tightness of our schedulability test and analyze its computational complexity. We perform our empirical analysis using randomly generated task sets. The number of subtasks m_i for a task τ_i is generated according to $m_i \sim U(1, 2n)$, where n is the number of tasks. If $m_i = 1$, 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 $D_i \sim U(\sum_{i,j} C_i^j, T_i)$.

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

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** $H_{LB} \geq T_i * \mathbf{I}[i] + \phi_i$ **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 2-2: Pseudo-code for **constructTaskSuperSet**($\boldsymbol{\tau}$), which constructs a task superset, $\boldsymbol{\tau}^*$ for $\boldsymbol{\tau}$.

self-suspensions per task. Recently, Abdeddaïm and Masson introduced an approach for scheduling self-suspending task sets using model checking with Computational Tree Logic (CTL) [2]. However, their algorithm is exponential in the number of tasks and does not currently scale to moderately-sized task sets of interest for real-world applications.

2.7.1 Tightness of the Test

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 2.14. This provides a comparison between our method and the naive worst-case analysis that treats all self-suspensions as idle time. We evaluate this metric as a function of task cost and the percentage of subtasks that are constrained by intra-task deadline constraints. We note that these parameters are calculated for $\boldsymbol{\tau}^*$ using **constructTaskSuperSet**($\boldsymbol{\tau}$) and randomly generated task sets $\boldsymbol{\tau}$.

$$\hat{E} = \frac{W_{free} + W_{embedded}}{\sum_{i,j} E_i^j} * 100 \tag{2.14}$$

Figure 2-3 presents the empirical results evaluating the tightness of our schedulability test for randomly generated task sets with 2 to 50 tasks. $\hat{D}$ denotes the ratio of subtasks that are released during the hyperperiod and constrained by intra-task deadline constraints to the total number of subtasks released during the hyperperiod. Fifty task sets were randomly generated for each data point. We see that for small or highly constrained task sets, the amount of self-suspension time treated as task cost is relatively high ($> 50\%$). However, for problems with relatively fewer intra-task deadline constraints, our schedulability test for the JSF priority scheduling policy produces a near-zero upperbound on processor idle time due to self-suspensions.

2.7.2 Computational Scalability

Our 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

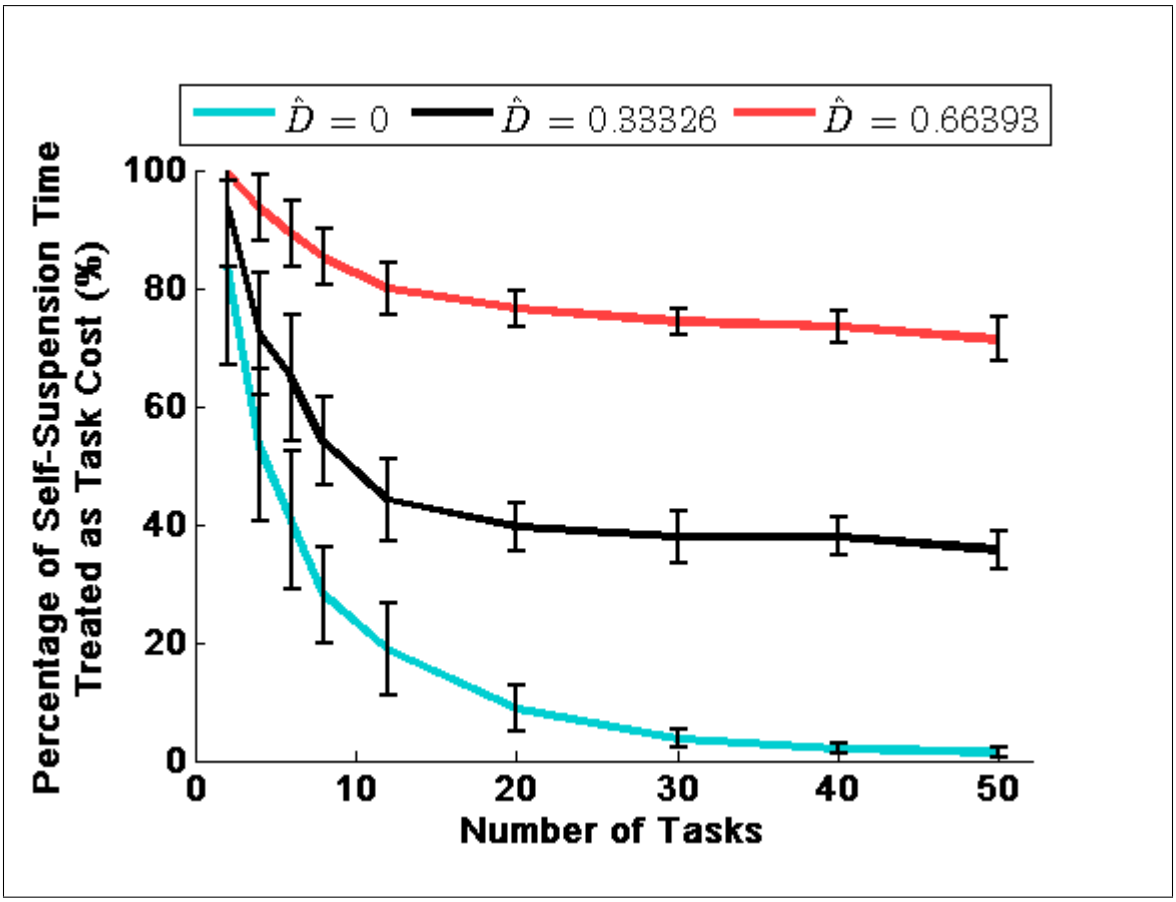

Figure 2-3: The amount of self-suspension time our schedulability test treats as task cost as as a percentage of the total self-suspension time. Each data point and errors bar represents the mean and standard deviation evaluated for fifty randomly generated task sets.

T_{min} is the shortest period of any task in $\boldsymbol{\tau}$. The complexity of evaluating Equation 2.11 for $\boldsymbol{\tau}^*$ is upperbounded by $O\left(n^2 m_{max}\frac{H}{T_{min}}\right)$ where $O\left(nm_{max}\frac{H}{T_{min}}\right)$ bounds the number of self-suspensions in $\boldsymbol{\tau}^*$. The complexity of **testDeadline**() is dominated by evaluating Equation 2.11. 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 2-1 and 2-2, 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 2.11.

2.8 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 deadlines constraining 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 jth 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 our test, empirically evaluate the tightness of our upperbound on processor idle time, and analyze the computational complexity of our method.

Chapter 3

Uniprocessor Scheduling Policy for j^{th} Subtask First

3.1 Introduction

In Chapter 2, we introduce a novel scheduling priority, which we call j^{th} Subtask First (JSF), that we use to develop the first schedulability test for hard, non-preemptive, periodic, self-suspending task set[20]. While JSF provides a framework for scheduling these task sets, it does not fully specify the orderings of tasks most notably when there are deadlines constraining the start and finish times of two subtasks.

In this chapter, we present a near-optimal method for scheduling under the JSF scheduling priority. The main contribution of this work is a polynomial-time, online consistency test, which we call the Russian Dolls Test. The name comes from determining whether we can "nest" a set of tasks within the slack of another set of tasks. Our scheduling algorithm is not optimal; 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-complete [17, 18]. However, we show through empirical evaluation that schedules resulting from our algorithm are within a few percent of the best possible schedule.

We begin in Section 3.2 by introducing new definitions to supplement those from 2.4 that we use to describe our scheduling algorithm and motivate our approach. Next,

we empirically validate the tightness of the schedules produced by our scheduling algorithm relative to a theoretical lowerbound of performance in Section 3.4. We conclude in Section 3.5.

3.2 Terminology

In this section we introduce new terminology to help describe our uniprocessor scheduling algorithm. Specifically, these terms will aid in intuitively explaining the mechanism for our Russian Dolls online temporal consistency test, which we describe in Section 3.3.2. In Chapter 2, we define *free* and *embedded* subtasks (Definitions 1 and 2) as well as *free* and *embedded* self-suspensions (Definitions 3 and 4). We now introduce three new terms that we use in Section 3.3.2 to describe how we schedule self-suspending task sets while ensuring temporal consistency.

Definition 6. *A subtask group, $\boldsymbol{G_i^j}$, is an ordered set of subtasks that share a common deadline constraint. If we have a deadline constraint $D^{rel}_{(i,a),(i,b)}$, then the subtask group for that deadline constraint would be the $\boldsymbol{G_i^j} = \{\tau_i^y | j \leq y \leq b\}$. Furthermore, $\boldsymbol{G_i^j}(k)$ returns the k^{th} element of $\boldsymbol{G_i^j}$, where the set is ordered by subtask index (e.g., y associated with τ_i^y).*

Definition 7. *An active intra-task deadline is an intra-task deadline constraint, $D^{rel}_{(i,a),(i,b)}$, where the processor has at some time t started τ_i^a (or completed) but has not finished τ_i^b. Formally $\boldsymbol{D^{rel}_{active}} = \{D^{rel}_{(i,a),(i,b)} | D^{rel}_{(i,a),(i,b)} \in \boldsymbol{D^{rel}}, s_i^a \leq t < f_i^b\}$, where $\boldsymbol{D}$ is the set of all intra-task deadlines.*

Definition 8. *The set of active subtasks, $\boldsymbol{\tau_{active}}$, are the set of all unfinished subtasks associated with active deadlines. Formally $\boldsymbol{\tau_{active}} = \{\tau_i^j | \tau_i^j \in \boldsymbol{\tau}, \exists D^{rel}_{(i,a),(i,b)}, a \leq j \leq b, s_i^a \leq s_i^j \leq t < f_i^b\}$.*

Definition 9. *The set of next subtasks, $\boldsymbol{\tau_{next}}$, are the set of all subtasks, τ_i^j, such that the processor has finished τ_i^{j-1} but not started τ_i^j. Formally, $\boldsymbol{\tau_{next}} = \{\tau_i^j | \tau_i^j \in \boldsymbol{\tau}, f_i^{j-1} \leq t \leq f_i^j\}$.*

3.3 JSF Scheduling Algorithm

To fully describe our JSF scheduling algorithm, we will first give an overview of the full algorithm. Second, we describe a subroutine that tightens deadlines to produce better problem structure. The key property of this tightened form is that a solution to this reformulated problem also is guaranteed to satisfy the constraints of the original problem. Third, we describe how we use this problem structure to to formulate an online consistency test, which we call the Russian Dolls Test.

3.3.1 JSF Scheduling Algorithm: Overview

Our JSF scheduling algorithm (Figure 3.3.1) receives as input a self-suspending task set, $\boldsymbol{\tau}$, according to Equation 2.1, and terminates after all completing all instances of each task $\tau_i \in \boldsymbol{\tau}$ have been completed. Because these tasks are periodic, scheduling can continue can continue scheduling tasks; however, for simplicity, the algorithm we present terminates after scheduling through one hyperperiod. The algorithm steps through time scheduling released tasks in $\boldsymbol{\tau}^*$. If the processor is available and there exists a released and unscheduled subtask, τ_i^j, the algorithm schedules τ_i^j at the current time t iff the policy can guarantee that doing so does not result in violating another temporal constraint. Now, we step through the mechanics of the algorithm.

In Line 1, we construct our task superset from $\boldsymbol{\tau}$ using **constructTaskSuperSet**($\boldsymbol{\tau}$) we describe in Chapter 2 Section 2.6. We recall that $\boldsymbol{\tau}$ is a hard, periodic, self-suspending task set with phase offsets, task deadlines less than or equal to periods, intra-task deadlines, and multiple self-suspensions per task. $\boldsymbol{\tau}^*$ is a task set, composed of each task instance of $\boldsymbol{\tau}$ released during the hyperperiod for $\boldsymbol{\tau}$. The tasks in $\boldsymbol{\tau}^*$ are restricted such that $T_i^* = T_j^* = H$ where H is the hyperperiod for $\boldsymbol{\tau}$, and T_i^* and T_j^* are periods of tasks τ_i^*, τ_j^* in $\boldsymbol{\tau}^*$. Most importantly, we know that if τ is found schedulable according to our schedulability test (Lines 2-4), then our JSF scheduling algorithm will be able to satisfy all task deadlines. Thus, our scheduling algorithm merely needs to satisfy intra-task deadlines by allowing or disallowing the interleaving of certain subtasks and self-suspensions.

```
JSFSchedulingAlgorithm(τ)
 1: τ* ← constructTaskSuperSet(τ)
 2: if τ* = ∅ then
 3:     return FALSE
 4: end if
 5: D^rel* ← simplifyIntraTaskDeadlines(D^rel*)
 6: t ← 0
 7: while TRUE do
 8:     if processor is idle then
 9:         availableSubtasks ← getAvailableSubtasks(t);
10:         for (k = 1 to |availableTasks|)) do
11:             τ_i^j ← availableTasks[k];
12:             if russianDollsTest(τ_i^j) then
13:                 t_s ← t
14:                 t_s ← t_s + C_i^j
15:                 scheduleProcessor(τ_i^j,t_s,t_f)
16:                 break
17:             end if
18:         end for
19:     end if
20:     if all tasks in τ* have been finished then
21:         return TRUE
22:     else
23:         t ← t + 1
24:     end if
25: end while
```

Figure 3-1: Pseudocode describing our **JSFSchedulingAlgorithm(τ)**

In Line 5, we simplify the intra-task deadlines so that we can increase the problem structure. The operation works by mapping multiple, overlapping intra-task deadlines constraints into one intra-task deadline constraint such that, if a scheduling algorithm satisfies the one intra-task deadline constraint, then the multiple, overlapping constraints will also be satisfied. For example, consider two intra-task deadline constraints, $D^{rel*}_{(i,a),(i,b)}$ and $D^{rel*}_{(i,y),(i,z)}$, such that $a \leq y \leq b$. First, we calculate the tightness of each deadline constraint, as shown in Equation 3.1. Second, we construct our new intra-task deadline, $D^{rel*}_{(i,a),(i,\max(b,z))}$, such that the slack provided by $D^{rel*}_{(i,a),(i,\max(b,z))}$ is equal to the lesser of the slack provided by $D^{rel*}_{(i,a),(i,b)}$ and $D^{rel*}_{(i,y),(i,z)}$, as shown in Equation 3.2. Lastly, we remove $D^{rel*}_{(i,a),(i,b)}$ and $D^{rel*}_{(i,y),(i,z)}$ from the set of intra-task deadline constraints. We continue constructing new intra-task deadline constraints until there are no two deadlines that overlap (i.e., $\neg\exists$ $D^{rel*}_{(i,a),(i,b)}$ and $D^{rel*}_{(i,y),(i,z)}$, such that $a \leq y \leq b$).

$$\delta^{*}_{(i,a),(i,b)} = d^{rel*}_{(i,a),(i,b)} - \left(C^{*b}_{i} + \sum_{j=a}^{b-1} C^{*j}_{i} + E^{*j}_{i} \right) \tag{3.1}$$

$$d^{rel*}_{(i,a),(i,\max(b,z))} = \min\left(\delta^{*}_{(i,a),(i,b)}, \delta^{*}_{(i,y),(i,z)}\right) + C^{*\max(b,z)}_{i} + \sum_{j=a}^{\max(b,z)-1} C^{*j}_{i} + E^{*j}_{i} \tag{3.2}$$

Next, we initialize our time to zero (Line 6) and schedule all tasks in $\boldsymbol{\tau}$ released during the hyperperiod (i.e., all τ^{*}_{i} in $\boldsymbol{\tau^{*}}$) (Lines 7-23). At each step in time, if the processor is not busy executing a subtask (Line 8), we collect all available subtasks (Line 9). There are three conditions necessary for a subtask, τ^{*j}_{i}, to be available. First, an available subtask, τ^{*j}_{i} must have been released (i.e., $t \geq r^{*j}_{i}$). Second, the processor must have neither started nor finished τ^{*j}_{i}. If τ^{*j}_{i} is a free subtask, then all τ^{*j-1}_{i} subtasks must have been completed. This third condition is derived directly from the JSF scheduling policy.

In Lines 10-18, we iterate over all available subtasks. If the next available subtask (Line 11) is temporally consistent according to our online consistency test (Line 12), which we describe in Section 3.3.2, then we schedule the subtask at time t. We

note that we do not enforce a priority between available subtasks. However, one could prioritize the available subtasks according to EDF, RM, or another scheduling priority. For generality in our presentation of the JSF Scheduling Algorithm, we merely prioritize based upon the i index of $\tau_i^* \in \boldsymbol{\tau}^*$. If we are able to schedule a new subtask, we terminate the iteration (Line 16). After either scheduling a new subtask or if there are no temporally consistent, available subtasks, we increment the clock (Line 23). If all tasks (i.e. all subtasks) in $\boldsymbol{\tau}^*$ have been scheduled, then the scheduling operation has completed (Line 21).

3.3.2 The Russian Dolls Test

The Russian Dolls Test is a method for determining whether scheduling a subtask, τ_i^j, at time t, will result in a temporally consistent schedule. Consider two deadlines, $D^{rel}_{(i,j),(i,b)}$ and $D^{rel}_{(x,y),(x,z)}$ such that $D^{rel}_{(i,j),(i,b)} \leq D^{rel}_{(x,y),(x,z)}$, with associated subtask groups G_i^j and G_x^y. Futhermore, the processor has just finished executing τ_x^w, where $y \leq w < z$, and we want to determine whether we can next schedule τ_i^j. To answer this question, the Russian Dolls Test evaluates whether we can nest the amount of time that the processor will be busy executing G_i^j within the slack of $D^{rel}_{(x,y),(x,y+z)}$. If this nesting is possible, then we are able to execute τ_i^j and still guarantee that the remaining subtasks in G_i^j and G_x^y can satisfy their deadlines. Otherwise, we assume that scheduling G_i^j at the current time will result in temporal infeasibility for the remaining subtasks in G_x^y.

To understand how the Russian Dolls Test works, we must know three pieces of information about τ_i^j, and $\boldsymbol{\tau_{active}}$. We recall an in intra-task deadline, $D^{rel}_{(i,a),(i,b)}$, is active if the processor has started τ_i^a and has not finished τ_i^b. In turn, a subtask is in $\boldsymbol{\tau_{active}}$ if it is associated with an active deadline.

Definition 10. *$t_{max}|_i^j$ is defined as remaining time available to execute the unexecuted subtasks in G_i^j. We compute $t_{max}|_i^j$ using Equation 3.3.*

$$t_{max}|_i^j = \min \begin{bmatrix} D^{rel}_{(i,a),(i,b)} + s_i^a \\ T_i - \left(\sum_{q=b+1}^{m_i} C_i^q + E_i^{q-1}\right) \end{bmatrix} - t \tag{3.3}$$

Definition 11. *$t_{min}|i^j$ is the a lowerbound on the time the processor will be occupied while executing subtasks in G_i^j. We compute $t_{min}|_i^j$ using Equation 3.4. Because the processor may not be able to interleave a subtask τ_x^y during the self-suspensions between subtasks in G_i^j, we conservatively treat those self-suspensions as task cost in our formulation of $t_{min}|_i^j$. If there is no intra-task deadline associate with τ_i^j, then $t_{min}|_i^j = C_i^j$.*

$$t_{min}|_i^j = C_i^b + \sum_{q=j}^{b-1} C_i^q + E_i^q \tag{3.4}$$

Definition 12. *$t_\delta|_i^j$ is the slack time available for the processor to execute subtasks not in the G_i^j. This duration is equal to the difference between $t_{max}|_i^j$ and $t_{min}|_i^j$.*

$$t_\delta|_i^j = t_{max}|_i^j - t_{min}|_i^j \tag{3.5}$$

Having defined these terms, we can now formally describe the Russian Dolls Test, as described in Definition 13.

Definition 13. *The Russian Dolls Test determines whether we can schedule τ_i^j at time t by evaluating two criteria. First the test checks whether the direct execution of τ_i^j at t will result in a subtask, τ_x^y, missing its deadline during the interval $t = [s_i^j, f_i^j]$ due to some $D^{rel}_{(x,w),(x,z)}$, where $w \leq y \leq z$. Second, if $\exists D^{rel}_{(i,j),(i,b)}$, the test checks whether activating this deadline will result in a subtask missing its deadline during the interval $t = [f_i^j, d^{rel}_{(i,j),(i,b)}]$ due to active intra-task deadlines.*

To check the first consideration, we can merely evaluate whether the cost of τ_i^j (i.e., C_i^j) is less than or equal to the slack of every active deadline. For the second consideration, if there is a deadline $D^{rel}_{(x,w),(x,z)}$ such that $x = i$ and $w = j$, then we must consider the indirect effects of activating $D^{rel}_{(i,j),(i,z)}$ on the processor. If $\{\tau_i^{j+1}, \ldots, \tau_i^b\}$ is the set of all unexecuted tasks in G_i^j after executing τ_i^j, then we must ensure that

the can nest amongst the other active subtasks. If, for all active deadlines $D^{rel}_{(x,w),(x,z)}$*, where* $\tau^y_x \in \boldsymbol{\tau_{next}}$*, we can nest* $\{\tau^{j+1}_i, \ldots, \tau^b_i\}$ *within the slack of* $\{\tau^y_x, \ldots, \tau^z_x\}$ *or vice versa, then we can guarantee that the processor will find a feasible schedule.*

We note if τ^j_i, with associated deadline $D^{rel}_{(i,j),(i,z)}$, passes the Russian Dolls Test, we do not need to re-test $D^{rel}_{(i,j),(i,z)}$ when attempting to execute any subtask in the set $\{\tau^{j+1}_i, \ldots, \tau^z_i\}$. For the processor to execute a subtask in $\{\tau^{j+1}_i, \ldots, \tau^z_i\}$, we merely need to test whether the cost of the subtask is less than or equal to the slack of every other active deadlines not including $D^{rel}_{(i,j),(i,z)}$.

We provide pseudocode to describe the Russian Dolls Test in Figure 3.3.2. In Line 1, we iterate over all subtasks that are *active* and *next*. For a subtask, τ^y_x to be in *active* and *next*, then τ^{y-1}_x must have been completed and there must be an intra-task deadline $D^{rel}_{(x,w),(x,z)}$ such that $w \leq y \leq z$. If the r^{th} subtask (Line 2) in the set of active and next subtasks is not the equal to the τ^j_i, then we proceed with testing the r^{th} subtask in this set (Line 3). In Lines 4-6, we evaluate the first consideration of the Russian Dolls Test: whether the cost of τ^j_i (i.e., C^j_i) is less than or equal to the slack of $D^{rel}_{(x,w),(x,z)}$. If not, then executing τ^j_i at time t will directly result in τ^y_x missing its deadline, so we return that the nesting is not possible (Line 5).

Next, we evaluate the second consideration of the Russian Dolls Test: if there is a deadline $D^{rel}_{(x,w),(x,z)}$ such that $x = i$ and $w = j$, then we must consider what happens after executing τ^j_i the indirect effects of activating $D^{rel}_{(i,j),(i,b)}$ on the processor. If there is such a deadline $D^{rel}_{(i,j),(i,z)}$ (Line 7), then we consider whether we can nest the execution of $\{\tau^{j+1}_i, \ldots, \tau^b_i\}$ within the slack of $D^{rel}_{(x,w),(x,z)}$ or nest the execution of $\{\tau^y_x, \ldots, \tau^z_x\}$ within the slack of $D^{rel}_{(i,j),(i,b)}$ (Line 8). If not, then we cannot guarantee that all subtasks in these sets (i.e., $\{\tau^{j+1}_i, \ldots, \tau^b_i\} \cup \{\tau^y_x, \ldots, \tau^z_x\}$) will satisfy their deadline requirements, so we return false (Line 9). After iterating over all active, next subtasks, and we are able to satisfy both criteria of the Russian Dolls Test, then we may execute τ^j_i at time t.

russianDollsTest(τ_i^j,t)

1: **for** $r = 1$ to $|\boldsymbol{\tau_{active}} \cap \boldsymbol{\tau_{next}}|$ **do**
2: $\quad \tau_x^y \leftarrow \{\boldsymbol{\tau_{active}} \cap \boldsymbol{\tau_{next}}\}(r)$
3: $\quad$ **if** $\tau_x^y \neq t_i^j$ **then**
4: $\quad\quad$ **if** $C_i^j > t_\delta|_x^y$ **then**
5: $\quad\quad\quad$ **return** false
6: $\quad\quad$ **end if**
7: $\quad\quad$ **if** $\exists D^{rel}_{(i,j),(i,b)}$ **then**
8: $\quad\quad\quad$ **if** $\neg\left(\left(t_{max}|_x^y \leq t_\delta|_i^{j+1}\right) \vee \left(t_\delta|_x^y \geq t_{max}|_i^{j+1}\right)\right)$ **then**
9: $\quad\quad\quad\quad$ **return** false
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end if**
12: $\quad$ **end if**
13: **end for**

3.4 Results

In this section, we empirically validate the tightness of the scheduler and analyze its computational complexity. We perform our empirical analysis using randomly generated task sets. The number of subtasks m_i for a task τ_i is generated according to $m_i \sim U(1, 2n)$, where n is the number of tasks. If $m_i = 1$, 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 $D_i \sim U(\sum_{i,j} C_i^j, T_i)$.

3.4.1 Empirical Validation

The metric we use to evaluate the tightness of our JSF Scheduling Algorithm is similar to the metric we used in Chapter 2 to test the tightness of our Schedulability Test. For our Schedulability Test, we consider the percentage of self-suspension time our method treats as task cost. This measure provides a comparison between our schedulability test and the naive worst-case analysis that treats all self-suspensions as idle time. For our JSF Scheduling Algorithm, we now consider the percentage of self-suspension time that the processor is *actually* is idle.

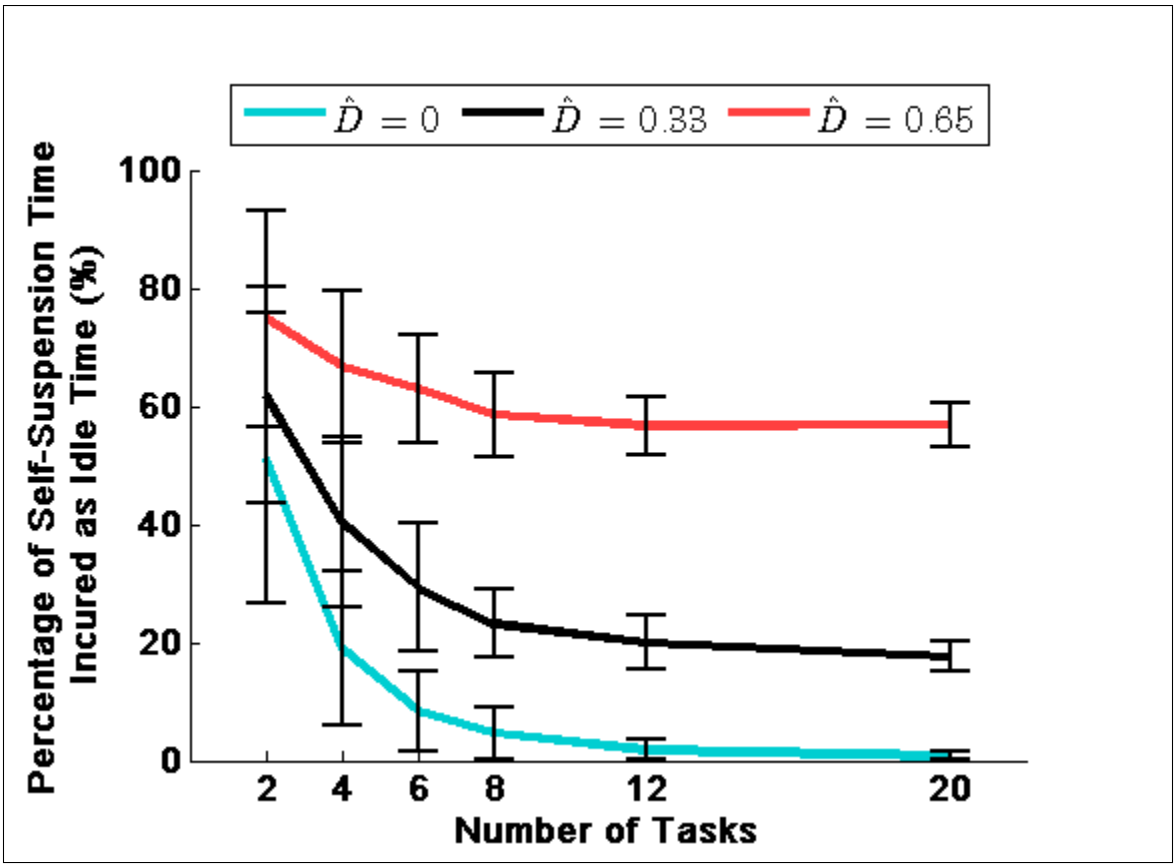

Figure 3-2: Percentage of self-suspension time that the processor is is idle.

Figure 3-2 presents the empirical results of evaluating the tightness of our JSF scheduling algorithm for randomly generated task sets with 2 to 20 tasks, and Figure 3-3 includes the tightness of the schedulability test for comparison. Each data point and error bars represent the mean and standard deviation valuated for ten randomly generated task sets. $\hat{D}$ denotes the ratio of subtasks that are released during the hyperperiod and constrained by intra-task deadline constraints to the total number of subtasks released during the hyper period. We see the amount of idle time due to self-suspensions is inversely proportional to problem size. For large problems, our JSF scheduling algorithm produces a near-zero amount of idle time due to self-suspensions relative to the total duration of all self-suspensions during the hyperperiod.

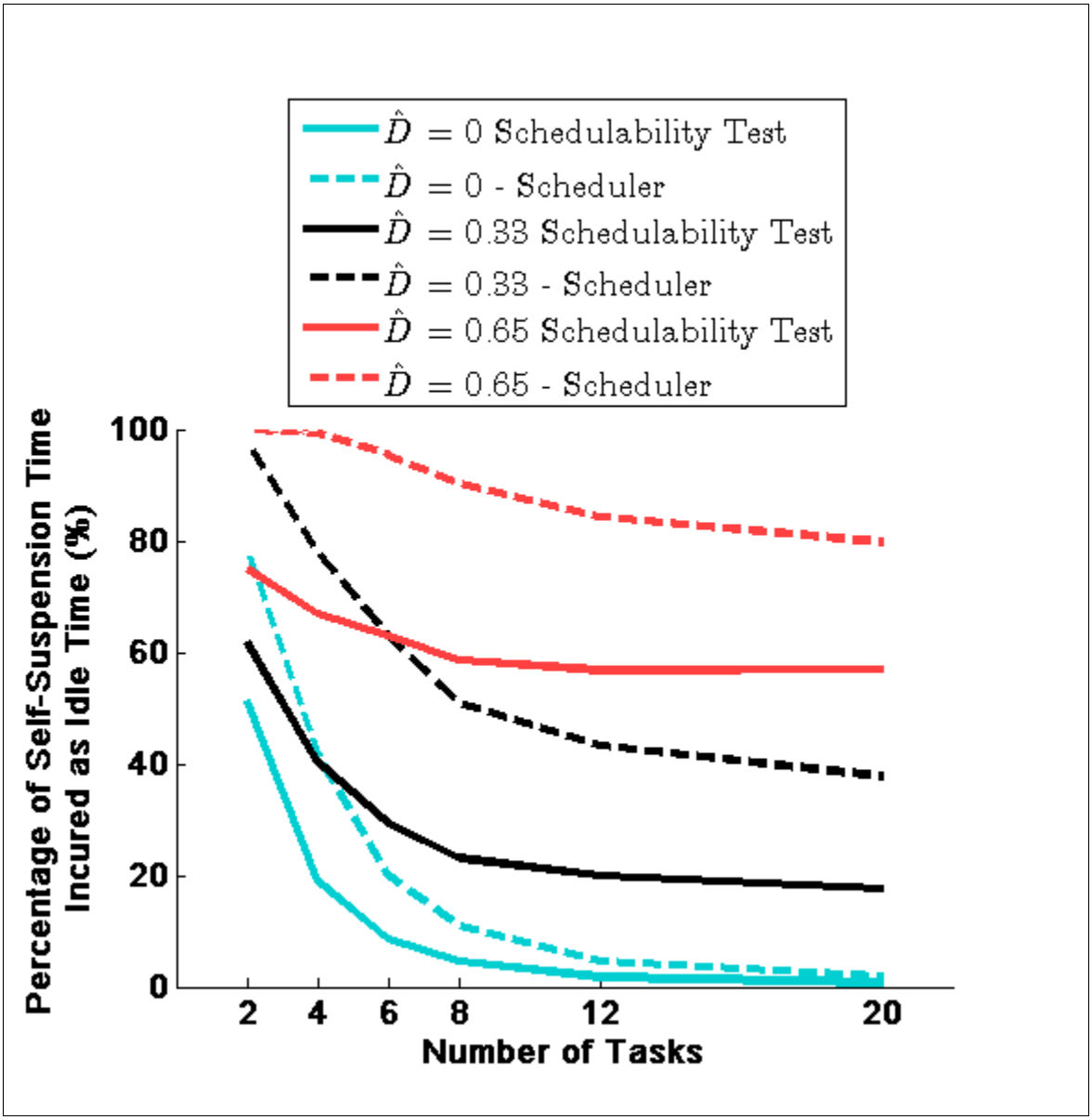

Figure 3-3: Percentage of self-suspension time that the processor is is idle compared to the percentage of self-suspension time the schedulability test assumed as idle time.

3.4.2 Computational Complexity

We upperbound the computational complexity of our JSF Scheduling Algorithm at each time step. At each time step, the processor must consider n tasks in the worst case. For each of the n tasks, the scheduler would call **russianDollsTest**(τ_i^j). In the worst case, the number of active deadlines is upperbounded by n; thus, the complexity of the Russian Dolls Test is $O(n)$. In turn, the JSF Scheduling algorithm performs at most $O(n^2)$ operations for each time step.

3.5 Conclusion

We have presented a uniprocessor scheduling algorithm for hard, non-preemptive, self-suspending task sets with intra-task deadline constraints and . Our polynomial-time scheduling algorithm leverages problem structure by scheduling according to the j^{th} subtask first (JSF) scheduling policy that we developed in Chapter 2. Our algorithm also utilizes an polynomial-time, online consistency test to ensure temporal consistency due to intra-task deadlines, called the Russian Dolls Test. Although the JSF policy and Russian Dolls Test are not optimal, we show through empirical evaluation that our scheduling algorithm produces schedules that are within a few percent of the best possible makespan.

Chapter 4

Multiprocessor Scheduling Policy

4.1 Introduction

In Chapter 3, we present a near-optimal method for uniprocessor scheduling of hard, non-preemptive, self-suspending task sets. These task sets were defined by at least one subtask per task, intra-task deadlines, and phase offsets. In this chapter, we extend our method for scheduling uniprocessor systems to handle multiprocessor systems. We use this extension in Chapter 5 to develop a task allocation and scheduling algorithm for robotic manufacturing of aerospace structures. To schedule these tasks sets, we need to incorporate spatial constraints (i.e., shared memory resources) to ensure that agents do not interfere with each other in space (e.g., collisions). In Section 4.2, we extend the self-suspending task model we present in Chapters 2 and 3 for the multiprocessor case. We next extend some of the terminology we introduced in Chapters 2-3 to the multiprocessor case in Section 4.3 In Section 4.4, we present our scheduling policy. In Section 4.5, we analyze the computational complexity of our algorithm, and we conclude in Section 4.6.

4.2 Our Augmented Task Model

In Chapter 2, we present the self-suspending task model we use to better reflect the constraints of interest (Equation 2.1). For the multiprocessor case, we incorporate

both an assignment of processors to subtasks (i.e., task allocation) and shared memory resources as shown in Equation 4.1. In this model, $R_i = \{R_i^1, R_i^2, \ldots, R_i^m\}$ is the set of shared memory resource constraints such that R_i^j is the set of shared memory resources require to execute τ_i^j. $A_i = \{A_i^1, A_i^2, \ldots, A_i^m\}$ is the processor allocation for τ_i such that A_i^j is the processor assigned to execute τ_i^j. Furthermore we restrict the periodicity and absolute deadline of all tasks to be equal to a user-specified hyperperiod. This constraint corresponds to a pulse-line in the manufacturing environment where all tasks in the task set must be accomplished once every pulse of the line.

$$\tau_i : (\phi_i, (C_i^1, E_i^1, C_i^2, E_i^2, \ldots, E_i^{m_{i-1}}, C_i^{m_i}), R_i, A_i, T_i = H, D_i = H, \boldsymbol{D_i^{rel}}, \boldsymbol{D_i^{abs}}) \quad (4.1)$$

In addition to intra-task deadlines $\boldsymbol{D_i^{rel}}$ for τ_i, we extend our task model to include subtasks deadlines, where $\boldsymbol{D_i^{abs}}$ is the set of subtask deadlines for subtasks in τ_i. As shown in Equation 4.2, if a subtask τ_i^j is constrained by a subtask deadline constraint, then f_i^j must not exceed $d_{i,j}^{abs}$.

$$D_{i,j}^{abs} : \left(f_i^j \leq d_{i,j}^{abs}\right) \quad (4.2)$$

4.3 Terminology

In this section we extend the terminology from Chapter 3 to help describe our multiprocessor scheduling algorithm. In Section 4.4, we use the terms to aid in explaining how we can ensure temporal consistency due to absolute subtask deadlines. In Chapter 3, we defined a *subtask group*, *active intra-task deadline* and *active subtasks*. We now augment the definitions of a subtask group and an active subtask to include absolute subtask deadline; we also define a new term: *active subtask deadline*.

Definition 14. *A subtask group, $\boldsymbol{G_i^j}$, is an ordered set of subtasks that share a common deadline constraint. If we have a deadline constraint $D_{(i,a),(i,b)}$, then the subtask group for that deadline constraint would be the $\boldsymbol{G_i^j} = \{\tau_i^y | j \leq y \leq b\}$. If we have a subtask deadline constraint $D_{i,j}^{abs}$, then a subtask group for that deadline constraint*

would be the $\boldsymbol{G_i^a} = \{\tau_i^y | a \leq y \leq j\}$. *Furthermore,* $\boldsymbol{G_i^j}(k)$ *returns the* k^{th} *element of* $\boldsymbol{G_i^j}$, *where the set is ordered by subtask index (e.g., y associated with* τ_i^y*).*

Definition 15. *An active subtask deadline is a subtask deadline constraint,* $D_{i,j}^{abs}$, *where the processor has not finished* τ_i^j. *Formally* $\boldsymbol{D_{active}^{abs}} = \{D_{i,j}^{abs} | D_{i,j}^{abs} \in \boldsymbol{D^{abs}}, 0 \leq t < f_i^j\}$, *where* $\boldsymbol{D^{abs}}$ *is the set of all intra-task deadlines.*

Definition 16. *The set of active subtasks,* $\boldsymbol{\tau_{active}}$, *are the set of all unfinished subtasks associated with active intra-task deadlines or subtask deadlines. Formally* $\boldsymbol{\tau_{active}} = \{\tau_i^j | \tau_i^j \in \boldsymbol{\tau}, \Big(\exists D_{(i,a),(i,b)}^{rel} \in \boldsymbol{D^{rel}}, a \leq j \leq b, s_i^a \leq s_i^j \leq t < f_i^b\Big) \vee \big(\exists D_{i,a}^{abs} \in \boldsymbol{D^{abs}}, 1 \leq j \leq a, 0 \leq t < f_i^a\big)\}$.

4.4 Multiprocessor Scheduling Algorithm

Our multiprocessor algorithm (Figure 4.4.1) receives as input a self-suspending task set, $\boldsymbol{\tau}$, according to Equation 4.1, and terminates after scheduling all subtasks. Because these tasks have a uniform periodicity, only one instance of each task is scheduled. The algorithm steps through time to schedule released tasks in $\boldsymbol{\tau}$. The order with which processors attempt to schedule subtasks is based on an *processor-priority* heuristic, and the order with which subtasks assigned to a given processor are considered by that processor is based on a set of *subtask priorities*. The scheduler only schedules a subtask iff an online consistency test (i.e., our Multiprocessor Russian Dolls Test) guarantees that doing so does not result in violating another temporal or shared memory resource constraint.

4.4.1 Multiprocessor Scheduling Algorithm: Walk-Through

We now step through the mechanics of our scheduling algorithm. In Line 1, we tighten the intra-task deadlines and subtask deadlines to produce better problem structure. We note that, for the uniprocessor case (Chapter 3.3), we only considered intra-task deadlines. The self-suspending task model for the multiprocessor case (Equation 4.1) includes intra-task deadlines *and* subtask deadlines. These deadlines may "overlap",

```
multiprocessorSchedulingAlgorithm(τ)
 1: D^rel, D^abs ← simplifyDeadlines(D^rel,D^abs)
 2: if ∃D^abs_{i,j} ∈ D^abs then
 3:     if ¬multiprocessorRussianDollsTest(τ_i^j,τ,0,1) then
 4:         return null
 5:     end if
 6: end if
 7: t ← 0
 8: while TRUE do
 9:     A ← prioritizeAgents(A);
10:     for α = 1 to |A| do
11:         if processor a is idle then
12:             availableSubtasks ← getAvailableSubtasks(t,a);
13:             availableSubtasks ← prioritizeTasks(availableSubtasks);
14:             for (k = 1 to |availableTasks|)) do
15:                 τ_i^j ← availableTasks[k];
16:                 if multiprocessorRussianDollsTest(τ_i^j,τ,t,0) then
17:                     scheduleProcessor(τ_i^j,t,α)
18:                     scheduleResource(τ_i^j,t_s,R_i^j)
19:                     break
20:                 end if
21:             end for
22:         end if
23:     end for
24:     if all tasks in τ have been finished then
25:         return schedule
26:     else
27:         t ← t + 1
28:     end if
29: end while
```

Figure 4-1: Pseudocode describing the multiprocessor scheduling algorithm.

meaning that there are at least two deadline constraints who's interaction can categorized by one of three cases:

1. $D^{rel}_{(i,a),(i,b)}$ and $D^{rel}_{(i,y),(i,z)}$ such that $a \leq y \leq b$, which we considered in Chapter 3
2. $D^{abs}_{i,a}$ and $D^{abs}_{i,b}$ such that $a \leq b$.
3. $D^{rel}_{(i,a),(i,b)}$ and $D^{abs}_{i,j}$ such that $a \leq b$.

To tighten the deadlines for all three cases, we can apply the methodology described in Chapter 3 Section 3.3 with augmented formulae to handle subtask deadlines. Intuitively, we remove two overlapping deadline constraints and replace each pair with a new deadline constraint such that any schedule that satisfies the new deadline also satisfies the original deadlines.

For case one where we have two overlapping intra-task deadlines $D^{rel}_{(i,a),(i,b)}$ and $D^{rel}_{(i,y),(i,z)}$), we can apply Equations 4.3 and 4.5. For case two where we have two overlapping absolute deadlines $D^{abs}_{i,a}$ and $D^{abs}_{i,b}$, we can apply Equations 4.4 and 4.6. Finally, for case three where we have an overlapping intra-task deadline constraint and a subtask deadline constraint, we can apply Equations 4.3 and 4.3 for the deadlines' slack and Equation 4.7 to compute the tightness of the new absolute deadline constraint. After replacing all overlapping deadlines such that there are no remaining overlapping in the task set, we proceed to scheduling the task set.

$$\delta_{(i,a),(i,b)} = d^{rel}_{(i,a),(i,b)} - \left(C^b_i + \textstyle\sum_{j=a}^{b-1} C^j_i + E^j_i\right) \tag{4.3}$$

$$\delta^{abs}_{(i,j)} = d^{abs}_{(i,j)} - \left(C^j_i + \textstyle\sum_{j=1}^{j-1} C^j_i + E^j_i\right) \tag{4.4}$$

$$d^{rel}_{(i,a),(i,\max(b,z))} = \min\left(\delta_{(i,a),(i,b)}, \delta_{(i,y),(i,z)}\right) + \left(C^{\max(b,z)}_i + \textstyle\sum_{j=a}^{\max(b,z)-1} C^j_i + E^j_i\right) \tag{4.5}$$

$$d^{abs}_{i,\max(a,b)} = \min\left(\delta^{abs}_{(i,a)}, \delta^{abs}_{(i,b)}\right) + \left(C^{\max(a,b)}_i + \textstyle\sum_{j=1}^{\max(a,b)-1} C^j_i + E^j_i\right) \tag{4.6}$$

$$d^{abs}_{i,\max(j,b)} = \min\left(\delta^{abs}_{(i,j)}, \delta_{(i,a),(i,b)} + \phi_i\right) + \left(C^{\max(b,z)}_i + \textstyle\sum_{j=a}^{\max(b,z)-1} C^j_i + E^j_i\right) \tag{4.7}$$

Next, we determine whether the set of subtask deadlines is temporally consistent (Line 2-6). Our new task model (Equation 4.1) for the multiprocessor case includes subtask deadlines. These deadlines, which activate as soo as the scheduling process

begins (i.e., at $t = 0$, $\boldsymbol{D^{abs}} \in \boldsymbol{D_{active}}$). Therefore, we need to be able to determine at the start if the set of processors will be able to satisfy those deadline constraints. To perform this test, we extend our Russian Dolls Test to handle subtask deadlines, which was originally applied to intra-task deadlines in Chapter 3. We describe this extension is Section 4.4.2

After tightening deadlines and ensuring schedule feasibility due to $\boldsymbol{D^{abs}}$, we initialize our time to zero (Line 7) and schedule all tasks in $\boldsymbol{\tau}$ released during the hyperperiod (i.e., all τ_i in $\boldsymbol{\tau}$) (Lines 8-29). In Line 9, we prioritize the order with which processors attempt to schedule subtasks. Our *processor-priority* heuristic works as follows: consider two processors, α and α'. If the number of subtasks returned by **getAvailableSubtasks**(t,α) is less than or equal to the number of subtasks returned by **getAvailableSubtasks**(t,α'), then we attempt to schedule a subtask on processor α before α'.

To understand the motivation for our *processor-priority* heuristic, we consider the following. Consider a set of processors, each with a set of available subtasks, where any one available subtask could be scheduled at time t. When the first processor schedules one of its available subtasks at time t, that subtask will occupy a set of shared memory resource, R_i^j, and may activate an intra-task deadline. When the other processors attempt to schedule their available subtasks, the algorithm must ensure consistency relative to the constraints generated by the scheduling of the first processor's subtask. Therefore, when the first processor schedules a subtask, the domain of available, feasible subtasks for the other processors equal to or smaller than the domain of available, feasible subtasks before the first processor schedules a subtask.

Since each time a processor schedules a subtask at time t it reduces the number of available, feasible subtasks for other agents at time t, we want to reduce the probability that an agents domain of available, feasible subtasks will be reduced to zero. To reduce this risk, our heuristic orders the scheduling process so that processors with larger domains of available subtasks schedule after processors with smaller domains of available subtasks. We do not consider the set of available *and*

feasible subtasks for this heuristic so that we can reduce the average computational complexity of the algorithm.

In Line 10, we iterate over each processor, α, prioritized by our *processor-priority* heuristic. If the processor α is not busy executing a subtask (Line 11), we collect all released, unexecuted subtasks assigned to processor α (Line 12). In Line 13, we prioritize the order in which processor α considers scheduling its available subtasks. Subtasks are prioritized according to three heuristics.

1. *Precedence* - Consider a situation where processor α' is assigned to subtask τ_i^{j+1} and processor α is assigned to τ_i^j. Furthermore, the only released, unscheduled subtask assigned to α' is τ_i^{j+1}. Until either α schedules τ_i^j or another subtask assigned to α' is released, α' will idle. Recall that τ_i^{j+1} is dependent on τ_i^j by precedence. As such, we prioritize τ_i^j assigned to α according to whether τ_i^{j-1} is completed by a different processor α'. Formally, our metric for subtask precedence prioritization, $\pi_p(\tau_i^j)$, is shown in Equation 4.8.

$$\pi_p(\tau_i^j) = \mathbf{1}_{\left(A_i^j \neq A_i^{j-1}\right)} \tag{4.8}$$

2. *Resource* - If two subtasks, τ_i^j and τ_x^y are both available and unexecuted at time t such that $R_i^j \cap R_x^y \neq \emptyset$ and $A_i^j \neq A_x^y$, the processor will not be able to concurrently execute τ_i^j and τ_x^y due to their shared memory resource constraints. We want to reduce the prevalence of processors being forced to idle due to shared memory resource constraints, so we prioritize subtasks that requires resource set R_i^j based on how many unexecuted subtasks need those resources. Formally, our metric for subtask resource prioritization,$\pi_R(\tau_i^j)$, is shown in Equation 4.9, where $1_{(R_i^j \cap R_x^y \neq \emptyset)}$ equals one if one of the shared memory resources required by τ_i^j and τ_x^y is the same. Otherwise, $1_{(R_i^j \cap R_x^y \neq \emptyset)}$ equals zero. Thus, Equation 4.9 returns the number of unexecuted subtasks using a resource required by τ_i^j.

$$\pi_R(\tau_i^j) = \sum_{\tau_x^y \in \boldsymbol{\tau_{unexecuted}}} \mathbf{1}_{(R_i^j \cap R_x^y \neq \emptyset)} \tag{4.9}$$

3. *Location* - We recall our real-time processor scheduling analogy from Chapter 1, where we model robot workers as computer processors. Because we want to reduce the travel distance and time of the workers, we include a heuristic that prioritizes subtasks according to how close they are to the current location of the worker. If $x_i^j \in \mathbb{R}^\eta$ is the centroid of the space required by the set of resources required to execute τ_i^j, then we can compute a metric for the proximity of one subtask to another, $\pi_l(\tau_i^j, \tau_a^b)$, by Equation 4.10, where τ_i^j is the subtask processor α is considering scheduling, and τ_a^b is the last subtask scheduled by the processor α. We use the square of the Euclidean norm because the square root operation is computationally expensive. If the robot workers are moving along a rail, then $\mathbb{R}^\eta = \mathbb{R}^1$. If the robot workers are moving along a floor or around the surface of a fuselage (i.e., polar coordinates), then $\mathbb{R}^\eta = \mathbb{R}^2$.

$$\pi_l(\tau_i^j, \tau_a^b) = \|x_i^j - x_a^b\|^2 \tag{4.10}$$

To combine these three heuristics into one priority metric, we order subtasks according to a tiered-sort. First, we say that all available subtasks assigned to processor α are prioritized first by one heuristic (e.g., π_p). Among those subtasks that have an equal priority, we further prioritize by a second heuristic (e.g., π_R). We repeat this operation for the third heuristic.

After prioritizing the available subtasks for processor α, we iterate over those subtasks according to their priority (Line 14). We get the next available subtask, τ_i^j (Line 15), and determine whether we can guarantee that our algorithm can find a feasible schedule if processor α and schedules τ_i^j for resource set R_i^j at time t using our online consistency test (Line 16), which we describe in Section 4.4.2. Our test considers both τ_i^j, and the subtasks in τ_i^j's subtask group, G_i^j.

If τ_i^j passes our online consistency test, then we schedule τ_i^j on processor $A_i^j = \alpha$ and shared memory resource set R_i^j (Lines 17-18). After attempting to schedule subtasks on all processors, we increment the clock (Line 27). If all tasks (i.e. all subtasks) in $\boldsymbol{\tau}$ have been scheduled, then the scheduling operation has completed

(Line 25).

4.4.2 Multiprocessor Russian Dolls Test

The Multiprocessor Russian Dolls Test is a schedulability test for ensuring temporal feasibility while scheduling tasks against deadline constraints and shared memory resources. To understand how the Russian Dolls Test works, we must know three temporal and three spatial pieces of information about $\tau_i^j \in \boldsymbol{\tau_{active}}$. We recall a subtask is active if it is associated with an active intra-task or subtask deadline. We define three temporal parameters describing τ_i^j in Definitions 17, 19, and 21, and three spatial parameters describing τ_i^j and Definitions 18, 20, and 22.

Definition 17. *$t_{max}|_{i,\alpha}^j$ is defined as the remaining time available to execute the unexecuted subtasks in G_i^j assigned to processor α provided that at least one subtask in G_i^j is assigned to α. We compute $t_{max}|_{i,\alpha}^j$ using Equation 4.11, where j' is the subtask index of the last, unexecuted subtask in G_i^j assigned to processor α.*

$$t_{max}|_{i,\alpha}^j = \begin{cases} \min \begin{bmatrix} D_{(i,a),(i,b)}^{rel} + s_i^a - \left(\sum_{q=j'+1}^{b} C_i^q + E_i^{q-1}\right) \\ T_i - \left(\sum_{q=j'+1}^{m_i} C_i^q + E_i^{q-1}\right) \end{bmatrix} - t, & \text{if } \exists D_{(i,a),(i,b)}^{rel} | a \leq j \leq b \\ \min \begin{bmatrix} D_{i,b}^{abs} - \left(\sum_{q=j'+1}^{b} C_i^q + E_i^{q-1}\right) \\ T_i - \left(\sum_{q=j'+1}^{m_i} C_i^q + E_i^{q-1}\right) \end{bmatrix} - t, & \text{if } \exists D_{i,b}^{abs} | 1 \leq j \leq b \end{cases} \tag{4.11}$$

Definition 18. *$t_{max}|_{i,r}^j$ is the remaining time available to execute the unexecuted subtasks in G_i^j that require resource r. We compute $t_{max}|_{i,r}^j$ using Equation 4.12, where j' is the subtask index of the last, unexecuted subtask in G_i^j that requires r provided that at least on subtask in G_i^j requires r.*

$$t_{max}|_{i,r}^{j} = \begin{cases} \min \begin{bmatrix} D_{(i,a),(i,b)}^{rel} + s_i^a - \left(\sum_{q=j'+1}^{b} C_i^q + E_i^{q-1}\right) \\ T_i - \left(\sum_{q=j'+1}^{m_i} C_i^q + E_i^{q-1}\right) \end{bmatrix} - t, & \text{if } \exists D_{(i,a),(i,b)}^{rel} | a \leq j \leq b \\ \min \begin{bmatrix} D_{i,b}^{abs} - \left(\sum_{q=j'+1}^{b} C_i^q + E_i^{q-1}\right) \\ T_i - \left(\sum_{q=j'+1}^{m_i} C_i^q + E_i^{q-1}\right) \end{bmatrix} - t, & \text{if } \exists D_{i,b}^{abs} | 1 \leq j \leq b \end{cases} \tag{4.12}$$

Definition 19. *$t_{min}|_{i,\alpha}^{j}$ is a lowerbound on the time processor α will be occupied executing the remaining subtasks in G_i^j assigned to processor α provided that at least on subtask in G_i^j is assigned to α. We compute $t_{min}|_{i,\alpha}^{j}$ using Equation 4.13, where j' is the subtask index of the last, unexecuted subtask in G_i^j assigned to processor α. Because a processor may not be able to interleave a subtask τ_x^y during the self-suspensions between subtasks in G_i^j, we conservatively treat those self-suspensions as task cost in our formulation of $t_{min}|_{i,\alpha}^{j}$.*

$$t_{min}|_{i,\alpha}^{j} = C_i^b + \sum_{q=j}^{b-1} C_i^q + E_i^q, \text{ if } \left(\exists D_{(i,a),(i,b)}^{rel} | a \leq j \leq b\right) \vee \left(\exists D_{i,b}^{abs} | 1 \leq j \leq b\right) \tag{4.13}$$

Definition 20. *$t_{min}|_{i,r}^{j}$ is a lowerbound on the time resource r will be occupied executing the remaining subtasks in G_i^j provided that at least on subtask in G_i^j requires r. We compute $t_{min}|_{i,r}^{j}$ using Equation 4.14, where j' is the subtask index of the last, unexecuted subtask in G_i^j that requires resource r. Because a processor may not be able to interleave a subtask τ_x^y during the self-suspensions between subtasks in G_i^j, we conservatively treat those self-suspensions as task cost in our formulation of $t_{min}|_{i,r}^{j}$.*

$$t_{min}|_{i,r}^{j} = C_i^b + \sum_{q=j}^{b-1} C_i^q + E_i^q, \text{ if } \left(\exists D_{(i,a),(i,b)}^{rel} | a \leq j \leq b\right) \vee \left(\exists D_{i,b}^{abs} | 1 \leq j \leq b\right) \tag{4.14}$$

Definition 21. *$t_{\delta}|_{i,\alpha}^{j}$ is slack time available for the processor to execute subtasks not*

in G_i^j. This duration is equal to the difference between $t_{max}|_{i,\alpha}^j$ and $t_{min}|_{i,\alpha}^j$.

$$t_\delta|_{i,\alpha}^j = t_{max}|_{i,\alpha}^j - t_{min}|_{i,\alpha}^j \tag{4.15}$$

Definition 22. *$t_{slack}|_{i,r}^j$ is slack time available to schedule subtasks not in G_i^j that require resource r not in the G_i^j. This duration is equal to the difference between $t_{max}|_{i,r}^j$ and $t_{min}|_{i,r}^j$.*

$$t_\delta|_{i,r}^j = t_{max}|_{i,r}^j - t_{min}|_{i,r}^j \tag{4.16}$$

Multiprocessor Russian Doll Test: Definition

The Multiprocessor Russian Dolls Test extends from the uniprocessor version of the Russian Dolls Test in Chapter 3, which considers intra-task deadlines for one processor. Different from the uniprocessor version, the multiprocessor test must also consider processor-subtask assignments, shared memory resources, and subtask deadlines.

For intra-task deadlines with shared memory resources, the Multiprocessor Russian Dolls Test uses two criteria akin to the criteria used for the uniprocessor test: first, whether the direct execution of τ_i^j at time t will result in a subtask, τ_x^y, missing its deadline during the interval $t = [s_i^j, f_i^j]$ due to some $D_{(x,w),(x,z)}^{rel}$ or R_x^y, where $w \leq y \leq z$, and, second whether activating this deadline will result in a subtask missing its deadline during the interval $t = [f_i^j, d_{(i,j),(i,b)}^{rel}]$ due to some $D_{(x,w),(x,z)}^{rel}$ or R_x^y.

To check the first criteria, we can merely evaluate whether the cost of τ_i^j (i.e., C_i^j) is less than or equal to the processor slack (e.g., $t_\delta|_{x,\alpha'}^y$) and resource slack (e.g., $t_\delta|_{x,r'}^y$) of every active intra-task or subtask deadline. For the second criteria, if there is a deadline $D_{(i,j),(i,b)}^{rel}$, then we must consider the indirect effects of activating that deadline on the processors assigned to and resources required by $\boldsymbol{G_i^j \cup G_x^z}$ after executing τ_i^j. To satisfy the second criteria we must evaluate two sets of conditions. First, for all active deadlines $D_{(x,w),(x,z)}^{rel}$ and $D_{x,w}^{abs}$ and processors α, we must be able to nest

the subtasks in G_x^w assigned to processor α within the slack of G_a^b ($a \neq x$) assigned to processor α, or vice versa. Second, for all active deadlines $D^{rel}_{(x,w),(x,z)}$ and $D^{abs}_{x,w}$ and shared memory resources r, we must be able to nest the subtasks in G_x^w that require r within the slack of G_a^b ($a \neq x$) that require r, or vice versa.

Lastly, the Russian Dolls Test also accounts for the consistency between subtask deadlines. Because all of the subtask deadlines will be activated as soon as scheduling begins (i.e., at $t = 0$), we need to ensure that all subtasks constrained by these deadlines can satisfy their temporal and shared memory resource. To perform this check, we can use the method for the second criteria for intra-task deadlines, but exclusively for subtask deadlines. First, for all subtask deadlines $D^{abs}_{x,w}$ and processors α, we must be able to nest the subtasks of each G_x^w assigned to processor α within the slack of of G_a^b ($a \neq x$) assigned to processor α, or vice versa. Second, for all active deadlines $D^{abs}_{x,w}$ and shared memory resources r, we must be able to nest the subtasks of each G_x^w that require r within the slack of of the subtask each G_a^b ($a \neq x$) that require r, or vice versa.

Multiprocessor Russian Doll Test: Walk-Through

We provide pseudocode to describe the multiprocessor Russian Dolls Test in Figure 4.4.2. The Russian Dolls Test takes as input a subtask τ_i^j, the task set $\boldsymbol{\tau}$, the current simulation time t, and the *type* of test. The Russian Dolls Test returns the feasibility of the set of subtask deadlines (if type = 1) or the feasibility of scheduling τ_i^j at time t (if type $\neq$ 1).

If the scheduling algorithm calls the Russian Dolls Test to determine the feasibility of subtask deadlines, then we first determine whether we can nest the set of subtask deadlines within each other for all processors required by those subtasks (Line 2). Second, we determine whether we can nest the set of subtask deadlines within each other for all resources required by those subtasks (Line 3). If the nesting for processors and resources is possible, then we guarantee that our multiprocessor scheduling algorithm will find a feasible schedule with respect to subtask deadline constraints.

If the scheduling algorithm calls the Russian Dolls Test to determine the feasibility

scheduling τ_i^j at time t, we first store the processor and resource required to execute τ_i^j (Lines 9-10). In Line 11, we iterate over all subtasks that are *active* and *next* and not to τ_i^j. For a subtask, τ_x^y to be in *active* and *next*, τ_x^{y-1} must have been completed and there must be an intra-task deadline $D^{rel}_{(x,w),(x,z)} \ni w \leq y \leq z$ or a subtask deadline $D^{abs}_{x,z} \ni y \leq z$. In Line 12, we store the k^{th} active and next subtask not τ_i^j.

In Lines 13-17, we evaluate the first consideration of the Russian Dolls Test: whether the cost of τ_i^j (i.e., C_i^j) is less than or equal to the processor and resource slack of all active deadlines that require that processor and resource, respectively. If not, then executing τ_i^j at time t will directly result in τ_x^y missing its deadline, so we return that the nesting is not possible (Line 14). We note that if τ_i^j, with associated deadline $D^{rel}_{(i,j),(i,z)}$, passes the Russian Dolls Test, we do not need to re-test $D^{rel}_{(i,j),(i,z)}$ when attempting to execute any subtask in the set $\{\tau_i^{j+1}, \ldots, \tau_i^z\}$. For the processor to execute a subtask in $\{\tau_i^{j+1}, \ldots, \tau_i^z\}$, we simply test whether the cost of the subtask is less than or equal to the resource and processor slack of every other active deadline (not including $D^{rel}_{(i,j),(i,z)}$) that requires R_i^j and A_i^j.

Next, we evaluate the second consideration of the Russian Dolls Test: if there is a deadline $D^{rel}_{(i,j),(i,b)}$, then we must consider the indirect effects of activating $D^{rel}_{(i,j),(i,b)}$ on the processor after executing τ_i^j. To determine whether the scheduling algorithm will find a feasible schedule if we activate $D^{rel}_{(i,j),(i,b)}$ at time t, we consider whether we can nest the execution of $\{\tau_i^{j+1}, \ldots, \tau_i^b\}$ amongst all other active deadline constraints for the processors (Line 16) and resources (Line 17) required to execute those subtasks. If not, then we cannot guarantee that the scheduling algorithm will find a feasbile schedule, so we return false (Line 18). After iterating over all active, next subtasks not equal to τ_i^j, and we are able to satisfy both criteria of the Russian Dolls Test, we may execute τ_i^j at time t.

4.5 Computational Complexity

We upperbound the computational complexity of our multiprocessor scheduling algorithm at each time step. At each time step, the algorithm would call **multiproces-**

multiprocessorRussianDollsTest(τ_i^j,$\boldsymbol{\tau}$,t,type)

1: **if** type $= 1$ **then**
2: **if** $\left(t_{max}|^{y}_{x,\alpha} \leq t_{\delta}|^{j}_{i,\alpha}\right) \vee \left(t_{\delta}|^{y}_{x,\alpha} \geq t_{max}|^{j}_{i,\alpha}\right), \forall D^{abs}_{i,j}, D^{abs}_{x,y}, \alpha$ **then**
3: **if** $\left(t_{max}|^{y}_{x,r} \leq t_{\delta}|^{j}_{i,r}\right) \vee \left(t_{\delta}|^{y}_{x,r} \geq t_{max}|^{j}_{i,r}\right), \forall D^{abs}_{i,j}, D^{abs}_{x,y}, r$ **then**
4: **return** true
5: **end if**
6: **end if**
7: **return** false
8: **else**
9: $\alpha' \leftarrow A_i^j$
10: $r' \leftarrow R_i^j$
11: **for** $k = 1$ to $|\{(\boldsymbol{\tau_{active}} \cap \boldsymbol{\tau_{next}}) \setminus \tau_i^j\}|$ **do**
12: $\tau_x^y \leftarrow \{(\boldsymbol{\tau_{active}} \cap \boldsymbol{\tau_{next}}) \setminus \tau_i^j\}(k)$
13: **if** $C_i^j > t_{\delta}|^{y}_{x,\alpha'} \wedge C_i^j > t_{\delta}|^{y}_{x,r'}$ **then**
14: **return** false
15: **end if**
16: **if** $\neg\left(\left(t_{max}|^{y}_{x,\alpha} \leq t_{\delta}|^{j+1}_{i,\alpha}\right) \vee \left(t_{\delta}|^{y}_{x,\alpha} \geq t_{max}|^{j+1}_{i,\alpha}\right), \forall\alpha\right)$ **then**
17: **if** $\neg\left(\left(t_{max}|^{y}_{x,r} \leq t_{\delta}|^{j+1}_{i,r}\right) \vee \left(t_{\delta}|^{y}_{x,r} \geq t_{max}|^{j+1}_{i,r}\right), \forall r\right)$ **then**
18: **return** false
19: **end if**
20: **end if**
21: **end for**
22: **end if**
23: **return** true

Figure 4-2: Pseudocode describing the Multiprocessor Russian Dolls Test.

sorRussianDollsTest(τ_i^j,$\boldsymbol{\tau}$,t,type) at most n times, because, at most n subtasks are available at any time t. In the worst case, the Russian Dolls Test evaluations n active deadlines with subtasks assigned to $a = |A|$ processors and $r = |R|$ shared memory resources, where R is the set of all shared memory resources. Thus, the complexity of the Russian Dolls Test is $O(n(a+r))$. At most, a subtasks can pass the Russian Dolls Test at each time step and be scheduled. Therefore, the JSF Scheduling algorithm performs at most $O(n^2(a+r)+a)$ operations for each time step.

4.6 Conclusion

We have presented a multiprocessor scheduling algorithm for hard, non-preemptive, self-suspending task sets with intra-task deadline constraints, subtask deadlines, and shared memory resources. Our scheduling algorithm utilizes a polynomial-time, online consistency test, which we call the Multiprocessor Russian Dolls Test, to ensure temporal consistency due to the temporal and shared memory resource constraints of the task set. Allowing for multiple processors, shared memory resources, and subtask deadlines is and extension from our uniprocessor Russian Dolls Test we develop in In Chapter 3. In Chapter 5, we describe how we embed the multiprocessor scheduling algorithm into a task allocation and scheduling algorithm, Tercio, to provide the capability of dynamically coordinating the teams of robots in the manufacturing environment.

Chapter 5

Fast Scheduling of Multi-Robot Teams with Temporospatial Constraints

5.1 Introduction

In this Chapter, we present Tercio, a centralized task assignment and scheduling algorithm that scales to multi-agent, factory-size problems and supports on-the-fly replanning with temporal and spatial-proximity constraints. We demonstrate that this capability enables human and robotic agents to effectively work together in close proximity to perform manufacturing-relevant tasks.

We begin in Section 5.2 with a review of prior work. In Section 5.3 we formally define the problem we solve, and we outline our technical approach in Section 5.4. In Section 5.5, we provide the Tercio algorithm including pseudocode, and Section 5.6 describes the fast multi-agent task sequencer, called as a subroutine within Tercio. Section 5.7 presents the empirical evaluation, and describes the application of the algorithm in a multi-robot hardware testbed.

5.2 Background

There is a wealth of prior work in task assignment and scheduling for manufacturing and other applications. While the problem in many cases may be readily formulated and solved as a mixed-integer linear program (MILP), the complexity of this approach is exponential and leads to computational intractability for problems of interest in large-scale factory operations [5]. To achieve good scalability characteristics, various hybrid algorithms have been proposed. A brief survey of these methods follows.

One of the most promising approaches has been to combine MILP and constraint programming (CP) methods into a hybrid algorithm using decomposition (e.g. Benders Decomposition) [25, 26, 27]. This formulation is able to gain orders of magnitude in computation time by using a CP subroutine to prune the domain of a relaxed formulation of the MILP. However, if the CP is unable to make meaningful cuts from the search space, this hybrid approach is rendered nearly equivalent to a non-hybrid formulation of the problem. Auction methods (e.g. [6]) also rely on decomposition of problem structure and treat the optimization of each agent's schedule as independent of the other agents' schedules. These techniques preclude explicit coupling in each agent's contribution to the MILP objective function. While the CP and auction-based methods support upperbound and lowerbound temporal deadlines among tasks, they do not handle spatial proximity constraints, as these produce tight dependencies among agents' schedules that make decomposition problematic. Typically, allowing multiple robots to work closely in the same physical space produces dependencies among agents' temporal and spatial constraints, producing uninformative decompositions and non-submodular conditions.

Other hybrid approaches integrate heuristic schedulers within the MILP solver to achieve better scalability characteristics. For example, Chen *et al.* incorporate depth-first search (DFS) with heuristic scheduling [10], and Tan incorporates Tabu Search [49] within the MILP solver. Castro *et al.* use a heuristic scheduler to seed a feasible schedule for the MILP [9], and Kushleyev *et al.* [30] apply heuristics for abstracting the problem to groupings of agents. These methods solve scheduling

problems with 5 agents (or groups of agents) and 50 tasks in seconds or minutes, and address problems with multiple agents and resources, precedence among tasks, and temporal constraints relating task start and end times to the plan epoch time. However, more general task-task temporal constraints are not considered.

5.3 Formal Problem Description

In this section, we formulate the task assignment and scheduling problem for multiple robots moving and working in the same physical space as a mixed-integer linear program (MILP). Problem inputs include:

- a **Simple Temporal Problem (STP)** [14] describing the interval temporal constraints (lowerbound lb and upperbound ub) relating tasks (e.g. "the first coat of paint requires 30 minutes to dry before the second coat may be applied" maps to interval constraint $secondCoatStart - firstCoatFinish \in [30, inf)$, and the constraint that "the entire task set must be finished within the 120 minutes" maps to $finishTaskset - startTaskSet \in [0, 120]$),
- **two-dimensional cartesian positions** specifying the floor spatial locations where tasks are performed (in our manufacturing application this is location on the factory floor),
- **agent capabilities** specifying the tasks each agent may perform and the agent's expected time to complete each task, and
- **allowable spatial proximity** between each pair of agents.

A solution to the problem consists of an assignment of tasks to agents and a schedule for each agent's tasks such that all constraints are satisfied and the objective function is minimized. The mathematical formulation of the problem is presented below:

$$\min(z),\ z = f(A, A_p, x, s, f, \boldsymbol{\tau}, \boldsymbol{\gamma}) \tag{5.1}$$

subject to

$$\sum_{a \in \boldsymbol{A}} A_{a,i} = 1, \forall i \in \boldsymbol{\tau} \tag{5.2}$$

$$ub_{i,j} \leq f_i - s_j \leq ub_{i,j}, \forall i \in \boldsymbol{\gamma} \tag{5.3}$$

$$f_i - s_i \geq lb_{a,i} - M(1 - A_{i,k}), \forall \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A} \tag{5.4}$$

$$f_i - s_i \leq ub_{a,i} + M(1 - A_{a,i}), \forall \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A} \tag{5.5}$$

$$s_j - f_i \geq M(1 - x_{i,j}), \forall \tau_i, \tau_j \in \boldsymbol{\tau_R} \tag{5.6}$$

$$s_i - f_j \geq M x_{i,j}, \forall \tau_i, \tau_j \in \boldsymbol{\tau_R} \tag{5.7}$$

$$s_j - f_i \geq M(1 - x_{i,j}) + M(2 - A_{a,i} - A_{a,j}) \forall \tau_i, \tau_j \in \boldsymbol{\tau} \tag{5.8}$$

$$s_i - f_j \geq M x_{i,j} + M(2 - A_{a,i} - A_{a,j}) \forall \tau_i, \tau_j \in \boldsymbol{\tau} \tag{5.9}$$

In this formulation, $A_{a,i} \in \{0, 1\}$ is a binary decision variable for the assignment of agent a to task τ_i, $x_{i,j}$ is a binary decision variable specifying whether τ_i comes before or after τ_j, and s_i, f_i are the start and finish times of τ_i. $\boldsymbol{A}$ is the set of all agents a, $\boldsymbol{\tau}$ is the set of all tasks, τ_i, $\boldsymbol{\tau_R}$ is the set of all the set of task pairs (i, j) that are separated by less than the allowable spatial proximity. $\boldsymbol{\gamma}$ is the set of all temporal constraints defined by the task set, and is equivalently encoded and referred to as the Simple Temporal Problem (STP) [14]. M is an artificial variable set to a large positive number, and is used to encode conditional constraints. Figure 5-1 visually depicts a problem instance of this MILP, with two robots and six tasks (depicted as six stripes on the workpiece).

Equation 5.2 ensures that each task is assigned to one agent. Equation 5.3 ensures that the temporal constraints relating tasks are met. Equations 5.4 and 5.5 ensure that agents are not required to complete tasks faster or slower than they are capable. Equations 5.6 and 5.7 sequence actions to ensure that agents performing tasks maintain safe buffer distances from one another. Equations 5.8 and 5.9 ensure that each agent only performs one task at a time. Note Equations 5.6 and 5.7 couple the variables relating sequencing constraints, spatial locations, and task start and end

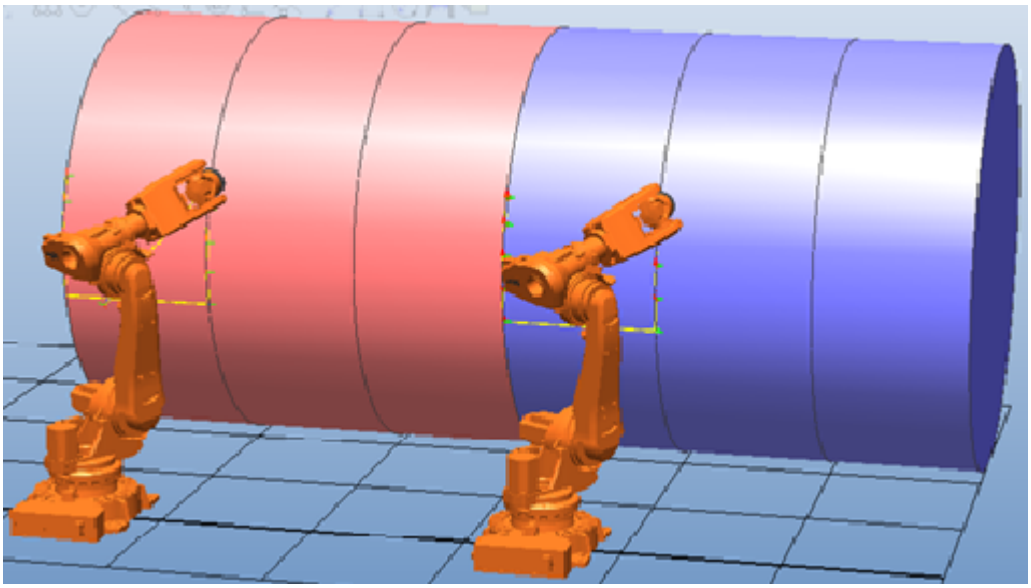

Figure 5-1: Example of a team of robots assigned to tasks on a cylindrical structure.

times, resulting in tight dependencies among agents' schedules.

The objective function $ff(A, A_p, x, s, f, R, \boldsymbol{\tau}, \boldsymbol{\gamma})$ is application specific. In our empirical evaluation in Section 5.7 we use an objective function that includes three equally weighted terms. The first term minimizes $f_1(A, A_p, \boldsymbol{\tau})$, the difference between the previous agent assignment and the returned agent assignment. Minimizing this quantity helps to avoid oscillation among solutions with equivalent quality during replanning. The second term $f_2(A)$ minimizes the number of spatial interfaces between tasks performed by different robots. Inter-robot accuracy is challenging for multi-robot systems of standard industrial robots. In robot painting, this can lead to gaps or overlaps at interfaces between work done by two different robots, and so we seek a task assignment with the fewest interfaces possible. In Figure 5-1 the agent allocation results in one interface between the red work assigned to the left robot and the blue work assigned to the right robot. The third term $f_3(A, x, s, f, \gamma)$ minimizes the sum of the idle time for each agent in the system, which is functionally equivalent to minimizing the time to complete the entire process (i.e. the makespan).

5.4 Our Approach

In this section, we outline our technical approach for efficiently solving this MILP. Tercio is made efficient through a fast, satisficing, incomplete multi-agent task sequencer that is inspired by real-time processor scheduling techniques that leverage problem struture. We decompose the MILP into a task allocation and a task sequencing problem. We modify the MILP to support this decomposition, and use the task sequencer to efficiently solve for the subproblem involving Equations 5.3-5.9 and objective term $f_3(A, J, S, E, \gamma)$. We demonstrate that this approach is able to generate near-optimal schedules for up to 10 agents and 500 work packages in less than 20 seconds.

Real-Time Processor Scheduling Analogy

We use a processor scheduling analogy to inspire the design of an informative, polynomial-time task sequencer. In this analogy, each agent is a computer processor that can perform one task at a time. A physical location in discretized space is considered a shared memory resource that may be accessed by up to one processor at a time. Wait constraints (lowerbounds on interval temporal constraints) are modeled as "self-suspensions," [33, 44] times during which a task is blocking while another piece of hardware completes a time-durative task.

Typically, assembly manufacturing tasks have more structure (e.g., parallel and sequential subcomponents), as well as more complex temporal constraints than are typical for real-time processor scheduling problems. AI scheduling methods handle complex temporal constraints and gain computational tractability by leveraging hierarchical structure in the plan [46]. We bridge the approaches in AI scheduling and real-time processor scheduling to provide a fast multi-agent task sequencer that satisfies tightly coupled upperbound and lowerbound temporal deadlines and spatial proximity restrictions (shared resource constraints). While our method relies on a plan structure composed of parallel and sequential elements, we nonetheless find this structural limitation sufficient to represent many real-world factory scheduling

TERCIO($STP, P_{a,i}, Ag, \gamma, R, cutoff$)

1: $makespan \leftarrow$ inf
2: **while** $makespan \geq cutoff$ **do**
3: $A \leftarrow$ exclude previous allocation $P_{a,i}$ from agent capabilities
4: $A \leftarrow$ TERCIO-ALLOCATION(γ, STP, Ag)
5: $STP \leftarrow$ update agent capabilities
6: $makespan, seq \leftarrow$ TERCIO-SEQUENCER($A, STP, R, cutoff$)
7: **end while**
8: $STP \leftarrow$ add ordering constraints to enforce seq
9: $STP \leftarrow$ DISPATCHABLE(STP)
10: **return** STP

Figure 5-2: Psuedo-code for the Tercio Algorithm.

problems.

5.5 Tercio

In this section, we present Tercio, a centralized task assignment and scheduling algorithm that scales to multi-agent, factory-size problems and supports on-the-fly replanning with temporal and spatial-proximity constraints. Pseudo-code for the Tercio algorithm is presented in Figure 5-2.

The inputs to Tercio are as described in Section 5.3. Tercio also takes as input a user-specified makespan $cutoff$ (Line 2) to terminate the optimization process. This can often be derived from the temporal constraints of the manufacturing process. For example, a user may specify that the provided task set must be completed within an eight-hour shift. Tercio then iterates (Lines 3-7) to compute an agent allocation and schedule that meets this makespan. Because Tercio uses a satisficing and incomplete sequencer, it is not guaranteed to find an optimal solution, or even a satisficing solution if one exists. In practice, we show (Section 5.7) Tercio produces makespans within about 10% of the optimal minimum makespan, for real-world structured problems.

5.5.1 Tercio: Agent Allocation

Tercio performs agent-task allocation by solving a simplified version of the MILP from Section 5.3. The objective function for the agent allocation MILP is formulated as follows:

$$\min(z),\ z = f_1(A, P, \gamma) + f_2(A) + v \tag{5.10}$$

where, recall g minimizes the difference between the previous agent assignment and the returned agent assignment to help avoid oscillations between equivalent quality solutions during replanning, and h minimizes the number of spatial interfaces between tasks performed by different robots.

We introduce a proxy variable v into the objective function to perform work-balancing and guide the optimization towards agent allocations that yield a low makespan. The variable v encodes the maximum total task time that all agents would complete their tasks, if those tasks had no deadline or delay dependencies and is defined as:

$$v \geq \sum_j c_j \times A_{a,j} \forall a \in \boldsymbol{A} \tag{5.11}$$

where c_j is a constant representing the expected time of each task. We find in practice the addition of this objective term and constraint guides the solution to more efficient agent allocations. The agent allocation MILP must also include Equations 5.2 and 5.3 ensuring each task is assigned to exactly one agent and that the agent-task allocation does not violate the STP constraints.

5.5.2 Tercio: Pseudocode

A third-party optimizer [1] solves the simplified agent-allocation MILP (Line 4) and returns the agent allocation matrix A. Interval temporal (STP) constraints are updated based on this agent allocation matrix by tightening task time intervals (Line 5). For example, if a task is originally designated to take between five and fifteen minutes but the assigned robot can complete it no faster than ten minutes, we tighten

the interval from $[5, 15]$ to $[10, 15]$.

The agent allocation matrix, the capability-updated STP, and the spatial map of tasks are then provided as input to the Tercio multi-agent task sequencer (Line 6). The task sequencer (described further in Section 5.6) returns a tight upperbound on the optimal makespan for the given agent allocation as well as a sequence of tasks for each agent.

While this makespan is longer than $cutoff$, the algorithm iterates (Lines 3-7), each time adding a constraint (Line 3) to exclude the agent allocations tried previously:

$$\sum_{a,i|L_{a,i}=0} A_{a,i} + \sum_{a,i|L_{a,i}=1} (1 - A_{a,i}) > 0 \tag{5.12}$$

where $L_{a,i}$ is the solution from the last loop iteration.

Tercio terminates when the returned makespan falls beneath $cutoff$, or else when no solution can be found after iterating through all feasible agent allocations. If the cutoff makespan is satisfied, agent sequencing constraints (interval form of $[0, \infty)$) are added to the STP constraints (Line 8). Finally the resulting Simple Temporal Problem is compiled to a dispatchable form (Line 9) [38, 52], which guarantees that for any consistent choice of a timepoint within a flexible window, there exists an optimal solution that can be found in the future through one-step propagation of interval bounds. The dispatchable form maintains flexibility to increase robustness to disturbances, and has been shown to decrease the amount of time spent recomputing solutions in response to disturbances by up to 75% for randomly generated structured problems [52].

5.6 Tercio: Multi-agent Task Sequencer

The key to increasing the computational speed of Tercio is our hybrid approach to task sequencing. Tercio takes as input a set of agent-task assignments and a well-formed self-suspending task model (defined below), and returns a valid task sequence if one can be found by the algorithm. The task sequencer is merely satisficing

and is not complete; however, we empirically validate that it returns near-optimal makespans when integrated with the Tercio Agent Allocation algorithm (See Section 5.7). In this section, we provide an overview of the multi-agent task sequencer by first introducing our task model, which is inspired by real-time systems scheduling. Second we describe how our fast task sequencer works to satisfy temporospatial constraints. A full treatment can be found in Chapter 4.

5.6.1 Well-Formed Task Model

The Tercio Task Sequencer relies on a well-formed task model that captures hierarchical and precedence structure in the task network. The basis for our framework is the self-suspending task model [33], described in Equation 5.13.

$$\tau_i : ((C_i^1, E_i^1, C_i^2, E_i^2, \ldots, E_i^{m_{i-1}}, C_i^{m_i}), A_i, T_i, D_i). \tag{5.13}$$

In this model, there is an instance, I, with a set of tasks, $\boldsymbol{\tau}$, that must be processed by the computer. For each task, there are m_i subtasks with $m_i - 1$ self-suspension intervals for each task $\tau_i \in \boldsymbol{\tau}$. We use τ_i^k to denote the k^{th} subtask of τ_i, C_i^k is the expected duration (cost) of τ_i^k. E_i^k is the expected duration of the k^{th} self-suspension interval of τ_i. T_i and D_i are the period and deadline of τ_i, respectively. Furthermore, subtask τ_i^j is assigned to processor A_i^j.

The standard self-suspending task model provides a solid basis for describing many real-world processor scheduling problems of interest. In this work we present an augmented model to better capture problem structure inherent in the manufacturing environment:

$$\tau_i : (\phi_i, (C_i^1, E_i^1, \ldots, E_i^{m_{i-1}}, C_i^{m_i}), A_i, T_i = H, D_i = H, \boldsymbol{D_i^{rel}}, \boldsymbol{D_i^{abs}}, \boldsymbol{R_i}) \tag{5.14}$$

where we set the implicit deadlines of the tasks equal to the period of the task set. This modification models well many assembly line manufacturing processes where the set of tasks at one location is repeated once every "pulse" of the production line. In

this scenario, the user allots a certain amount of time, T, for the set of tasks to be accomplished, and the set of tasks is repeated with a period of T. For convenience, we will not explicitly state ($T_i = T, D_i = T$) from Equation 5.14 in the remainder of the paper; however, these constraints are implied. For convenience, we also assume all tasks are non-preemptable, meaning the interruption of a subtask significantly degrades its quality.

The second adaptation we make is to allow phase offsets for each task, where a phase offset is a delay between the epoch time and the release of the given task. This allows a user expressiveness to require that an agent wait a specified time interval before starting the first subtask of a task.

The third change we make is to enable the user to specify intra-task deadlines, $D^{rel}_{(i,j),(i,b)}$ between the start and end of two subtasks for a given task and subtask deadlines, $D^{abs}_{i,j}$ for an individual subtask. We define an intra-task deadline in Equation 5.15 and a subtask deadline in Equation 5.16. The set of intra-task deadlines for τ_i is $\boldsymbol{D^{rel}_i}$.

$$D^{rel}_{(i,j),(i,b)} : \left(f(\tau^b_i) - s(\tau^j_i) \leq d_{(i,j),(i,b)}\right) \ni 1 \leq j \leq b \leq m_i \tag{5.15}$$

$$D^{abs}_{(i,b)} : \left(f(\tau^b_i) \leq d^{abs}_{(i,b)}\right) \ni 1 \leq b \leq m_i \tag{5.16}$$

where the operator f^b_i is the finish time of subtask τ^b_i, s^j_i is the start time of subtask τ^j_i. For Equation 5.15, $d^{rel}_{(i,j),(i,b)}$ is the upperbound temporal constraint between τ^j_i and τ^b_i. For Equation 5.16 $d^{abs}_{i,b}$ is the absolute deadline for τ^b_i. This deadline constraints provide additional expressiveness to encode binary temporal constraints relating tasks in the manufacturing process. For instance, these constraints may be used to specify that a sequence of subtasks related to sealant application must be completed within a half hour after opening the sealant container. These types of constraints are commonly included in AI and operations research scheduling models [5, 14, 38, 52].

We also extend the model to include shared memory resources. Each subtask τ^j_i requires a set of k_i shared memory resources $R^j_i = \{R^1_{i,j}, \ldots, R^{k_i}_{i,k}\}$ be utilized to

perform that subtask (e.g. for memory shared among multiple processors). In the manufacturing analogy, a shared memory resource corresponds to a region of space in the factory that must physically be unoccupied for an agent to execute a subtask there. These shared memory resources are used to encode hard spatial constraints that prohibit agents from working in close physical proximity.

Next we describe how the Tercio Task Sequencer leverages the structure of the well-formed task model to compute a schedule that satisfies upperbound and lowerbound temporal constraints, as well as spatial-proximity restrictions. To our knowledge, this is the first real-time scheduling method for multi-processor systems that (1) tests the schedulability of self-suspending, non-preemptive tasks where multiple tasks in $\boldsymbol{\tau}$ have more than one self-suspension [33, 32, 43], and (2) that extends self-suspending models to include shared memory resources.

5.6.2 Multi-agent Task Sequencer Overview

The Tercio multi-agent task sequencer pseudo-code takes as input a task set $\boldsymbol{\tau}$ of a well-formed task model (also called the STP constraints) and the user-specified makespan $cutoff$. The algorithm returns a valid task sequence for each agent, if one can be found, and an upperbound on the time to complete all tasks. This upperbound on completion time is compared to $cutoff$ to test schedulability of a well-formed task model.

Restriction on the Behavior of the Scheduler

We introduce three task-based heuristic restrictions an agent based heuristic restriction and on the scheduling simulation that address the types of schedule bottlenecks that hinder efficient execution in the well-formed task model augmented with shared memory resources.

First, we introduce a heuristic $\pi_p(\tau_i^j)$ (Equation 4.8) to ease bottlenecks due to inter-agent precedence constraints. Second, we introduce a heuristic $\pi_R(\tau_i^j)$ (Equation 4.9) that prioritizes subtasks according to the number of other subtasks in the model

that will require the same resource. Lastly, we introduce a heuristic $\pi_l(\tau_i^j)$ (Equation 4.10) to reduce travel distance. While travel time is small relatively to the task durations of interest, we do not want robots to move erratically about the space from the point of view of their human teammates.

These three heuristics can be combined in various formulations. We have chosen a tiered system, in which tasks are ordered according to one heuristic. Then, for all tasks that have the same value based on the first heuristic, those tasks are ordered according to a second heuristic. This process is repeated for the last heuristic. The order in which the heuristics are applied can be tuned *a priori* with the knowledge of what kind of bottlenecks govern the system. For our real-world problems in large-scale assembly manufacturing, we find the the key bottleneck is the need to satisfy precedence constraints between subtasks assigned to different agents. For the results discussed in this paper, we order the heuristics as follows: 1) $\pi_p(\tau_i^j)$, 2) $\pi_R(\tau_i^j)$, 3) $\pi_l(\tau_i^j)$.

We also use an agent-based heuristic to improve the efficiency of the schedule. The order with which agents are scheduled is chosen based on how many released, unexecuted subtasks are assigned to the agents. When each agent is scheduled, the domain of subtasks that can be correctly scheduled either is unchanged or decreases. If an agent with relatively few subtask options schedules last, that agent may be forced to idle because of constraints imposed by the other agents schedules. Therefore, we prioritize the order with which agents are scheduled such that agents with more released, unexecuted subtasks schedule after those with fewer released, unexecuted subtasks.

Finally, our scheduling policy requires that an agent not idle if there is an available subtask, unless executing that subtask will violate a subtask-to-subtask deadline constraint. This condition is checked via an online temporospatial consistency test.

Multi-Agent Online Consistency Check.

During the scheduling simulation, we perform an online consistency check, which we call the Multiprocessor Russian Dolls Test (Chapter 4), that ensures that the

scheduling of a next subtask, τ_i^j, will not result a missed deadline for a different subtask, τ_x^y, or violate a spatial-proximity constraint. Note, by the definition of a well-formed task model, τ_i and τ_j must be different tasks for their subtasks to be scheduled concurrently.

Our well-formed self-suspending task model includes absolute deadlines, $D_{(i,b)}^{abs}$ relating a subtask τ_i^b to the plan epoch time, and inter-subtask deadline, $D_{(i,j),(i,b)}$, from τ_i^j to τ_i^b. We now introduce a definition for the an *active deadline*, which we use to describe our online consistency test.

Definition 23. Active Deadline - *Consider an intra-task deadline* $D_{(i,j),(i,b)}^{rel}$*, or an absolute deadline* $D_{i,j}^{abs}$*. An intra-task deadline is considered* active *between* $0 \leq t \leq \min(f_{i,j}, D_{(i,j),(i,b)}^{rel})$*, and an absolute deadline is considered* active *between* $0 \leq t \leq \min f_i^j, D_{i,j}^{abs}$.

We readily formulate our online consistency test as a constraint satisfaction problem. First, we will consider the case where we have multiple, active, inter-task deadlines. We evaluate Equations 5.17-5.22 for the union of all active deadlines and the deadline we are considering activating.

$$\delta_{(i,j),(i,k)}^{a} \geq \gamma_{(x,y),(x,z)}^{a}, \forall D_i^{\beta_a(x,y,z,a)} D_i^{\beta_a(i,j,k,a)}, \forall a \in \boldsymbol{A} \tag{5.17}$$

$$\delta_{i,j:k}^{a} = \gamma_{(i,j),(i,k)}^{a} - \left(C_i^{\beta_a(i,j,k,a)} + \textstyle\sum_{\psi=\xi_a(i,j,k,a)}^{\beta_a(i,j,k,a)-1} (C_i^{\psi} + E_i^{\psi}) \right) \tag{5.18}$$

$$\gamma_{(x,y),(x,z)}^{a} = D_i^{\beta(i,j,k,a)} - t \tag{5.19}$$

$$\delta_{(i,j),(i,k)}^{r} \geq \gamma_{(x,y),(x,z)}^{r}, \forall D_i^{\beta_r(x,y,z,r)} D_i^{\beta_r(i,j,k,r)}, \forall r \in \boldsymbol{R} \tag{5.20}$$

$$\delta_{i,j:k}^{r} = \gamma_{(i,j),(i,k)}^{r} - \left(C_i^{\beta_r(i,j,k,r)} + \textstyle\sum_{\psi=\xi_r(i,j,k,r)}^{\beta_r(i,j,k,a)-1} (C_i^{\psi} + E_i^{\psi}) \right) \tag{5.21}$$

$$\gamma_{(x,y),(x,z)}^{r} = D_i^{\beta(i,j,k,r)} - t \tag{5.22}$$

Equation 5.17 determines whether or not we can "nest" a set of tasks within the "slack time" of the deadline associated with another set of tasks. Specifically, we must ensure that $\gamma_{(i,j),(i,k)}^{a}$, defined as the amount of time an agent a is occupied with subtasks $\{\tau_i^j, \ldots, \tau_i^k\}$ associated with $D_{(i,j),(i,k)}^{rel}$ (or $D_{i,k}^{abs}$ where $j = 1$), is less than

the slack time, $\delta^a_{x,y:z}$, for the agent a's other commitments to $\{\tau_i^j, \ldots, \tau_i^k\}$ associated with active deadline $D^{rel}_{(x,y),(x,z)}$ or $D_(x,y)^{abs}$. Slack is calculated in Equations 5.18-5.19, where $\xi_a(i,j,k,a)$ and $\beta_a(i,j,k,a)$ refer respectively the next and last subtask to which agent a is assigned to in $\{\tau_i^j, \ldots, \tau_i^k\}$.

We utilize the same methodology for spatial constraints in Equations 5.17- 5.19. Specifically, we test whether $\gamma^r_{(i,j),(i,k)}$, defined as the amount of time an resource r is occupied with subtasks $\{\tau_i^j, \ldots, \tau_i^k\}$ associated with $D^{rel}_{(i,j),(i,k)}$ or $D^{abs}_{i,k}$, is less than the slack time, $\delta^r_{x,y:z}$, for the resource r's other commitments to $\{\tau_i^j, \ldots, \tau_i^k\}$ associated with active deadline $D^{rel}_{(x,y),(x,z)}$ or $D_(x,y)^{abs}$.

Our multi-agent sequencer uses a polynomial-time version of this online consistency test to evaluate the feasibility of scheduling subtasks. A full description of the online consistency test is provided in Chapter 4 Section 4.4.2. The complexity of this consistency check is $O(n(a+r))$ where n is the number of tasks, a is the number of agents, and r is the number of resources.

5.7 Evaluation and Discussion

In this section, we empirically validate that Tercio is fast and produces near-optimal solutions for the multi-agent task assignment and scheduling problem with temporal and spatial-proximity constraints.

5.7.1 Generating Random Problems

We evaluate the performance of Tercio on randomly generated, structured problems that simulate multi-agent construction of a large structural workpiece, such as an airplane fuselage or wing.

Task times are generated from a uniform distribution in the interval $[1, 10]$. Approximately 25% of the subtasks are related via a nonzero wait duration (lowebound constraint) drawn from the interval $[1, 10]$, and approximately 25% of the subtasks are related via an upperbound temporal deadline generated randomly to another subtask. The upperbound of each deadline constraint, $D^{rel}_{(i,j),(x,y)}$, is drawn from a normal

distribution with mean set to the lowerbound temporal duration between the start and end of the set of subtasks in $D^{rel}_{(i,j),(x,y)}$. Physical locations of a subtask are drawn from a uniform distribution in $[1, n]$ where n is the total number of subtasks in the problem instance, $\boldsymbol{\tau}$. Lastly, we vary the number of subtasks, m_i, within each task, τ_i, from a uniform distribution in the interval $[0.25 \times numTasks, 1.25 \times numTasks]$, where $numTasks$ is the number of $\tau_i \in I$.

5.7.2 Computation Speeds

In Figure 5-3 we evaluate scalability and computational speed of Tercio. We show the median and quartiles of computation time for 25 randomly generated problems, spanning 4 and 10 agents, and 5 to 500 tasks (referred to as subtasks in the well-formed model). For comparison, we show computation time for solving the full MILP formulation of the problem, described in Section 5.3. Tercio is able to generate flexible schedules for 10 agents and 500 tasks in seconds. This is a significant improvement over prior work [9, 10, 49, 30], which report solving up to 5 agents (or agent groups) and 50 tasks in seconds or minutes.

5.7.3 Optimality Levels

Our fast computation relies on the known structure of our well-formed task model, but it is desirable to be able to take as input general sets of temporal (STP) constraints. General STPs can be reformulated into well-formed task models by adding and tightening well-formed temporal constraints to make the constraints that violate the well-formed model redundant. We present results with both random problems that are well-formed and problems that are general but have been reformulated into a well-formed task model.

In Figures 5-4-5-5 we show that Tercio is often able to achieve makespans within 10% of the optimal makespan for well-formed models and 15% of the optimal makespan for general STPs; Tercio is also able to produce less than four additional interfaces when compared to the optimal task allocation for well-formed models and less than

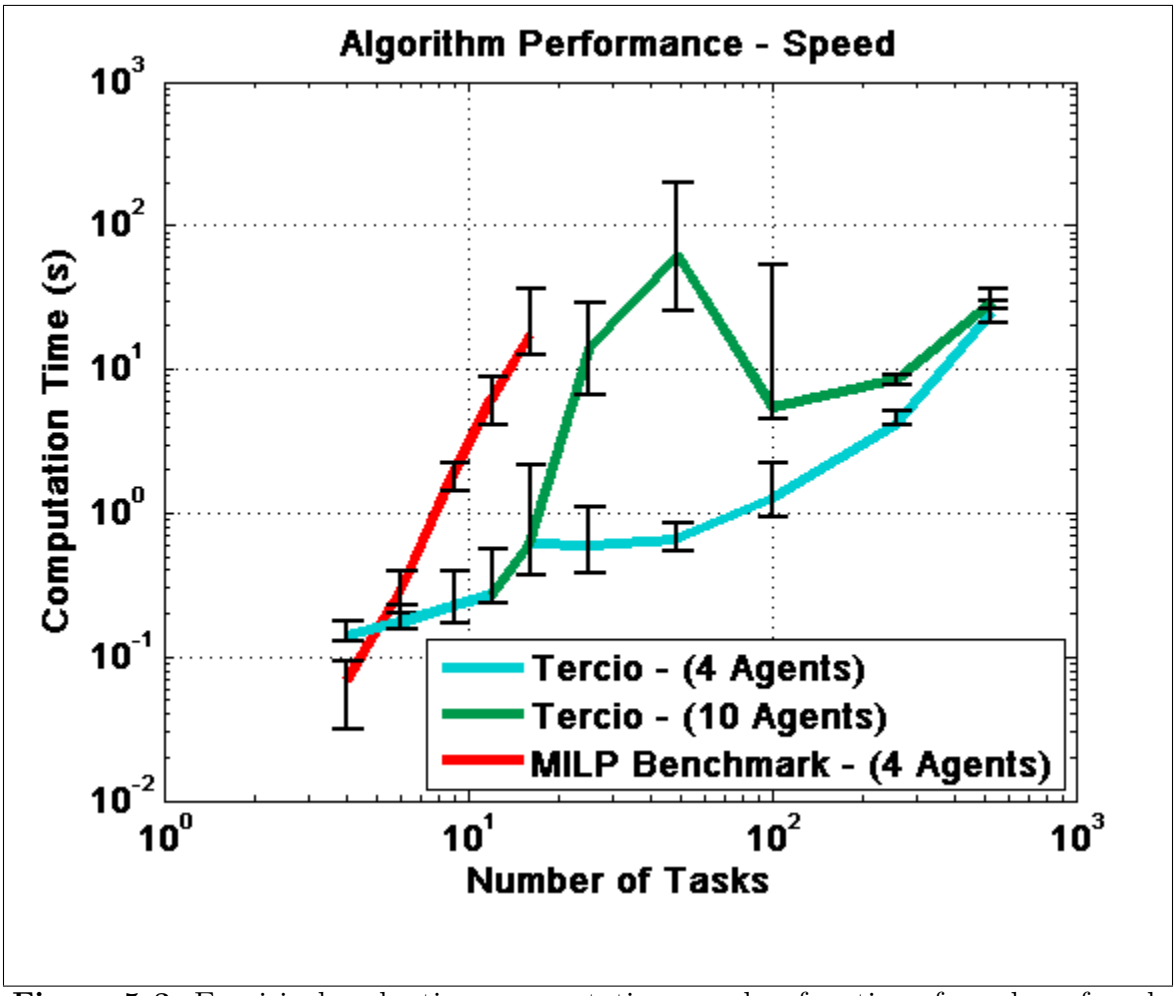

Figure 5-3: Empirical evaluation: computation speed as function of number of work packages and number of agents. Results generated on an Intel Core i7-2820QM CPU 2.30GHz.

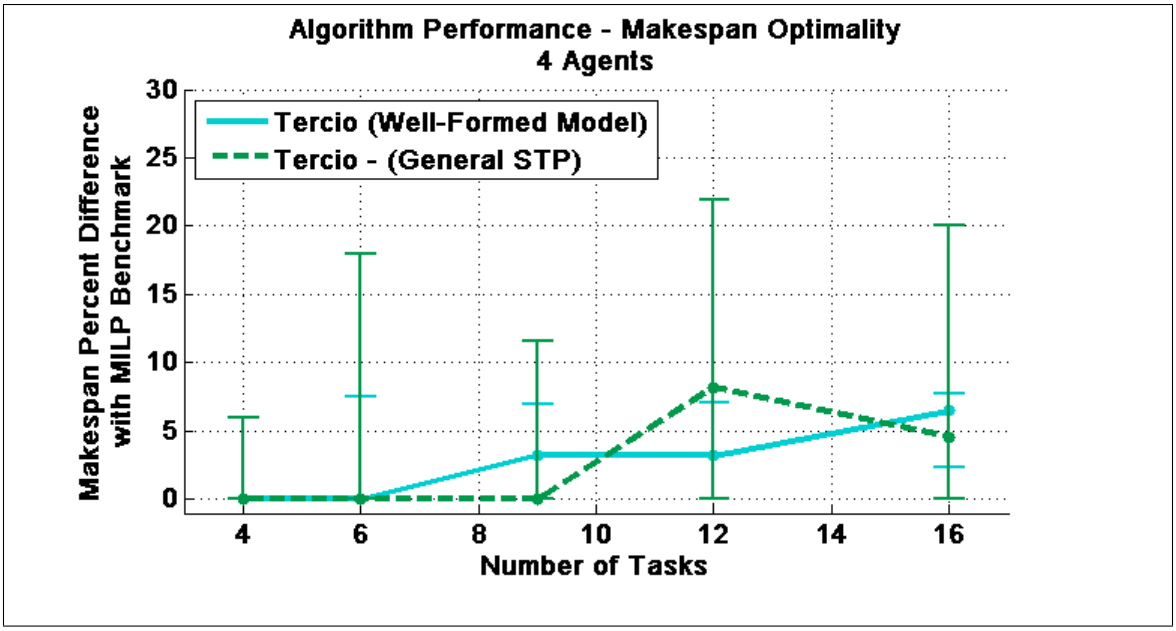

Figure 5-4: Empirical evaluation: Tercio suboptimality in makespan for problems with 4 agents.

eight additional interfaces for general models. We are unable to measure the suboptimality gap for larger problem instances due to the computational intractability of the full MILP. The purpose of Tercio is to solve the problem of scheduling with tens of agents and hundreds of tasks; as we can see in Figure 5-4, Tercio tightly tracks the optimal solution.

5.7.4 Robot Demonstration

We demonstrate the use of Tercio to plan the work of two KUKA Youbots. Video can be found at `http://tiny.cc/t6wjxw`. The two robots are working to assemble a mock airplane fuselage. The robots perform their subtasks at specific locations on the factory floor. To prevent collisions, each robot reserves both the physical location for its subtask, as well as the immediately adjacent subtask locations. Initially, the robots plan to split twelve identical tasks in half down the middle of the fuselage. After the robots finish their first subtasks, a person requests time to inspect the work completed on the left half of the fuselage. In the problem formulation, this corresponds to adding a resource reservation for the left half of the fuselage for a specified period of time. Tercio replans in response to the addition of this new constraint, and reallocates the

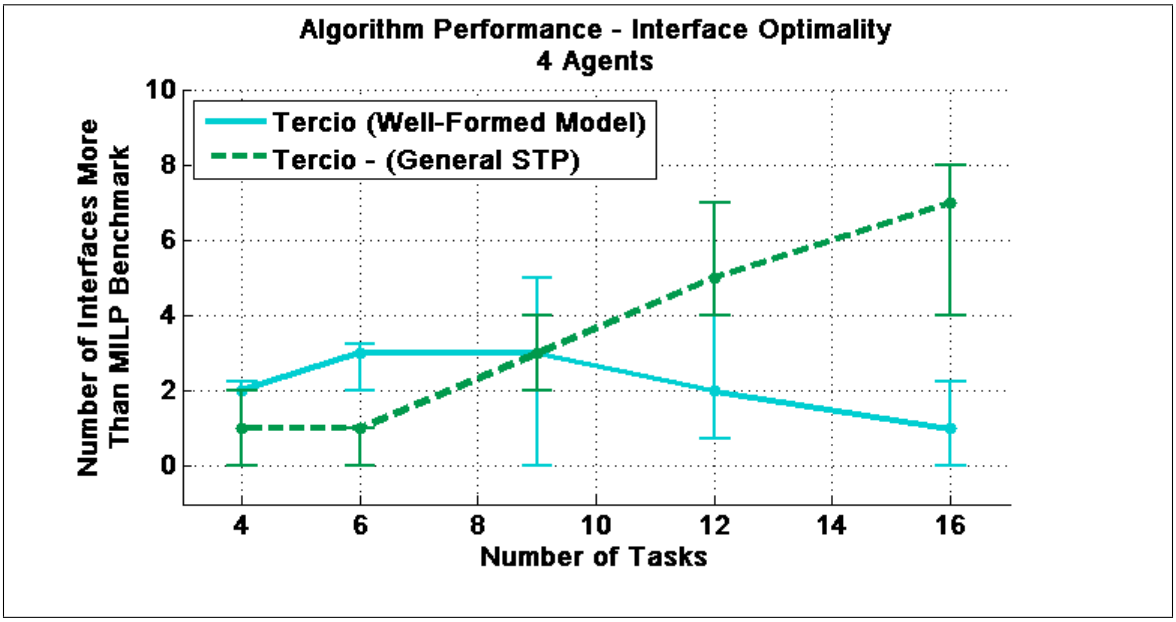

Figure 5-5: Empirical evaluation: Tercio suboptimality in number of interfaces for problems with 4 agents.

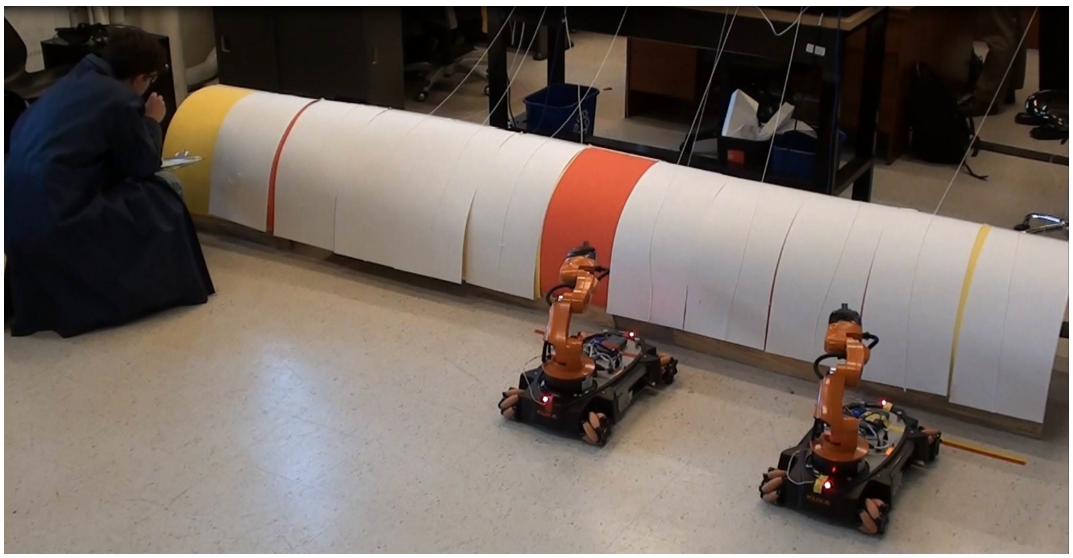

Figure 5-6: Hardware demonstration of Tercio. Two KUKA Youbots build a mock airplane fuselage. A human worker requests time on the left half of the fuselage to perform a quality assurance inspection, and the robots replan.

work among the robots in a near-optimal manner to make productive use of both robots and to keep the number of interfaces low.

5.8 Conclusion

We present Tercio, a task assignment and scheduling algorithm that is made efficient through a fast, satisficing, incomplete multi-agent task sequencer inspired by real-time processor scheduling techniques. We use the fast task sequencer in conjunction with a MILP solver to compute an integrated multi-agent task sequence that satisfies precedence, temporal and spatial-proximity constraints. We demonstrate that Tercio generates near-optimal schedules for up to 10 agents and 500 tasks in less than 20 seconds on average.

Chapter 6

Towards Successful Coordination of Human and Robotic Work using Automated Scheduling Tools: An Initial Pilot Study

6.1 Introduction

In this chapter, we will seek to understand how much control human workers should have over the assignment of roles and schedules, when working in a team with robot partners. Successful integration of robot systems into human teams requires more than tasking algorithms that are capable of adapting online to the dynamic environment. The mechanisms for coordination must be valued and appreciated by the human workers.

Human workers often find identity and security in their roles or jobs in the factory and are used to some autonomy in decision-making. A human worker that is instead tasked by an automated scheduling algorithm may begin to feel that he or she is diminished. Even if the algorithm increases process efficiency at first, there is concern that taking control away from the human workers may alienate them and ultimately

damage the productivity of the human-robot team. The study of human factors can inform the design of effective algorithms for collaborative tasking of humans and robots.

In this work, we present results from a pilot study ($n = 8$) where participants collaborate with a robot on a construction task. In one condition, both the human and robot are tasked by Tercio, the automatic scheduling algorithm. In the second condition, the human worker is provided with a limited set of task allocations from which he/she can choose. We hypothesize that giving the human more control over the decision-making process will increase worker satisfaction, but that doing so will decrease system efficiency in terms of time to complete the task.

We begin in Section 6.2 with a brief review of related work in human factors studies of human-machine systems and task allocation. In Section 6.3 and Section 6.4 we describe our experimental method and we report the results from the human subject experiment. We then discuss the implications, limitations, and lessons learned from the results of our pilot study in Section 6.5. In Section 6.6, we discuss recommendations for a full human subjects experiment. Lastly, we summarize our findings and discuss future work in Section 6.7. For a full description of Tercio, see Chapter 5.

6.2 Background

Developing man-machines systems that are able to leverage the strengths of both humans and their artificial counterparts has been focus of much attention from both human factors engineers as well as researchers in artificial intelligence. Parasuraman has pioneered work examining adaptive automation to regulate operator workload [40]. When the operator is over-tasked, the automation can reduce the burden on the operator by assuming certain tasks. If the human-automation system experiences a period of low workload, the automation can shift more responsibility to the human operator to mitigate possible complacency or a reduction in manual skills [39].

The human-robot interface has long been identified as a major bottleneck in utilizing these robotic systems to their full potential [7]. As a result, we have seen significant

research efforts aimed at easing the use of these systems in the field, including careful design and validation of supervisory and control interfaces [4, 12, 23, 28]. Other, related research efforts have focused on utilizing a human-in-the-loop to improve the quality of task plans and schedules [11, 12, 16]. This is particularly important when the search space is large or if it is not possible to represent all aspects of the environment in the problem formulation.

These prior efforts have focused primarily on utilizing a human-in-the-loop to improve plans and schedules for work performed by other agents. In this work, we are motivated by applications in the manufacturing domain where human workers will be performing physical tasks in coordination with robotic partners. In some cases, the human workers may also be responsible to tracking the progress and tasking the team. We seek to understand how much control human workers should have over the assignment of roles and schedules, when working in teams with robots. While we necessarily want to maximize the efficiency of human-robot teams, we also desire for the human workers to value, accept, and cooperate with the new technology. With this in mind, the following sections of this paper will describe the pilot study we conducted to lend insight into trade-offs among flexibility in decision-making, overall task efficiency, and worker satisfaction.

6.3 Methods

The purpose of this pilot study is to gain insight into how to integrate multi-agent task allocation and scheduling algorithms to improve the efficiency of coordinated human and robotic work. We hypothesize that keeping the human worker in the decision making process by letting him/her decide the task assignments will decrease performance of the system (i.e., increase time to complete the task objective) but will increase the human appreciation of the overall system. We conducted a pilot study to assess the tradeoffs in task efficiency and user satisfaction, depending on whether or not the worker is allowed control over task allocation.

For our pilot study, we consider two experimental conditions:

- Condition 1: A task allocation and scheduling algorithm (i.e., Tercio [22]) specifies the plan.

- Condition 2: The human worker is given a limited set task allocations from which he/she can choose.

6.3.1 Experimental Setup

The task objective given to the human-robot team is to complete two Lego kits, each consisting of seven steps. The parts required to build each of the seven steps for both Lego kits are placed into bins away from the build area. There are two classes of roles for the agents: *builder* or a *part fetcher*. If an agent is a builder, then the agent is responsible for building either one or both of the Lego kits. If an agent is a fetcher, that agent retrieves part bins and transports them to a specified builder. We enforce that the fetcher can only fetch one part bin at a time.

If a robot is assigned to fetch parts, we must have some mechanism of informing the robot when and to whom to fetch the part bin. *A priori* we tuned the temporal constraints of the task network so that the robot would fetch the next part bin for a human builder just before the human finished the previous step. In a future experiment, we want to incorporate a closed-loop feedback mechanism so that the timing of the fetching adapts to when the human is ready for the parts. We discuss this further in Section 6.5.

We use two KUKA Youbots (See Figure 6-1), which are mobile manipulator robots (height 65cm, weight 31.4 kg). To control the robots movement through the space, we implemented a simple close-loop control system. A PhaseSpace motion capture system, which tracks the locations of LEDs that pulse with unique frequencies, provides real-time feedback of the state of the robots (i.e., $\vec{x} = [x, y, z]$ for each robot).

When the experiment begins, the initial assignment of the roles is as follows:

1. The human subject is responsible for completing one of the Lego kits.

2. One Youbot, called "Burra", is responsible for completing the second Lego kit

Figure 6-1: A picture of a KUKA Youbot. Image Courtesy of KUKA Robotics.

3. A second Youbot, called "Kooka", is responsible for fetching the Lego parts for both the subject and the robot Burra.

After "Kooka" fetches the first three part bins for the human subject and "Burra", we simulate a failure of "Kooka" and inform the human subject of the malfunction. Because "Kooka" was responsible for fetching the remaining four part bins for the human subject and "Burra", the assignment of roles must change to complete the build process. We recall that we want to observe the effects of giving the human subject control over the his/her task assignment versus using a autonomous task allocation scheduling algorithm (i.e., Tercio).

For participants assigned to Condition 1, the Tercio algorithm is used to assign roles to the human and robot workers in an optimal manner. In the optimal solution, the participant and the robot Burra work independently, fetching their own part bins and completing their own Lego kits. For participants assigned to Condition 2, the human workers decides the roles of both him/herself and the robot Burra. The two options proposed to the human worker are:

1. The human worker and Burra work independently

2. "Burra" fetches all remaining part sets and the human worker completes the two Lego kits.

After task allocation has been performed, either by Tercio or the human participant, the human and "Burra" complete their respective tasks. The completion of both Lego construction tasks marks the end of the trial.

6.3.2 Data Collection

All experiments were performed at the Interactive Robotics Group (IRG) laboratory at MIT. We tested eight subjects (3 males and 5 females; 25 $\pm$ 2 years ranging between 23-27 years), and four subjects were randomly assigned to each of the two conditions. All subjects were recruited from the MIT graduate student population. The experiment was approved by the Massachusetts Institute of Technology's Committee on the Use of Humans as Experimental Subjects (COUHES) and informed consent was obtained from each subject prior to each experimental session.

Time to complete the task (i.e. finish building both Legos sets) was measured using a stopwatch. The time to finish the build includes the time that the human spent deciding how to reallocate him/herself and "Burra" to the remaining fetching and building tasks after "Kooka" malfunctions. We include this time to better simulate the extra time the decision-making process would take versus the computer-assigned decision. At the end of the trial, each participant completed a questionnaire asking them to rate their level of agreement with the seven statements using a five-point Likert scale from 1 (strongly disagree) to 5 (strongly agree):

1. I was satisfied by the robot system's performance.
2. I would use this robot system next time this set of tasks was to be repeated.
3. The robots collaborated well with me.
4. The robots and I performed the tasks in the least possible time.
5. I was content with the task allocation after the robot malfunctioned.

6. I felt safe and comfortable with the robots.

7. The robots were necessary to the successful completion of the tasks.

6.3.3 Statistical Analysis

Performance data (time to complete the task objective in seconds) were tested for normality using the Kolmogorov-Smirnov test (Condition 1: $p = 0.994$; Condition 2: $p = 0.49$). A one tail t-test for two independent samples with equal variances was used to compare the two conditions. Prior to that, the samples were tested for equal variances using the F test ($p = 0.08$).

Human appreciation of the system (or human satisfaction) data were calculated for each subject as the average of all seven questions in the questionnaire. Thus, every subject had an ordinal score between 1 and 5. The MannWhitney U test (a.k.a Wilcoxon test) were used to compare the two conditions. In all cases, significance was taken at the $\alpha = 0.05$ level. Data is presented as the average standard deviation.

6.4 Results

6.4.1 Performance

The average time for the four participants in Condition 1 was found to be 436 ± 19.1 s. Similarly, the average time for the four participants in Condition 2 was found to be 598.8 ± 47.5 s. Figure 6-2 shows the boxplot of the performance results (time to complete the two Lego kits) across the two conditions (in Condition 1, the algorithm decides assignments; in Condition 2, the subject decides assignments). Time to complete the task was significantly higher in Condition 2 than in Condition 1 ($p < 0.001$).

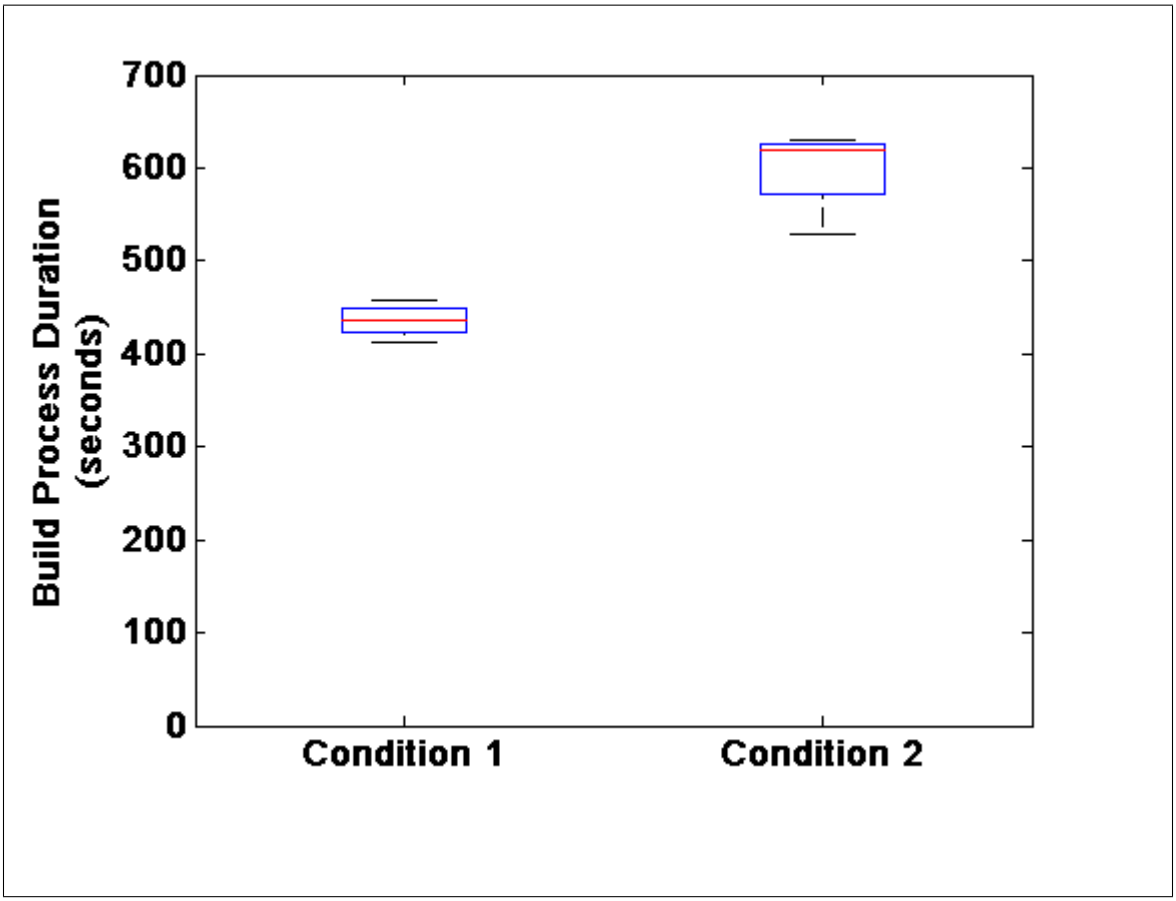

Figure 6-2: Boxplot showing the median, quartile and standard deviations of the performance of the human subjects in both conditions.

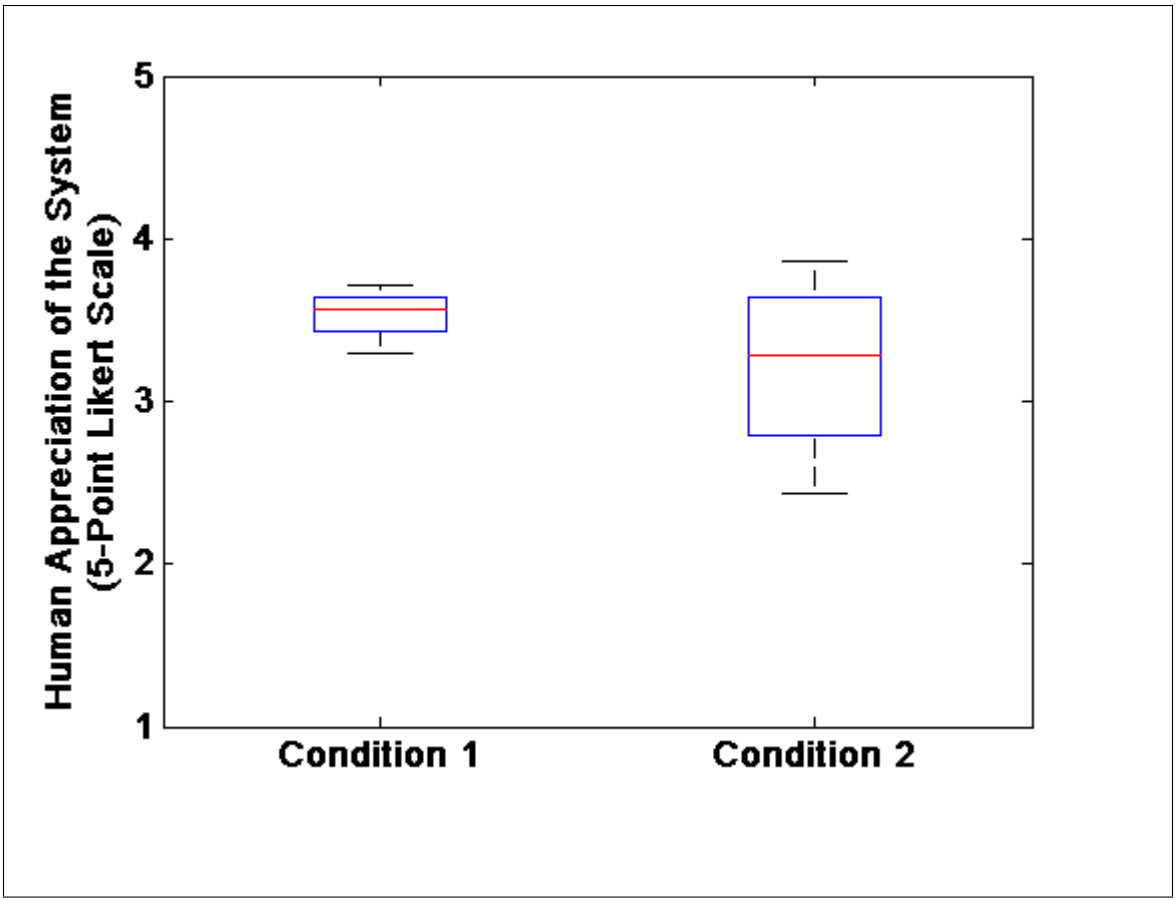

Figure 6-3: Boxplot showing the median, quartile and standard deviations of our measure of human appreciation of the autonomous system based on a five-point Likert scale.

6.4.2 Human Appreciation of the System

The average questionnaire rating for the four participants in Condition 1 was found to be 3.5 ± 0.2. Similarly, the average questionnaire rating for the four subjects in Condition 2 was found to be 3.21 ± 0.6. Figure 6-3 shows the boxplot of the rating results across the two conditions (in Condition 1, the algorithm decides assignments; in Condition 2, the subject decides assignments) The non-parametric Mann-Whitney U test did not find any significant difference between the two conditions ($U = 5 > U_{crit} = 1$). Furthermore, the average rating in Condition 1 is higher than the average rating in Condition 2, which is in disagreement with the initial hypothesis.

6.5 Discussion

6.5.1 Evaluation of Time to Complete the Task

Time to complete the task objective was significantly higher in Condition 2 than Condition 1. All four human-robot teams in Condition 2 needed more time than any of the human-robot teams in Condition 1. Surprisingly, three of the four participants in Condition 2 chose the non-optimal solution (human completes both Legos and the robot "Burra fetches the parts). Only one participant chose the optimal solution and his/her time needed to complete the task was the least in his/her group, although still higher than all four teams in Condition 1. These results indicate that making decisions takes time. Even in the case when workers chose the optimal solution, the time needed to complete the tasks was higher, possibly due to additional time for decision making.

6.5.2 Evaluation Human Appreciation of the System

No statistically significant differences were observed in worker appreciation of the system contrary to what was hypothesized. The three participants from Condition 2 that chose the non-optimal solution present the worst ratings among all participants in the experiment. Interestingly, the participant from Condition 2 that chose the optimal-condition presents the highest rating, even above the ratings from participants in Condition 1. These results lend support to the hypothesis that human satisfaction is most affected by an inherent sense of efficiency while freedom of choice plays a secondary role. In our pilot study, high efficiency with freedom of choice yields the highest satisfaction, while high efficiency without freedom was second. When the participant is given freedom of choice, resulting in low efficiency, human satisfaction appears to decrease drastically.

The lower average scores in Condition 2 could alternatively be attributed to the disruption and difficulty the robot malfunction caused the human participant. From the human's perspective, deciding under time pressure amongst a set of options for

how to resolve the problem may be less preferable than having an automated algorithm resolve the problem. To understand the true factors that affect both performance and human appreciation of the system, we will conduct a full experiment based on the observations from this pilot study.

6.6 Recommendations for a Full Experiment

For the purpose of design of a future experiment to fully investigate this integration, I discuss several threats to internal validity of our pilot study and recommendations to reduce those threats:

1. Our pilot study used a one-group post-test-only design. This experimental design is susceptible because it increases the likelihood that a confounding factor affected the results of the experiment. For example, the worker might have made a mistake in building the Lego, which would likely decrease efficiency and satisfaction. We recommend using a one-group double-pretest post-test design. This design would enable us to account for factors such as statistical regression (i.e. natural variations in Lego-building performance will be less consequential). One threat from this design is testing, where subjects might improve or get to know the system better. To account for these concerns, we should have the worker practice building the Lego at home before the experiment. With practice, the performance will approach a plateau. Lastly, if the subject were assigned to the group where the subject assigns the roles of the workers, we would enforce that the worker performs the assignment of roles for all three trials *a priori*.

2. We chose graduate students from MIT as our population; however, our selection of subjects is quite different than the target population of factory workers. Yet, because we are designing a system that has not yet been used in a factory, MIT students, who have more experience with advanced technology, may be more akin to factory workers who will have worked with this system in the future.

3. Subjects were asked to build Legos that they had never built before and were asked to work with a robot to build Legos, which is a task they had most likely never done before either. As a result, the subjects who had control over the agent-task assignments made educated guesses as to which assignment of roles they would choose. Subject dis/satisfaction may come from whether or not the subject liked the particular role selected rather than from the availability of control or lack thereof. We recommend that subjects are given an opportunity to practice each of the roles of the agents to develop an accurate mental model of the trade-offs for efficiency and satisfaction. We posit that factory workers would also have accurate mental models of the reward and cost for each of the relevant manufacturing roles that the worker might experience.

It is also important to address the threat of experimenter-expectancy effects. We felt it necessary to have one experimenter follow the robots with tethers and a kill-switch to satisfy the concerns of the MIT Internal Review Board. Furthermore, an experimenter was required to place the bins of Lego parts on the manipulator arm of the robot. With such a pronounced involvement of the experimenters, there is an increased likelihood that the subjects may have altered their behavior based upon what they thought we expected from them. Perhaps, if the robots ran without a the visible supervision of the experimenters, the subjects may have been more immersed in the task at hand and, in turn, produce data with higher internal validity. We recommend for the future experiment that the experimenters are physically removed as much as possible.

Additionally, we must consider an important nuisance factor. The pilot study used a scheduling algorithm that assumed uniform performance across all workers. However, workers, whether in the factory or in a laboratory environment, work at different rates. Our goal is to determine how much control a worker should have over a system; however, our subjective and objective measures are likely affected by how well the timing of the actions of the robots coincide with the actions of the humans. For future experiment, we could include this nuisance factor of the coordination quality as one of the factors in the experiment. This technique is known as local control or

blocking and is a common technique. The drawback from this approach is that it would require yet more subjects because we have added a third experimental factor.

6.7 Conclusions and Future Work

We aim to understand how best to integrate robots to work in coordination with a human workforce. In the manufacturing domain, human workers are often provided some flexibility in decision-making for how to execute their work. As a means of integrating robots into human teams, we have developed an algorithm that takes a centralized approach to producing agent-task assignments and schedules. However, concerns exists that humans may not appreciate or may even reject a system that does not allow them enough flexibility in how to do their work. The pilot study we have conducted is a first step towards understanding much control over decision-making a human worker should be provided.

We varied the amount of control that our participants had over the task assignments in the human-robot team. Results from the study supported our first hypothesis; giving the human workers more control decreased temporal efficiency. Our second hypothesis stated that worker appreciation of the technology would benefit from providing the human with some control over the decision-making. This hypothesis was not supported with statistical testing. However, we did find a trend where workers with more control who chose the optimal solution were the most satisfied, and workers with more control who chose the suboptimal solution were the least satisfied.

Pilot studies are designed to provide guidance on how to design a follow-on large scale experiment. While our pilot provides initial results and trends, these results were obtained through a small scale experiment and are not sufficient to provide recommendations on algorithm and interface design. Based on this pilot study, we plan on running a full human subject experiment with a number of changes: 1) the roles of the robots will be redesigned so that they are more highly valued by the human participants (e.g., having the fetching task be more realistic) 2) experimenter interference will be reduced by removing tethered power, and 3) the experiment will

be redesigned to better isolate the dependent variable of worker satisfaction from confounding factors (e.g., uncoordinated timing between the fetching robot and the human).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, I present contributions from my research that advance the capability of large-scale, multi-agent task allocation and scheduling with tightly intercoupled temporospatial constraints. The efficient solution to this problem utilizes a real-time processor scheduling analogy and fast techniques for scheduling and testing the schedulability of these non-preemptive task sets.

I began in Chapter 1 with an introduction to the problem of interest and my approach. In Chapter 2, I present a 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 in polynomial time [33]. This schedulability test leverage a new scheduling priority that restricts the behavior of a self-suspending task set to provide an analytical basis for an informative schedulability test.

Next, I developed a near-optimal, uniprocessor scheduling algorithm in Chapter 3. If my schedulability test determines that a task set can be scheduled under JSF, then the uniprocessor scheduling algorithm can produce a feasible schedule. The main technical innovation of this contribution is a polynomial-time, online consistency test, which I call the Russian Dolls test. In Chapter 4, I extend the uniprocessor scheduling algorithm and Russian Dolls Test to the multiprocessor case, which includes spatial or shared memory resource constraints.

In Chapter 5, I describe a task allocation and scheduling system, named Tercio[1]. The system uses a MILP to determine the task allocation and my multiprocessor scheduling algorithm to perform fast sequencing of the tasks. The algorithm is able to scale up to problem sets that are an order of magnitude larger than prior state-of-the-art solvers. Lastly, I present a pilot study in Chapter 6 to determine how best to integrate Tercio into the manufacturing environment from a human-centered point of view.

7.2 Future Work

There are two primary areas of future work. First, the multiprocessor scheduling algorithm (Chapter 4) that is embedded within Tercio must be extended to include more general temporal constraints. Second, we need to develop of a full-factory scheduling algorithm based on the Tercio (Chapter 5).

7.2.1 Extending Tercio to Allow More General Temporal Constraints

The fast task sequencer within Tercio is based on an augmented self-suspending task model, which I present in Chapter 4. While the augmented model includes intra-task and subtask deadlines, I seek to generalize the breadth of temporal constraints relating tasks and subtasks. The current self-suspending task model enforces that subtask τ_i^j precedes τ_i^{j+1} by duration E_i^j; however, τ_x^y and τ_i^j are unordered for $x \neq i$. In future work, I will incorporate a mechanism for relating τ_x^y and τ_i^j through explicit temporal constraints.

I also seek to extend Tercio to be able to solve hierarchical problems, where there are a set of processes, each comprised of a set of tasks. This hierarchical structure provides further generality and provides an intuitive method for a human operator to specify the constraints of the manufacturing task set. The premise of the extension

[1]Joint work with Ronald Wilcox

is that schedules for each process would be solved at the task level. Then, we would solve the process-level problem based on the schedules generated the tasks within each process.

7.2.2 Full Factory Scheduling

Tercio was designed as a single-cell scheduling algorithm, here a single cell consists of a set of workers working in close proximity with a specific, common goal (e.g., one location along an assembly pulse line). I seek a near-optimal system that can schedule multiple cells in real-time. However, there is an inherent trade-off between global optimality (across multiple cells) and local optimality (within a single cell). Because multiple cells are readily modeled as distributed processes, Tercio could serve as the basis for scheduling intra-cell activity. To perform multi-cell task allocation in response to a significant dynamic disturbance e.g., a robot from one cell brakes and is offline until repaired), the multi-cell algorithm could conduct a trade study evaluating the quality of Tercio-generated schedules based on which workers are assigned to which cells. In this way, robots could be shifted from one cell to another if necessary to satisfy temporal constraints. However, the multi-cell system would need to ensure that the cost to local optimality did not suffer to such an extent as to violate any minimum production rates within a cell.

Bibliography

[1] Gurobi optimizer version 5.0, 2012.

[2] Yasmina Abdeddaïm and Damien Masson. Scheduling self-suspending periodic real-time tasks using model checking. In *Proc. Real-Time Systems Symposium (RTSS)*, 2011.

[3] Amr Ahmed, Abhilash Patel, Tom Brown, MyungJoo Ham, Myeong-Wuk Jang, and Gul Agha. Task assignment for a physical agent team via a dynamic forward/reverse auction mechanism. In *Proc. International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, 2005.

[4] Michael J. Barnes, Jessie Y.C. Chen, Florian Jentsch, and Elizabeth S. Redden. Designing effective soldier-robot teams in complex environments: training, interfaces, and individual differences. In *EPCE*, pages 484–493. Springer, 2011.

[5] Dimitris Bertsimas and Robert Weismantel. *Optimization over Integers*. Dynamic Ideas, 2005.

[6] Luc Brunet, Han-Lim Choi, and Jonathan P. How. Consensus-based auction approaches for decentralized task assignment. In *AIAA Guidance, Navigation, and Control Conference (GNC)*, Honolulu, HI, 2008 (AIAA-2008-6839).

[7] Jennifer Casper and Robin Roberson Murphy. Human-robot interaction in rescue robotics. *IEEE Transaction on Systems, Man, and Cybernetics (SMCS)*, 34(2):138–153, 2004.

[8] David A. Castañón and Cynara Wu. Distributed algorithms for dynamic reassignment. In *IEEE Conference on Decision and Control (CDC)*, volume 1, pages 13–18, December 2003.

[9] Elkin Castro and Sanja Petrovic. Combined mathematical programming and heuristics for a radiotherapy pre-treatment scheduling problem. 15(3):333–346, 2012.

[10] Jiaqiong Chen and Ronald G. Askin. Project selection, scheduling and resource allocation with time dependent returns. 193:23–34, 2009.

[11] Andrew S Clare, Mary (Missy) L. Cummings, Jonathan P. How, Andrew K. Whitten, and Olivier Toupet. Operator objective function guidance for a real-time unmanned vehicle scheduling algorithm. 9:161–173, 2012.

[12] Mary (Missy) L. Cummings, Amy S. Brzezinski, and John D. Lee. Operator performance and intelligent aiding in unmanned aerial vehicle scheduling. *IEEE Intelligent Systems*, 22(2):52–59, March 2007.

[13] Jess Willard Curtis and Robert Murphey. Simultaneous area search and task assignment for a team of cooperative agents. In *IEEE GNC*, 2003.

[14] Rina Dechter, Italy Meiri, and Judea Pearl. Temporal constraint networks. *AI*, 49(1), 1991.

[15] UmaMaheswari C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Euromicro Technical Committee on Real-Time Systems*, 2003.

[16] Edmund H. Durfee, James C. Boerkoel Jr., and Jason Sleight. Using hybrid scheduling for the semi-autonomous formation of expert teams. *Future Generation Computer Systems*, July 2013.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[18] Laurent George, Paul Muhlethaler, and Nicolas Rivierre. Optimality and non-preemptive real-time scheduling revisited. In *Proc. INRIA*, April 1995.

[19] Matthew C. Gombolay and Julie A. Shah. Multiprocessor scheduler for task sets with well-formed precedence relations, temporal deadlines, and wait constraints. In *Proc. AIAA Infotech@Aerospace*, 2012.

[20] Matthew C. Gombolay and Julie A. Shah. Uniprocessor schedulability test for hard, non-preemptive, self-suspending task sets with multiple self-suspensions per task. In *Proc. Real-Time Systems Symposiums (RTSS) (Under Review)*, 2013.

[21] Matthew C. Gombolay, Ronald J. Wilcox, Ana Diaz Artiles, Fei Yu, and Julie A. Shah. Towards succesful coordination of human and robotic work using automated scheduling tools: An initial pilot study. In *Proc. Robotics: Science and Systems (RSS) Human-Robot Collaboration Workshop (HRC)*, 2013.

[22] Matthew C. Gombolay, Ronald J. Wilcox, and Julie A. Shah. Fast scheduling of multi-robot teams with temporospatial constrints. In *Proc. Robots: Science and Systems (RSS)*, 2013.

[23] Michael A. Goodrich, Bryan S. Morse, Cameron Engh, Joseph L. Cooper, and Julie A. Adams. Towards using UAVs in wilderness search and rescue: Lessons

from field trials. *Interaction Studies, Special Issue on Robots in the Wild: Exploring Human-Robot Interaction in Naturalistic Environments*, 10(3):453–478, 2009.

[24] Michael González Harbour and José C. Palencia. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proc. Real-Time Systems Symposium (RTSS)*, 2003.

[25] John N. Hooker. A hybrid method for planning and scheduling. In *Proc. Carnegie Mellon University Research Showcase*, 2004.

[26] John N. Hooker. An improved hybrid MILP/CP algorithm framework for the job-shop scheduling. In *Proc. IEEE International Conference on Automation and Logistics*, 2009.

[27] Vipul Jain and Ignacio E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. 13, 2001.

[28] Henry L. Jones, Stephen M. Rock, Dennis Burns, and Steve Morris. Autonomous robots in SWAT applications: Research, design, and operations challenges. *AUVSI*, 2002.

[29] In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *Proc. Conference on Real-time Computing Systems and Applications*, 1995.

[30] Alex Kushleyev, Daniel Mellinger, and Vijay Kumar. Towards a swarm of agile micro quadrotors. *Robotics: Science and Systems (RSS)*, July 2012.

[31] Philippe Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, February 2003.

[32] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Open problems in scheduling self-suspending tasks. In *Proc. Real-Time Scheduling Open Problems Seminar*, 2010.

[33] Karthik Lakshmanan and Ragunathan (Raj) Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[34] Cong Liu and James H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proc. Real-Time Systems Symposium (RTSS)*, 2009.

[35] Cong Liu and James H. Anderson. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems. In *Proc. Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2010.

[36] Cong Liu and James H. Anderson. An O(m) analysis technique for supporting real-time self-suspending task systems. In *Proc. Real-Time Systems Symposium (RTSS)*, 2012.

[37] Jane Liu. *Real-Time Systems*. Prentice Hall, 2000.

[38] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. Principles of Knowledge Representation and Reasoning (KR&R)*, 1998.

[39] Raja Parasuraman and Dietrich H. Manzey. Complacency and bias in human use of automation: An attentional integration. 52:381–410, 2010.

[40] Raja Parasuraman, Mouloua Mustapha, and Brian Hilburn. Adaptive aiding and adpative task allocation enchance human-machine systems.

[41] Ragunathan (Raj) Rajkumar. Dealing with self-suspending period tasks. Technical report, IBM Thomas J. Watson Research Center, 1991.

[42] Wei Ren, Randal W. Beard, and Timothy W. McLain. Coordination variables, coordination functions, and cooperative timing missions. *AIAA Journal on Guidance, Control, and Dynamics*, 28(1):150–161, 2005.

[43] Pascal Richard. On the complexity of scheduling real-time tasks with self-suspensions on one processor. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[44] Frédéric Ridouard and Pascal Richard. Negative results for scheduling independent hard real-time tasks with self-suspensions. 2006.

[45] Sanem Sariel and Tucker Balch. Real time auction based allocation of tasks for multi-robot exploration problem in dynamic environments. In *Proc. AIAA Workshop on Integrating Planning into Scheduling*, 2005.

[46] David E. Smith, Feremy Frank, and Ari Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15, 2000.

[47] Jun Sun and Jane Liu. Synchronization protocols in distributed real-time systems. In *Proc. International Conference on Distributed Computing Systems*, 1996.

[48] Steven J. Rasmussen Tal Shima and Phillip Chandler. UAV team decision and control using efficient collaborative estimation. In *Proc. American Control Conference (ACC)*, volume 6, pages 4107–4112, June 2005.

[49] Wei Tan. Integration of process planning and scheduling - a review. 11:51–63, 2000.

[50] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. In *Proc. Microprocessing and Microprogramming*, volume 40, 1994.

[51] Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of O(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, October 2005.

[52] Ronald J. Wilcox, Stefanos Nikolaidis, and Julie A. Shah. Optimization of temporal dynamics for adaptive human-robot interaction in assembly manufacturing. In *Proc. Robotics: Science and Systems (RSS)*, 2012.