SMT-based Scheduling Methodology for Real-Time Systems

by

Zhuo Cheng

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Yuto Lim

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2017

Abstract

Real-time systems are playing an important role in our society. In the last two decades, there has been a dramatic rise in the number of real-time systems being used in our daily lives and in industry production. Representative examples include vehicle and flight control, chemical plant control, telecommunications, and multimedia systems. These systems all make use of real-time technologies.

The most important attribute that sets real-time systems apart from other systems is that the correctness of systems depends not only on the computed results but also on the time at which results are produced. In other words, a task in a real-time system is required to be completed before a specific time instant which is called *deadlines*. This sensitivity to timing is central feature of system behaviors. To satisfy this requirement, tasks need to be allocated sufficient resources (e.g., processor) so as to meet their deadlines (i.e., to be completed before their deadlines). Scientific community has made great efforts in developing scheduling polices for properly allocating resources to tasks. This field of study is referred to as *real-time scheduling*.

With decades of efforts, real-time scheduling on uniprocessor systems can be viewed as relatively mature. But there still remain many problems. For example, for a soft or firm real-time system, when the system is under overload condition, some tasks will miss deadlines. It is important to minimize the degree of system performance degradation caused by the missed deadline tasks. For this problem, the performance of existing scheduling algorithms is not satisfactory.

Compared to uniprocessor systems, real-time scheduling on multiprocessor systems is far from well-studied. Even for the simpler problem of scheduling identical multiprocessor systems, there still remain many problems. For the harder problem of scheduling heterogeneous multiprocessor systems, where we have to face an awkward reality that considering a relatively practical application scenario, there is still no solution to efficiently schedule heterogeneous multiprocessor systems. This becomes unfortunate since current progress in developing heterogeneous multiprocessor systems is a long way ahead of research efforts

to determine the best scheduling policies.

These situations motive this research. The vision of this research is to help developers easily and efficiently design scheduling for real-time systems with low cost. To approach this vision, a Real-time scheduling methodology based on Satisfiability Modulo Theories (RSMT) is proposed. In RSMT, the problem of scheduling is treated as a *satisfiability problem*. The key work is to formalize the satisfiability problem using first-order logical formulas. Through formalization, a *SAT model* is constructed to represent the scheduling problem. This SAT model is a set of first-order logic formulas (within linear arithmetic in the formulas) which express all the *scheduling constraints* that a desired schedule should satisfy. After the SAT model is constructed, a SMT (satisfiability modulo theories) solver (e.g., Z3, Yices) is employed to solve the formalized problem. A desired schedule can be generated based on a *solution model* returned by the underlying SMT solver.

After RSMT is proposed, it is first applied to uniprocessor time-driven systems to solve the overload problem. Then, RSMT is exploited to design scheduling for multiprocessor time-driven systems. Heterogeneous real-time systems have been considered. At last, in order to apply RSMT to design scheduling for event-driven systems, a method of combining RSMT and online scheduling algorithm is given. Through these studies, RSMT shows capabilities to be applied to design scheduling for various kinds of scheduling targets and systems, from uniprocessor to multiprocessor, from time-driven to event-driven. In addition, many practical requirements that are considered in real-time scheduling domain can be dealt with, e.g., task dependency relation, tasks with different degrees of importance, task preemption, and task migration. Benefit from these capabilities, RSMT leads us one step closer to the vision. To the best of my knowledge, it is the first time systematically introducing SMT to solve a series of problems covering a wide scope in real-time scheduling domain.

keywords: real-time systems, satisfiability modulo theories, real-time scheduling, overload problem, heterogeneous multiprocessor systems

Acknowledgement

I would like to first express my great gratitude to my supervisor, professor Yuto Lim. I am delighted and grateful to be able to work under his excellent supervision. He provided me a lot of useful suggestions, insightful criticism and professional guidance, without his consistent and illuminating instruction, this dissertation could not be presented in current form.

I wish to express my sincere gratitude to my second supervisor, professor Yasuo Tan. His great foresight ensured the research moving forward along the right direction. He gave me a lot of valuable comments and instructive advices to overcome my research problems, from whom I benefited a lot of knowledge and experience.

I should give my hearty thanks to professor Shaoying Liu. It is so luck to conduct minor research under his supervision. He offered me great help on my whole study. With his wealth of knowledge, rich experience, and professional spirit, he taught me a lot.

Moreover, sincerely thanks to professor Shinoda and professor Tanaka. Their constructive comments gave me new perspectives and motivated me to think more deeply and widely, which makes the dissertation better.

In addition, I would like to thank my friends, Fan Yang, Dawei Xu, Haitao Zhang. They kindly gave me a hand when I was in frustration or depression. Their encouragement and unwavering support sustained me through the hard time. Special thanks to Haitao Zhang who puts considerable time and efforts in my research starting from the beginning. Lots of ideas were driven from the discussions with him. It is quite lucky to have a friend like him.

Last but not the least, thanks my parents and other family members. Their great love and fully understanding throughout all the years is my strongest motivation to go on my study and pursue my life goals.

Contents

List of Figures

List of Symbols

t	system time instant
$\mathcal{T}$	set of real-time tasks
$\tau_i \in \mathcal{T}$	real-time task, i is index of the task
r_i	request time instant of τ_i
e_i	required execution time of τ_i
re_i	remaining execution time of τ_i
d_i	deadline of τ_i
s_i	start execution time of τ_i
$f_{i,j}$	j-th indivisible fragment of τ_i
$f_{i,e}$	last indivisible fragment of τ_i
$s_{i,j}$	start execution time of $f_{i,j}$
$e_{i,j}$	required execution time of $f_{i,j}$
δ	network precision
$\mathcal{F}$	set of functions in a real-time system
$\mathcal{FH} \subseteq \mathcal{F}$	set of functions with firm deadlines
$\mathcal{FS} \subseteq \mathcal{F}$	set of functions with soft deadlines
$F_i \in \mathcal{F}$	function in a real-time system, i is the index of the function
rf_i	triggered time instant of function F_i
df_i	deadline of function F_i
v_i	obtained value by completing function F_i before deadline
$T_i \subseteq \mathcal{T}$	set of tasks corresponding to function F_i

c_i	computation cost of task τ_i (for multiprocessor systems)
m_i	migration cost of τ_i (for multiprocessor systems)
τs_i	start task of task poset $(T_i, \prec)$
τe_i	end task of task poset $(T_i, \prec)$
$\mathcal{P}$	set of processors
$p_a \in \mathcal{P}$	processor, where a is the index of the processor
sp_a	speed of processor p_a
$TS_a \subseteq \mathcal{T}$	task set that can be completed by processor p_a
$\mathcal{N} \subseteq \mathcal{P} \times \mathcal{P}$	set of network channels
$n_{a \to b} \in \mathcal{N}$	network channel from processor p_a to p_b
$ns_{a \to b}$	speed of $n_{a \to b}$
$\mathcal{PF}$	set of periodic functions
PF_i	periodic function, i is index of the function
PF_i^j	j-th instant of periodic function PF_i
$\mathcal{SF}$	set of sporadic functions
SF_i	sporadic function, i is index of the function
SF_i^j	j-th instant of sporadic function SF_i
rd_i	relative deadline of sporadic or periodic function
p_i	period of sporadic or periodic function

Chapter 1

Introduction

1.1 Research Background

Real-time systems are playing an important role in our society. In the last two decades, there has been a dramatic rise in the number of real-time systems being used in our daily lives and in industry production. Representative examples include vehicle and flight control, chemical plant control, telecommunications, and multimedia systems. These systems all make use of real-time technologies [1].

The most important attribute that sets real-time systems apart from other systems is that the correctness of systems depends not only on the computed results but also on the time at which results are produced [1]. In other words, a task in the system is required to be completed before a specific time instant which is called *deadline*. This sensitivity to timing is central feature of system behaviors [38]. To satisfy this requirement, tasks need to be allocated sufficient resources (e.g., processor) so as to meet their deadlines. Scientific community has made great efforts in developing scheduling polices for properly allocating resources to tasks. This field of study is referred to as *real-time scheduling*.

1.1.1 Why Scheduling

The well-known Moore's law [72] tells that the number of transistors in a dense integrated circuit doubles approximately every two years. This prediction has been proved accurate for several decades and describes a driving force of technological and social change, productivity, and economic growth [73]. With this development rate, currently the price

of processor is quite cheap compared to 1970s in which the first real-time scheduling algorithm was proposed [14]. Based on this comparison, a question raises: considering the cheap price of processor, since we can afford many processors for our systems, it is doubtful that research on real-time scheduling is still necessary.

This question comes from our intuitive impression, but if we think deeper, we can find that there are lots of limitations and disadvantages to install more processors. For example, in space-sensitive industries (e.g., auto industry), the space of a system is very limited. In such a system, it is impractical to install more processors. Moreover, more processors results in more energy consumption. Even if we ignore the additional cost due to installing more processors, in the case of the same performance, a system consuming less energy can surely outstrips the one consuming more. More seriously, more energy consumption means more heat emission. From the observation that many data centers located in extremely cold environments, we can know that the heat emission is quite an important consideration, even a bottleneck, when designing such systems. Considering these limitations and disadvantages, how to effectively utilize processor resources becomes particularly important. Research on scheduling just aims at solving this problem.

1.1.2 Classification of Real-Time Systems

1.1.2.1 Based on timing requirements

Although all the real-time systems have the timing requirements, the stringency level of such requirements can be different [3, 28, 29], which depends on their applied scenarios.

a. Hard real-time systems

Some systems cannot tolerate any task missing deadline, as a missed deadline task can result in a catastrophe, such as loss of human lives. This kind of systems is referred to as *hard real-time systems* [14]. Some typical examples are cruise control systems, flight control systems.

Some other systems do not have such strict timing requirements. They allow tasks missing deadlines occasionally, since a missed deadline task will just decrease quality-of-service (QoS) rather than causing a catastrophe. For this kind of systems, based on

usefulness of a missed deadline task, systems can be further divided into three categories.

b. Firm real-time systems

For a firm real-time system, a missed deadline task is useless to the system. Such a task is called *firm deadline task*, in other words, we say this task has *firm deadline* [74]. A typical example is weather forecast system.

c. Soft real-time systems

For a soft real-time system, a missed deadline task may still be useful to the system. Such a task is called *soft deadline task*, in other words, we say this task has *soft deadline* [14]. A typical example is multimedia system.

d. Mixed critical real-time systems

A relatively complex real-time system may contain both firm deadline tasks and soft deadline tasks. For example, an integrated management system equipped on a vehicle can manage both cruise control system (comprising firm deadline tasks) and sound system (comprising soft deadline tasks). This kind of systems is referred to as *mixed critical real-time systems* [75].

This research considers all the kinds of systems mentioned above.

1.1.2.2 Based on processors

Another classification criterion is based on equipped processors.

a. Uniprocessor real-time systems

A simple real-time system can have only one processor[1]. This kind of systems is referred to as *uniprocessor real-time systems*[69].

[1]In this dissertation, as the specific architecture of a computing unit is not considered, the term "processor" is used to refer to all kinds of computing units.

b. Multiprocessor real-time systems

With respect to uniprocessor real-time systems, real-time systems which contain multiple processors are referred to as *multiprocessor real-time systems*. Based on types of the equipped processors, the multiprocessor real-time systems can be further divided into three categories [13].

- **Heterogeneous multiprocessor real-time systems**. The processors equipped in the systems are with different types. They can have different computing speeds, which means for the same task, different processors need different time to complete the task. Moreover, not all tasks may be able to be executed on all the processors.
- **Uniform multiprocessor real-time systems**. The processors equipped in the systems are the same type but with different computing speeds. With respect to heterogeneous multiprocessor real-time systems, all the tasks can be executed on all the processors.
- **Identical multiprocessor real-time systems**. The processors equipped in the systems are exactly the same, and of course, all the tasks can be executed on all the processors.

This research considers all the kinds of systems mentioned above.

1.1.2.3 Based on task triggered paradigms

a. Event-driven systems

In an event-driven system, tasks are triggered as a consequence of the occurrence of an event. Such an event is a change of state in a real-time object which has to be handled by the system [76, 77]. For example, in a cruise control system, when drive presses a button in the control lever, an event will be issued. Correspondingly, to response to this event, some specified tasks are triggered.

As the events are usually initiated by the outside environment in which the system operates (e.g., operations of users), the time at which the events are initiated is highly randomized. Due to this randomness, in many implementations of real-time systems,

signaling of the events is realized by interrupt mechanism, which brings the occurrence of an event to attention of the processor.

Although initiated times of the events are random, other information, such as which tasks should be initiated to response to a particular event, are predefined at the time when designing the system.

b. Time-driven systems

In a time-driven system, tasks are driven by system clocks which could be continuous as in physical time, or discrete as if the clock were a metronome [77]. The time points at which tasks will be triggered are predetermined. This driven mechanism makes the time-driven systems more controllable than event-driven systems due to more determinism.

One drawback of time-driven system is lack on flexibility [78]. All the parameters of tasks must be known in advance. Compared to time-driven systems, the main advantage of event-driven systems is their capability to fast react to asynchronous external events which are randomly triggered at system run-time [79]. Therefore, event-driven systems show better real-time performance.

RSMT first focuses on time-driven systems. Through combining with online scheduling algorithm, RSMT shows capability to design scheduling for event-driven systems.

1.1.3 Classification of Real-Time Scheduling

Generally speaking, scheduling (conducted by a scheduler) is the process of allocating resources (especially, processors) to tasks. The allocating result (i.e., scheduling result) is called *schedule* (noun). Research on real-time scheduling is to study how to properly allocate resources to tasks, so that, system timing requirements can be guaranteed.

1.1.3.1 Based on way to schedule tasks

In real-time scheduling, there are at least two different ways to schedule (verb) the tasks [11].

a. Offline scheduling

In offline scheduling, a scheduling table is generated before system run-time which specifies all the information needed to schedule tasks, e.g., start execution time of all the tasks contained in the system. Later, at system run-time, this scheduling table is used by the scheduler to schedule tasks. As in this scheduling paradigm, tasks are scheduled before system run-time, scheduling methods adopting this scheduling paradigm are usually applied to time-driven systems. Note that, offline scheduling is also called *table-driven scheduling* [69].

b. Online scheduling

In online scheduling, a number referring to as priority is assigned to each task at system run-time usually based on task parameters. For example, the priority assignment can be based on task deadline, say a task with earlier deadline has a higher priority. The task with the highest priority will be executed first. As in this scheduling paradigm, tasks are scheduled at system run-time, scheduling methods adopting this scheduling paradigm can be applied to both time-driven and event-driven systems. Note that, online scheduling is also called *priority-driven scheduling* [69].

RSMT focuses on offline scheduling, but a method of combining RSMT and online scheduling is given.

1.1.3.2 Based on task migration

For multiprocessor systems, a criterion to categorize the scheduling techniques is based on whether a task is allowed to migrate from one processor to another or not [69].

a. Non-migrative scheduling

In non-migrative scheduling, tasks migrating [2] among processors is not allowed. That is, if task τ_i needs the computed results of τ_j, task τ_i can only be executed on the processor on which task τ_j has been executed.

b. Intra-migrative scheduling

In intra-migrative scheduling, tasks can migrate among processors with the same type. For an identical or homogeneous multiprocessor systems, this means that tasks can migrate among all the processors, since processors are the same type in these systems.

c. Fully-migrative scheduling

In fully-migrative scheduling, tasks cannot only migrate among processors with the same type, but also among processors with different types.

For identical or homogeneous multiprocessor systems, fully-migrative scheduling is the same as intra-migrative scheduling. But for heterogeneous multiprocessor systems, these two types of scheduling are different, since different types of processors are equipped in a heterogeneous multiprocessor system. Note that, in a heterogeneous multiprocessor system, since not all tasks are able to be executed on all the processors, a task cannot migrate to the processor on which it cannot be executed.

This research considers all the kinds of scheduling methods mentioned above.

1.1.3.3 Based on task preemption

Task preemption is to denote the phenomenon that a task running (i.e., being executed) on a processor is forced by scheduler to relinquish the processor before it completes execution [13].

[2] Strictly speaking, the data of the computed results of the task (or the signal issued by the task) migrates among processors. More details will be explained in Chapter 4. For conciseness, tasks migrating is used to refer to the meaning mentioned above hereafter.

a. Preemptive scheduling

In preemptive scheduling, a running task can be preempted by another task.

b. Non-preemptive scheduling

In non-preemptive scheduling, task preemption is not allowed.

This research considers all the kinds of scheduling methods mentioned above.

1.1.3.4 Based on other practical requirements

Besides task preemption and task migration, there are still many other practical requirements considered in real-time scheduling domain. Similar to the classification based on task preemption, we can classify the scheduling methods based on if they can support the corresponding requirements. Here are some practical requirements: *task dependency relation*, *tasks with different degrees of importance*.

Task dependency relation restricts the execution order of tasks. A task depending on another one can start to run only after the depended task has been completed. Tasks have such dependency relation may be because one task needs the computed results of the others. Such situation widely exists in practical applications.

Tasks contained in a system may not be equally important. This situation also widely exists in practical applications. Scheduling methods not supporting the requirement, *tasks with different degrees of importance*, cannot be applied to these applications.

This research considers both task dependency relation and tasks with different degrees of importance.

1.1.4 Background on Real-Time Scheduling

Scheduling for uniprocessor real-time systems

Research on uniprocessor real-time scheduling can trace back to the late 1960s and early 1970s [14].

The paper by Liu and Layland [14] is regarded as the foundational work in real-time scheduling theory. The paper addressed the problem of multiprogramming scheduling on a uniprocessor real-time system. In the paper, two scheduling algorithms: fixed priority scheduling algorithm and deadline driven scheduling algorithm were proposed[3]. Some assumptions were applied to the proposed scheduling algorithms, e.g., *i*) The requests for all tasks for which hard deadlines exist are periodic; *ii*) Tasks are independent; *iii*) Run-time for each task is constant for that task and does not vary with time. Two theoretical bounds were adopted: *i*) an optimal fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets; *ii*) full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. To obtain theoretical bounds, some proof methods like interchanging task priorities were introduced. These outcomes have heavily influenced the course of research on real-time scheduling for decades.

Since the seminal paper, in the 1980s and 1990s, significant research effort has been made (e.g., [2, 4, 10, 12, 17]). Authors in [15, 16] provide historical accounts of the most important advances in the field of uniprocessor scheduling during those decades. For example, in 1982, Leung [17] considered fixed-priority scheduling for sets of tasks that have deadlines less than their periods; Authors in [5] provided analysis techniques for systems where tasks can communicate with each other.

Today, real-time scheduling on uniprocessor systems can be viewed as relatively mature [13]. A large number of key results have been obtained and documented in textbooks (e.g., [29, 40, 41]). But there is still significant scope for further research. For example, for a soft or firm real-time system, when the system is under overload condition, some tasks will miss deadlines. It is important to minimize the degree of system performance degradation caused by the missed deadline tasks. For this problem, the performance of existing scheduling algorithms is not satisfactory.

Scheduling for multiprocessor real-time systems

Although multiprocessor real-time scheduling theory also has its origins in the late 1960s and early 1970s [14], compared to uniprocessor systems, real-time scheduling on multi-

[3]As the system model studied in [14] is different as the model studied in this dissertation, some terms like "fixed priority" is not applied to the model studied in this dissertation.

processor systems is far from well-studied. Liu in [14] noted that real-time scheduling on multiprocessor systems is intrinsically a much more difficult problem than on uniprocessor systems: "*Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.*"

Most of existing works deal with the simpler problem, scheduling identical or uniform multiprocessor real-time systems, e.g., [62, 63, 64, 65, 66, 67]. But even for this simpler problem, there still remain many problems. For example, in a comprehensive survey [13], some open issues are emphasized: *i)* limits on processor utilization; *ii)* ineffective schedulability tests; *iii)* consideration of overheads; *iv)* limited task models for multiprocessor systems; and *v)* limited policies for access to shared resources.

For the harder problem of scheduling heterogeneous multiprocessor, some works have been conducted, e.g., [56, 57, 58, 59, 60, 61, 61, 68]. However, all these works consider the non-migrative scheduling. That is, tasks are not allowed to migrative among processors, which is a simpler case compared to full- and intra-migrative scheduling which allow task migrating among processors through network channels. Few works consider full- or intra-migrative scheduling for heterogeneous multiprocessor systems.

This results in an awkward reality that considering a relatively practical application scenario (e.g., considering task dependency relation), there is still no solution to efficiently schedule heterogeneous multiprocessor systems. This becomes unfortunate since current progress in developing heterogeneous multiprocessor systems is a long way ahead of research efforts to determine the best scheduling policies. Many chip makers have already offered chips having different types of processors, both for desktops and embedded devices [42], such as AMD Inc. [43], Apple Inc. [44], Intel Corporation [45, 46], Nvidia Inc. [47], Qualcomm Inc. [48], Samsung Inc. [49], ST Ericsson [50], and Texas Instruments [51]. Developing real-time systems based on these produces can results in systems that are heavily over specified and expensive due to lack of necessary theoretical underpinnings.

1.1.5 Shortcomings of Conventional Design Method

Conventional design method for scheduling real-time systems is to design scheduling algorithms which is adopted by most of existing works (e.g., [1, 3, 13, 14, 52, 54]). This design method has its intrinsic shortcomings, which inevitably hinders the scheduling problems from being solved. Its major shortcomings are as follows.

- It is difficult, even impossible, to adapt some existing scheduling algorithms to different systems or to different scheduling targets.
- When we design a scheduling algorithm for a newly developed system, it is difficult to benefit from the existing scheduling algorithms, which usually results in a high cost.
- For many existing scheduling algorithms, to enhance them to support some practical requirements (e.g., considering task dependency relation) is difficult, even impossible.

Let's use the scheduling algorithm GS (greedy scheduling) proposed in [52] as an example. GS dedicates to solving the overload problem. It can only be applied to uniprocessor systems for its specific scheduling target, maximizing total number of successfully completed tasks. The key idea of GS is based on the idea of greedy algorithm. Through analyzing characteristics of the uniprocessor systems and scheduling target, GS gives its processing flow. Such processing flow is customized for the uniprocessor systems and the specific scheduling target. Moreover, GS assumes that tasks are independent with each other. When we consider task dependency relation which widely exists in practical applications, GS cannot be applied.

These shortcomings hinder developers to easily and efficiently design scheduling for real-time systems, which usually results in a high cost. Mainly because of this, many problems exist in real-time scheduling domain. Based on these observations, I conduct this research.

1.2 Research Vision and Purpose

The vision of this research is to help developers easily and efficiently design scheduling for real-time systems with low cost.

To approach this vision, a Real-time scheduling methodology based on Satisfiability Modulo Theories (RSMT) is proposed. First, RSMT is applied to uniprocessor time-driven systems in which it focuses on solving the overload problem. Then, RSMT is exploited to design scheduling for multiprocessor time-driven systems. Heterogeneous real-time systems have been considered. At last, in order to apply RSMT to design scheduling for event-driven systems, a method of combining RSMT and online scheduling algorithm is given.

Through these studies, RSMT shows capabilities to design scheduling for various kinds of scheduling targets and systems, from uniprocessor to multiprocessor, from time-driven to event-driven. In addition, many practical requirements that are considered in real-time scheduling domain can be dealt with, e.g., task dependency relation, tasks with different degrees of importance, task preemption, and task migration (see details in Chapter 3, 4, and 5). Benefit from these capabilities, RSMT leads us one step closer to the vision.

1.3 Contribution

- **RSMT provides a unified design strategy for scheduling real-time systems**. In RSMT, scheduling problem is treated as a satisfiability problem which is formalized by first-order logical formulas. Such formalization provides a firm theoretic foundation to solve the scheduling problems (see details in Chapter 2, 3, and 4).

- **RSMT can reduce cost of designing scheduling for real-time systems.** When adapting a system to other scheduling targets, system constraints defined in the SAT model can be totally reused, only little modification on target constraints needs to make, which means the scheduling constraints defined for an existing system can greatly benefit the design for a newly developed system (see details in Chapter 3 and 4).

- **RSMT can help developers easily and efficiently design scheduling for real-time systems.** Based on the design guidelines given in this research, developers can directly apply RSMT to design scheduling for various kinds of systems and scheduling targets, and to handle many practical requirements (see details in Chapter 3, 4, and 5). Moreover, for a newly encountered scheduling problem, through modifying or adding scheduling constraints given in Chapter 3 and 4, developers may also apply RSMT to solve the problem (see details in section 6.3).

To the best of my knowledge, it is the first time systematically introducing SMT to solve a series of problems covering a wide scope in real-time scheduling domain.

1.4 Organization of Dissertation

Chapter 2. The framework of RSMT is given. To illustrate the usage of the framework, an example of designing scheduling for a simple scenario is shown.

Chapter 3. RSMT is applied to address the overload problem. The target system is time-driven uniprocessor real-time systems. Various kinds of scheduling targets and requirements are considered.

Chapter 4. RSMT is applied to design scheduling for time-driven multiprocessor real-time systems. Since the heterogeneous systems are considered, it means RSMT can also be applied to identical and uniform systems as scheduling problems for such systems can be treated as special cases of scheduling heterogeneous multiprocessor systems.

Chapter 5. Since RSMT performs in offline paradigm, it limits the capability of RSMT to be applied to event-driven systems. In this chapter, a method of combining RSMT and online scheduling algorithm is given. After the method is introduced, to show its usage in practical applications, a case study on a running car which comprises of radar, cruise control, and engine control is conducted. Through the case study, a method for generating task dependency relation based on SOFL (*structured-object-oriented-formal*) specification is also provided.

Chapter 6. Related works are summarized. Some discussions on validity of RSMT, considering critical resources are conducted. Advantages, disadvantages, and future works are given.

Chapter 2

Framework of the SMT-based Scheduling Methodology

In this chapter, the framework of RSMT which is based on satisfiability modulo theories (SMT) [18] is given. The design guidelines for scheduling uniprocessor and multiprocessor real-time systems described in Chapter 3 and 4 are all based on this framework.

Organization of this chapter. The brief introduction of SMT is given in section 2.1. The framework is explained in section 2.2. To illustrate the usage of the framework, a simple example is studied in this chapter. The application scenario and its applied task model is explained in section 2.3. The key work that using first-order logical formulas to formalize the scheduling problem is described in section 2.4. The process of generating the desired schedule is explained in section 2.5, while section 2.6 gives the scheduling result obtained by applying the framework to the studied example. Discussion and summary are given in section 2.7.

2.1 Satisfiability Modulo Theories (SMT)

Satisfiability modulo theories (SMT) is extension of boolean satisfiability (SAT). It checks satisfiability of logic formulas in first-order formulation with regard to certain background theories like linear integer arithmetic or bit-vectors [18]. A first-order logic formula uses variables as well as quantifiers, functional and predicate symbols, and logical operators

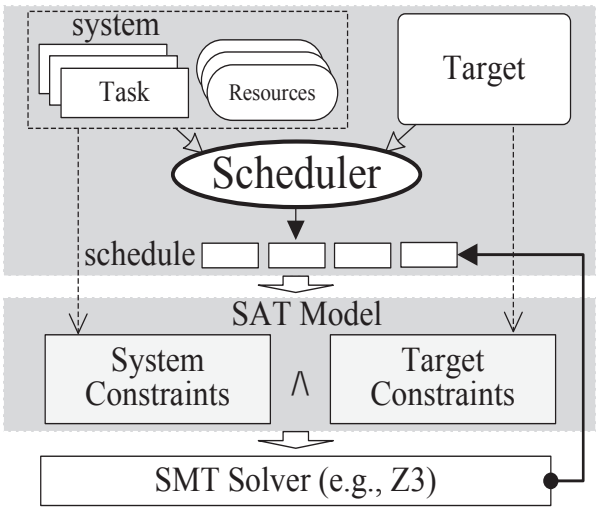

Figure 2.1: Framework of RSMT

[19]. A formula F is *satisfiable*, if there is an interpretation that makes F `true`. For example, formula $\exists a, b \in \mathbb{R}, (b > a + 1.0) \wedge (b < a + 1.1)$, where $\mathbb{R}$ is real number set, is satisfiable, as there is an interpretation, $a \mapsto -1.05, b \mapsto 0$, that makes F `true`. In contrast, a formula F is *unsatisfiable*, if there does not exist any interpretation that makes F `true`. For example, if we define $\exists a, b \in \mathbb{Z}$, where $\mathbb{Z}$ is integer set, the formula $(b > a + 1.0) \wedge (b < a + 1.1)$ will be unsatisfiable.

For a satisfiability problem that has been formalized by first-order logic formulas, a SMT solver (e.g., Z3 [20], Yices [21]) can be employed to solver such a problem. If all the logic formulas are satisfiable, the SMT solver will return the result `sat` and a *solution model* which contains an interpretation for all the variables defined in the formulas that makes all the formulas `true`. For the case $\exists a, b \in \mathbb{R}$, the model is: $a \mapsto -1.05, b \mapsto 0$. If there is an unsatisfiable logic formula, SMT solver returns the result `unsat` with an empty model, for the case $\exists a, b \in \mathbb{Z}$.

2.2 Framework

The framework of RSMT is illustrated in Figure 2.1. There are three layers included in the framework.

The first layer is *problem modeling* layer. In a real-time system, tasks are required to be completed before their deadlines. In order to satisfy this requirement, their required computation resources (e.g., processor) should be allocated to tasks at the right time.

The allocated result is called *schedule*, while the allocating process is called *scheduling* which is conducted by a *scheduler* equipped in the system.

In the framework, the tasks are modeled based on task model and function model (explained in Chapter 4), while the computation resources are modeled based on network channel model and processor model (only for multiprocessor systems, explained in Chapter 4). For a specific scheduling target, it can be expressed based on the modeled tasks and computation resources.

The second layer is *problem formalization* layer. In the framework, the problem of scheduling is treated as a *satisfiability problem*. In order to solve this satisfiability problem, the key work is to formalize the problem using first-order logical formulas (see details in Chapter 2, 3, and 4). Through formalization, a *SAT model* is constructed to represent the scheduling problem. This SAT model is a set of first-order logic formulas (within linear arithmetic in the formulas) which express all the *scheduling constraints* that a desired schedule should satisfy. There are two kinds of constraints: *system constraints* and *target constraints*. System constraints are based on the characteristics of the specific system (i.e., tasks and computation resources). For example, if two tasks run on a processor, a schedule should make sure that the execution time of these two tasks cannot have overlap. Target constraints are based on the scheduling targets. For example, for a hard real-time systems, a schedule should guarantee that all the tasks can meet their deadlines.

The third layer is *problem solving* layer. After the SAT model is constructed, a SMT solver (e.g., Z3, Yices) is employed to solve the formalized problem. A *solution model* can be returned by the SMT solver. This solution model gives an interpretation for all the variables defined in the SAT model, and under the interpretation, all the logic formulas in the SAT model are evaluated as `true`. It means the satisfiability problem represented by the SAT model (i.e., the scheduling problem) is solved, and based on this interpretation, a desired schedule can be generated. To illustrate the usage of the framework, a simple example is given as follows.

2.3 Application Scenario and Task Model

Considering a uniprocessor real-time system containing multiple tasks which are independent with each other. These tasks are required to be completed before their deadlines,

and missed deadline tasks are useless to the system. In other words, these tasks have firm deadlines. For simplicity, task preemption is not allowed.

Applying to this application scenario, the task model used in this example is given below. Each task τ_i is a 3-tuple $\tau_i = (r_i, e_i, d_i)$, where i is the index of a task, r_i is the request time instant, e_i is the required execution time, and d_i is the deadline. A task τ_i requires a processor to execute at the time instant r_i, and a reasonable task should meet that $r_i + e_i \leq d_i$ and $e_i > 0$. Symbol $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes the set of tasks contained in the system, where n is the number of tasks. Symbol s_i is used to represent the start execution time of task τ_i. A successfully completed task (i.e., a task which has been completed before its deadline) τ_i means it has been allocated e_i time slots in time interval $[r_i, d_i)$. A task τ_i should be discarded at system time t, if it features $e_i > d_i - t$, as such task cannot be successfully completed.

Based on this task model, let's consider a specific example. A uniprocessor real-time system comprises of four tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_4\}$, where $\tau_1 = (0, 1, 5), \tau_2 = (0, 2, 4), \tau_3 = (2, 1, 6)$, and $\tau_4 = (2, 1, 3)$. These tasks are independent with each other, and task preemption is not allowed. Notice that, as described in Chapter 1, since RSMT performs in offline paradigm, the parameters of the tasks that contained in the system are required to be known before system run-time, that is, the system is a time-driven system.

2.4 Scheduling Constraints

To apply the framework, the key work is to formalize the problem using first-order logical formulas to construct the SAT model. The variables defined in the SAT model are the start execution time for all the tasks contained in the system, i.e., s_i for $\forall \tau_i \in \mathcal{T}$.

System Constraints

2.4.1 Constraint on start execution time of tasks

As a task can start to run only after it requests, the start execution time s_i of a task should be larger than or equal to its request time instant r_i.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &s_i \geq r_i \end{aligned} \tag{2.1}$$

2.4.2 Constraint on processor

A processor can execute only one task at a time. This is interpreted as: there is no overlap of the execution time of any two tasks.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j \\ &(s_i \geq s_j + e_j) \vee (s_j \geq s_i + e_i) \end{aligned} \tag{2.2}$$

Target Constraints

2.4.3 Constraint on deadlines of tasks

A successfully completed task τ_i should be completed before its deadline.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &s_i + e_i \leq d_i \end{aligned} \tag{2.3}$$

2.5 Schedule Synthesis

After all the constraints are defined, now we can employ a SMT solver to generate a desired schedule. The whole process of generating the schedule is summarized in Alg. 1.

Function `Assert`($\mathcal{T}$) (line 1) interprets the constraints defined in section 2.4 as *assertions* $\mathcal{A}$ (boolean formulas that can be input into a SMT solver). The variables of these boolean formulas are the start time s_i for $\forall \tau_i \in \mathcal{T}$. Function `CallSMTSolver`($\mathcal{A}$) (line 2) calls a SMT solver to find a solution model $\mathcal{M}$ for $\mathcal{A}$.

If the solution model does exist, indicated by a non-empty model $\mathcal{M}$ is returned (line 3), based on the solution model $\mathcal{M}$, the schedule $\mathcal{S}$ can be returned by the function `GenSch`($\mathcal{M}$) (line 4). When there does not exist a solution model, indicated by returning an empty model $\mathcal{M}$ (line 5), which means the target constraints (defined in formula 2.3)

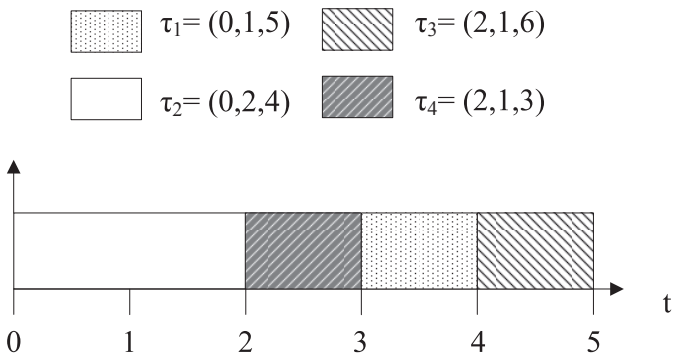

Figure 2.2: Scheduling result

Algorithm 1 Schedule synthesis

Input: task set $\mathcal{T}$

Output: schedule $\mathcal{S}$

1: $\mathcal{A} :=$ `Assert`$(\mathcal{T})$
2: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
3: **if** $\mathcal{M} \neq \emptyset$ **then**
4: **return** $\mathcal{S} :=$ `GenSch`$(\mathcal{M})$
5: **else**
6: **return** UNFEASIBLE
7: **end if**

cannot be satisfied, that is, there does not exist a schedule that can make all the tasks meet their deadlines. Under this case, the algorithms returns UNFEASIBLE.

2.6 Scheduling Result

Recall the example introduced in section 2.3. Through the schedule synthesis described in section 2.5, we can get the solution model as: $s_1 = 3, s_2 = 0, s_3 = 4, s_4 = 2$, and the corresponding schedule $\mathcal{S}$ shown in Figure 2.2. We can see that the generated schedule $\mathcal{S}$ can make all the tasks meet deadlines.

2.7 Discussion and Summary

In this chapter, the framework of RSMT is given. This framework gives a unified design strategy for developers designing scheduling for real-time systems. In the framework, scheduling problem is treated as a satisfiability problem which is formalized by first-order logical formulas. Such formalization provides a firm theoretic foundation to solve the

scheduling problems.

From the description of the *problem solving* layer of the framework, we can see that, after the SAT model is constructed, there are three steps for RSMT generating the scheduling table: *i)*: interpreting the SAT model as the input format of the underlying SMT solver (interpreting), *ii)*: SMT solver computing the scheduling problem and outputting a solution model (computing), and *iii)*: generating a scheduling table based on the solution model (generating). Compared to the operations of interpreting and generating which are quite straightforward, operation of the computing has exponential complexity [23] and becomes the major limitation of RSMT (more details will be discussed in Chapter 4).

Chapter 3

Scheduling for Uniprocessor Real-Time Systems

3.1 Introduction

A task in a real-time system is required to be completed before its deadline. Such sensitivity to timing is central feature of system behaviors [38]. For firm and soft real-time systems, under normal workload conditions, a scheduler with a proper scheduling policy can make all the tasks be completed before their deadlines. However, in practical environment, system workload may vary widely because of dynamic changes of work environment. Once system workload becomes too heavy so that there does not exist a feasible schedule can make all the tasks meet their deadlines, we say the system is *overloaded*. Note that, even in well-defined systems, under some special cases (e.g., transient transition between different operating modes), such overload problem may also happen.

When the overload problem happens, it is important to minimize the degree of system performance degradation caused by the missed deadline tasks. A system that panics and suffers a drastic fall in performance when a problem happens, is likely to contribute to this problem rather than help solve it [2]. To achieve this target, the design of scheduling is crucial, as different scheduling policies will lead to different degrees of performance degradation. Many objectives for the design of scheduling policies described in [3, 4] can be considered. For example, (*i*) maximizing total number of tasks that meet deadlines, (*ii*) maximizing effective processor utilization, (*iii*) maximizing obtained values of completed

tasks. Which objectives are appropriate in a given situation depends, of course, upon the application. I first focus on maximizing total number of tasks that meet their deadlines. This objective is reasonable for a firm real-time systems upon the application scenario that when a missed deadline task corresponds to a disgruntled customer, and the aim is to keep as many customers satisfied as possible [2]. Then, RSMT is applied to the design objectives: maximizing effective processor utilization and maximizing obtained values of completed tasks.

Organization of this chapter. In section 3.2, system model is presented and research preliminary is given. Scheduling constraints defined in the SAT model are given in section 3.3. The process of generating the desired schedule is explained in section 3.4. Simulations and performance evaluation are shown in section 3.5. In section 3.6, design guidelines for applying RSMT to different scheduling targets are given. Summary is given in section 3.7.

3.2 System Model, Definition, and Preliminary

3.2.1 System Model

In this chapter, the general *firm-deadline* model proposed in [6] is adopted for systems with uniprocessor. The "firm-deadline" means only tasks completed before their deadlines are considered valuable, and any task missing its deadline is worthless to system.

This task model is similar to the task model used in Chapter 2. Particularly, in order to allow task preemption, for all tasks in $\mathcal{T}$, task τ_i is defined as consisting of a series of indivisible fragment (atomic operation), denoted by $\tau_i : (f_1, f_2, \ldots, f_m)$, where $m = |\tau_i|$ is the number of fragments in task τ_i[1]. $f_{i,j}$ denotes the j-th fragment of τ_i, and the last fragment of task τ_i is denoted by $f_{i,e}$. Symbol $s_{i,j}$ is used to represent the start execution time of $f_{i,j}$. Symbol $e_{i,j}$ denotes the required execution time of $f_{i,j}$, and $\forall f_j \in \tau_i, \sum e_{i,j} = e_i$. For $1 \leq i < m$, f_{i+1} can start to run only when f_i has been completed. A successfully completed task τ_i means $f_{i,e}$ has been allocated $e_{i,e}$ time slots

[1]RSMT can also deal with the condition that task preemption is prohibited, by just constraining every tasks consisting of only one indivisible fragment.

in time interval $[r_i, d_i)$. Symbol re_i represents the remaining execution time of task τ_i. Initially, it equals to e_i. After τ_i has been executed for δ ($\delta \leq e_i$) time slots, $re_i = e_i - \delta$. If $re_i = 0$, it means τ_i has been completed. A task τ_i should be discarded at system time t, if it features $re_i > d_i - t$, as such task cannot be successfully completed.

3.2.2 Definition

In a real-time system, scheduler can use different schedules to schedule task set $\mathcal{T}$.

When there exists a schedule that can make all tasks meet their deadlines, the system is ***underloaded****, and the task set is* ***feasible****. In contrast, when there does not exist a schedule that can make all the tasks meet their deadlines, the system is* ***overloaded****, and the task set is* ***infeasible*** *(refer [4]).*

An example in Figure 3.1 is used to illustrate this definition. As shown in Figure 3.1 (a), at $t = 0$, $\mathcal{T}' = \{\tau_1, \tau_2\}$, where $\mathcal{T}'$ is the set of the tasks that have arrived in the system (not including tasks that have been successfully completed or have missed deadlines). Using earliest deadline first (EDF) algorithm to schedule $\mathcal{T}'$ can make all tasks meet their deadlines, where EDF first schedules the task with the earliest deadline. Thus, the system is underloaded, and the task set $\mathcal{T}'$ is feasible. EDF algorithm proposed in literature [14] has been proven as an *optimal* scheduling algorithm on uniprocessor [33]. That is, if using EDF to schedule a task set cannot make all tasks meet their deadlines, no other algorithms can. Thus, EDF scheduling algorithm can be used to tell if a task set is feasible.

After the system passed a time unit, as shown in Figure 3.1 (b), τ_1 has been successfully completed, and a new task τ_3 arrives in the system. At that time, $\mathcal{T}' = \{\tau_2, \tau_3\}$. Using EDF to schedule $\mathcal{T}'$ can only make τ_3 meet its deadline. Task τ_2 should be discarded at $t = 2$, as $re_2 > d_2 - t$, where $d_2 = 3, re_2 = 2$. Thus, the system is overloaded, and the task set $\mathcal{T}'$ is infeasible.

3.2.3 Preliminary

There are many scheduling algorithms used in various real-time systems. In this subsection, through an example described in Figure 3.2, the performance of three widely used scheduling algorithms, shortest remaining time first (SRTF), EDF, and least laxity first

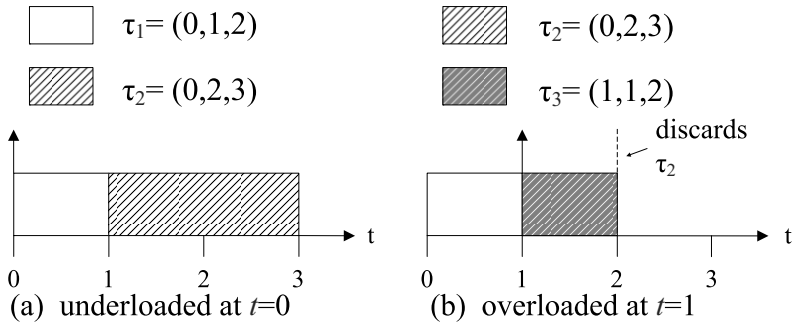

Figure 3.1: Example for *underloaded* and *overloaded*

(LLF), are studied. In the example, the lengths of all the indivisible fragments in all the tasks are set to one.

Scheduling results: (i): SRTF first schedules the task with the shortest remaining execution time. The scheduling sequence is $(\tau_3, \tau_1, \tau_1, \tau_4, \tau_1)$. By this sequence, τ_4 and τ_1 can be completed sequentially. (ii): EDF first schedules the task with the earliest deadline. The result of the scheduling sequence is $(\tau_2, \tau_2, \tau_2, \tau_2, \tau_2, \tau_4)$. It can complete τ_2 and τ_4 sequentially. (iii): LLF first schedules the task with the least laxity. For τ_i, the laxity l_i is computed as $l_i = d_i - re_i - t$. It can complete tasks τ_2 and τ_4 sequentially with the same scheduling sequence generated by EDF.

All of the three scheduling algorithms achieve two as the number of task completion. To know if it is the maximum value, for this simple example with only four tasks, we can enumerate all the schedule to find the maximum number of task completion. An optimal schedule (schedule that can achieve the maximum number of task completion) is $(\tau_3, \tau_3, \tau_3, \tau_3, \tau_1, \tau_1, \tau_1, \tau_4)$ which can complete three tasks τ_3, τ_1, and τ_4 sequentially.

Based on above analysis, through this example, we can see that, for overloaded real-time system, a new scheduling method is needed.

3.3 Scheduling Constraints

Compared to the scheduling problem studied in Chapter 2, to solve the overload problem, more scheduling constraints are needed to construct the SAT model. This section describes all the constraints expressed in the SAT model.

System Constraints

3.3.1 Constraint on start execution time of tasks

As a task can only start to run after it requests, the start time of the first fragment of a task should be larger than or equal to the request time instant r_i.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &s_{i,1} \geq r_i \end{aligned} \tag{3.1}$$

3.3.2 Constraint on start execution time of different fragments

The series of fragments contained in a task should be executed sequentially. Therefore, a fragment of a task can start to run only when its previous fragments of the task have been completed.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T}, \forall f_a, f_b \in \tau_i \\ &b > a \Rightarrow s_{i,b} \geq s_{i,a} + e_{i,a} \end{aligned} \tag{3.2}$$

Symbol $\Rightarrow$ denotes implication logical operator.

3.3.3 Constraint on processor

A processor can execute only one fragment at a time. This is interpreted as: there is no overlap of the execution time of any two fragments of any two different tasks.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall f_a \in \tau_i, \forall f_b \in \tau_j \\ &(s_{i,a} \geq s_{j,b} + e_{j,b}) \vee (s_{j,b} \geq s_{i,a} + e_{i,a}) \end{aligned} \tag{3.3}$$

3.3.4 Constraint on task dependency

In practical systems, tasks usually have dependency relation with each other. For example, task τ_j may require the computed results of τ_i; or task τ_i can issue a signal to activate task τ_j only when τ_i has been completed. Thus, τ_j can start to run only after τ_i has been completed. Such dependency relation is denoted as $\tau_i \prec \tau_j$.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T} \\ &\tau_i \prec \tau_j \Rightarrow (s_{j,1} \geq s_{i,e} + e_{i,e}) \wedge \\ &(s_{i,e} + e_{i,e} > d_i \Rightarrow s_{j,1} = +\infty) \end{aligned} \tag{3.4}$$

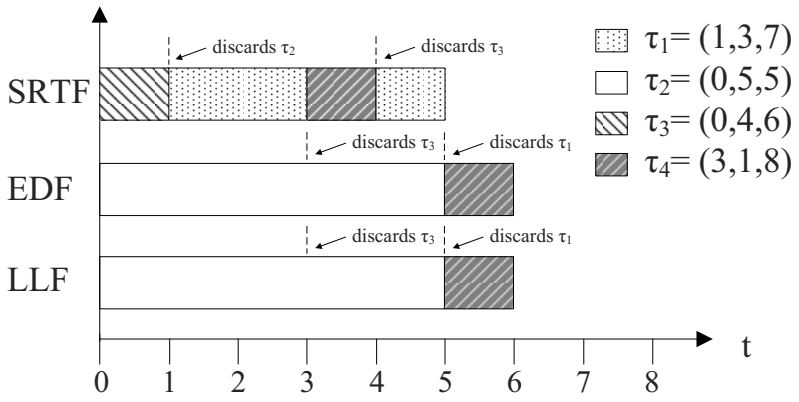

Figure 3.2: Performance of scheduling algorithms

This formula expresses that any two tasks that have dependency relation $\tau_i \prec \tau_j$, the first fragment of task τ_j can start to run only when the last fragment of task τ_i has been completed. As the series of fragments contained in a task are executed sequentially, this formula can make sure that task τ_j starts to run only after τ_i has been completed. Moreover, if task τ_i has not been successfully completed, task τ_j cannot start to run.

Target Constraints

3.3.5 Constraints on scheduling target

A successfully completed task τ_i should be completed before its deadline. As all the fragments contained in a task run sequentially, this constraint can be interpreted as: for task τ_i, if its last fragment $f_{i,e}$ can complete before its deadline d_i, task τ_i will be successfully completed. Let n be the number of successfully completed tasks, and its initial value is set to 0.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &\texttt{if } (s_{i,e} + e_{i,e} \leq d_i) \\ &\quad n := n + 1 \\ &\texttt{end} \end{aligned} \tag{3.5}$$

where symbol $e_{i,e}$ is the required execution time of $f_{i,e}$. Let symbol sn denote the maximum number of tasks in $\mathcal{T}$ that can be successfully completed, and obviously, $sn \in [0 \ |\mathcal{T}|]$.

The constraints on scheduling target can be expressed as:

$$n \geq sn \tag{3.6}$$

Note that, here the constraints on scheduling target is set as $n \geq sn$ rather than $n = sn$ is based on experience. With $n \geq sn$, following the schedule synthesis described in the next section, the time efficiency of RSMT can be better than with $n = sn$.

3.4 Schedule Synthesis

After all the constraints are defined, now we can employ a SMT solver to generate a desired schedule. The process of schedule synthesis is summarized in Alg. 2. Function `Assert`$(\mathcal{T}, |\mathcal{T}|)$ (line 1) interprets the constraints defined in section 3.3 as *assertions* (boolean formulas that can be input into a SMT solver) with $|\mathcal{T}|$ as the maximum number of successfully completed tasks (i.e., set $sn := |\mathcal{T}|$ in *constraints on scheduling target*). The variables of these boolean formulas are start time $s_{i,j}$ for $\forall \tau_i \in \mathcal{T}, \forall f_j \in \tau_i$. Function `CallSMTSolver`$(\mathcal{A})$ (line 2) calls a SMT solver to find a solution model $\mathcal{M}$ for $\mathcal{A}$. If such a model does exist, it will be returned by the function, otherwise an empty model will be returned.

For constraints on scheduling target, it is first set as $sn := |\mathcal{T}|$, that is to expect all the tasks in $\mathcal{T}$ can be successfully completed. If this expectation can be satisfied, it means overload problem does not happen, solution model $\mathcal{M}$ will be returned. As $\mathcal{M}$ contains all the values of $s_{i,j}$, for $\forall \tau_i \in \mathcal{T}, \forall f_j \in \tau_i$, the start execution time of all the fragments of all the tasks can be extracted, it means the schedule $\mathcal{S}$ for task set $\mathcal{T}$ can be generated (line 1-5).

When overload problem happens, not all tasks in $\mathcal{T}$ can be successfully completed. This condition is indicated by an empty model returned by function `CallSMTSolver`$(\mathcal{A})$, which means *constraints on scheduling target* cannot be satisfied. We need to decrease the set value of sn. To achieve the maximum number of task completion means to find the maximum value of sn with which there exists a solution model. *Binary search* is used to find the maximum value of sn (line 6–22). With the maximum value of sn, a solution model can be returned by function `CallSMTSolver`$(\mathcal{A})$. Meanwhile, with $sn := sn + 1$, `CallSMTSolver`$(\mathcal{A})$ will return an empty model. This is the criterion to judge if the value

Algorithm 2 Schedule synthesis

Input: task set $\mathcal{T}$

Output: schedule $\mathcal{S}$

1: $\mathcal{A} :=$ `Assert`$(\mathcal{T}, |\mathcal{T}|)$
2: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
3: **if** $\mathcal{M} \neq \emptyset$ **then**
4: **return** $\mathcal{S}$ based on model $\mathcal{M}$
5: **end if**
6: $start := 0, end := |\mathcal{T}|$
7: **while** `true` **do**
8: $mid := start + \lfloor (end - start)/2 \rfloor$
9: $\mathcal{A} :=$ `Assert`$(\mathcal{T}, mid)$
10: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
11: **if** $\mathcal{M} = \emptyset$ **then**
12: $end := mid - 1$
13: **else**
14: $\mathcal{A}' :=$ `Assert`$(\mathcal{T}, mid + 1)$
15: $\mathcal{M}' :=$ `CallSMTSolver`$(\mathcal{A}')$
16: **if** $\mathcal{M}' \neq \emptyset$ **then**
17: $start := mid + 1$
18: **else**
19: **return** $\mathcal{S}$ based on model $\mathcal{M}$
20: **end if**
21: **end if**
22: **end while**

of sn is the maximum value. When we get the solution model $\mathcal{M}$ with the maximum value sn, based on $\mathcal{M}$, the schedule $\mathcal{S}$ can be generated (line 19).

Through the procedure of the schedule synthesis, the maximum value of sn can be found. Meanwhile, as all the constraints defined in the SAT model have been satisfied, it means $\mathcal{S}$ can achieve the maximum number of task completion.

3.4.1 Scheduling Results

Recall the example shown in Figure 3.2. In this example, $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. Based on the schedule synthesis shown in Alg. 2, we can get the solution model $\mathcal{M}$ which defines the values of $s_{i,j}$ for $\forall \tau_i \in \mathcal{T}, \forall f_j \in \tau_i$. The model is as follows: $s_{1,1} = 4, s_{1,2} = 5, s_{1,3} = 6, s_{2,1} = 8, s_{2,2} = 9, s_{2,3} = 10, s_{2,4} = 11, s_{2,5} = 12, s_{3,1} = 0, s_{3,2} = 1, s_{3,3} = 2, s_{3,4} = 3, s_{4,1} = 7$. Based on this model, as shown in Figure 3.3 (without $\tau_1 \prec \tau_4$), we can get the scheduling sequence $\mathcal{S} = (\tau_3, \tau_3, \tau_3, \tau_3, \tau_1, \tau_1, \tau_1, \tau_4)$ (as τ_2 cannot be successfully completed, it should not be included in $\mathcal{S}$). This scheduling sequence can complete three

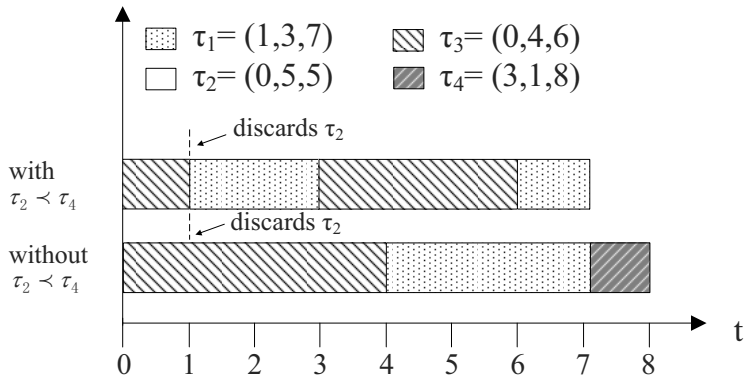

Figure 3.3: Results by using RSMT for the example shown in Figure 3.2

tasks τ_3, τ_1, and τ_4 consequently, which is the maximum number of task completion for $\mathcal{T}$.

If we add a task dependency relation $\tau_2 \prec \tau_4$, we can get model: $s_{1,1} = 1, s_{1,2} = 2, s_{1,3} = 6, s_{2,1} = 7, s_{2,2} = 8, s_{2,3} = 9, s_{2,4} = 10, s_{2,5} = 11, s_{3,1} = 0, s_{3,2} = 3, s_{3,3} = 4, s_{3,4} = 5, s_{4,1} = 12$. Based on this model, as shown in Figure 3.3 (with $\tau_2 \prec \tau_4$), we can get the scheduling sequence $\mathcal{S} = (\tau_3, \tau_1, \tau_1, \tau_3, \tau_3, \tau_3, \tau_1)$. This scheduling sequence can complete two tasks τ_3 and τ_1 consequently, which is also the maximum number of task completion for $\mathcal{T}$ with the dependency relation $\tau_2 \prec \tau_4$.

3.5 Simulation and Evaluation

In this section, results of the simulations which are conducted to study the performance of RSMT are presented. A prototype tool for RSMT, based on the system model, constraint formulation, and schedule synthesis described above, has been implemented. The underlying SMT solver employed by the tool is Z3 which is a state-of-the art SMT solver. In addition to the three well-known algorithms: SRTF, EDF, and LLF, two algorithms proposed in my previous works: DPSC [2] [54] and GSFC [3] [52], which are specific to the overload problem are also adopted as the baseline algorithms.

[2]DPSC uses the idea of dynamic programming and congestion control to deal with the overload problem.

[3]GSFC uses the idea of greedy algorithm and feedback control to deal with the overload problem.

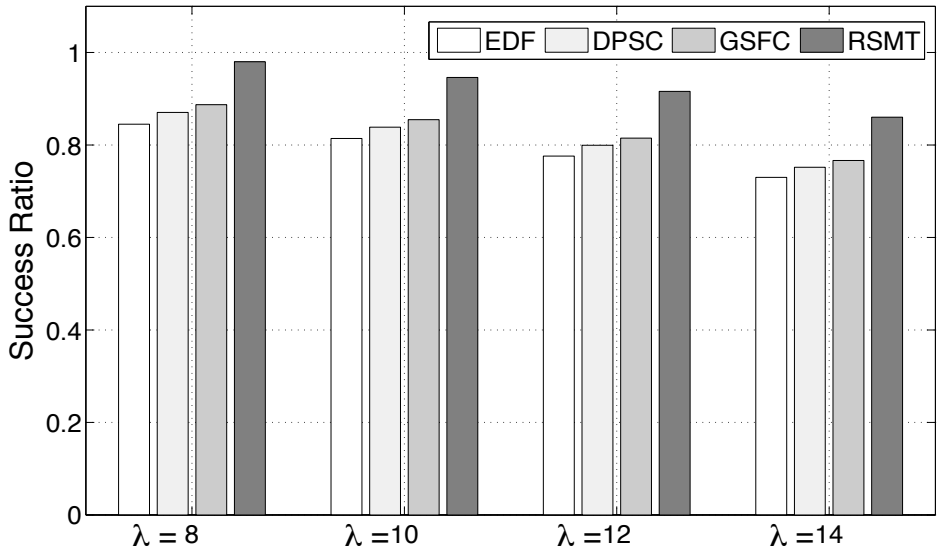

Figure 3.4: Success ratio of RSMT and the baseline scheduling algorithms

3.5.1 Simulation Settings

The metric used to compare the performance of different scheduling methods is *success ratio* which denotes the ratio of the input tasks that have been completed before their deadlines to all the input tasks. For the time complexity, RSMT has exponential complexity, which is much more complex than the baseline algorithms[4]. In order to evaluate the time that RSMT spends on scheduling all the input tasks, the simulation results of the SMT solver *computing* time and the *total spent* time (including interpreting, computing, and generating time) of RSMT are calculated in the simulation studies.

The input tasks are generated according to Poisson distribution with arriving rate λ which represents the number of tasks that arrive in the system per 100 time units. As the workload can be changed by λ, the attributes of tasks in the simulations are given a simple setting. For each task τ_i, e_i varies in $[1,\ 6]$ with exponential distribution. The number of indivisible fragments of a task varies in $[1,\ 3]$. To characterize tasks with different tightness deadline, the assignment of d_i is according to equation: $d_i = r_i + sf_i * e_i$, where sf_i is slack factor that indicates the tightness of task deadline. For each task τ_i, sf_i varies in $[1,\ 6]$.

[4]If quick sort algorithm is used to conduct sort operation, the worst-case time complexity of EDF, SRTF, LLF are quadratic, while DPSC and GSFC are pseudo-polynomial.

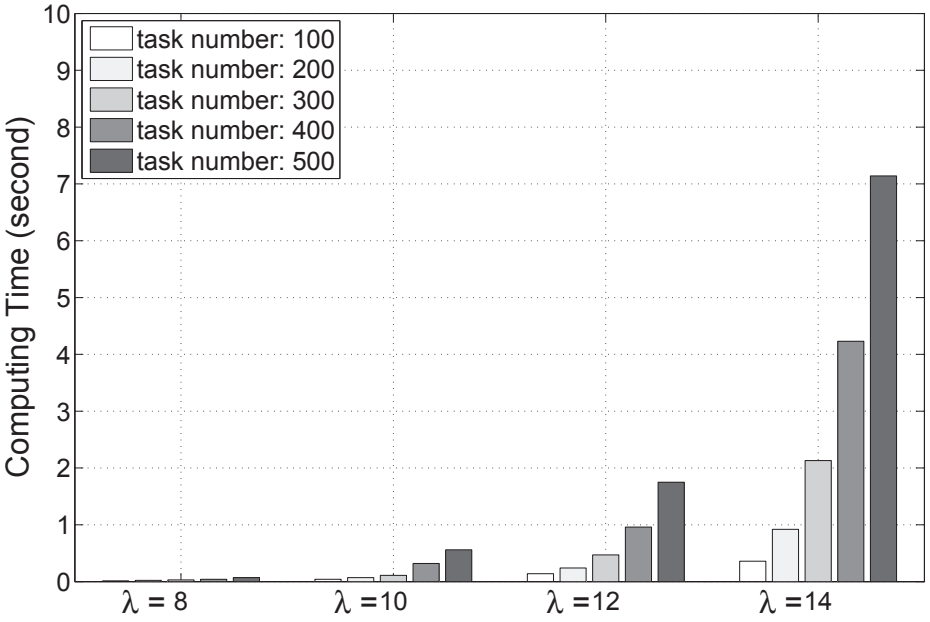

Figure 3.5: SMT solver computing time

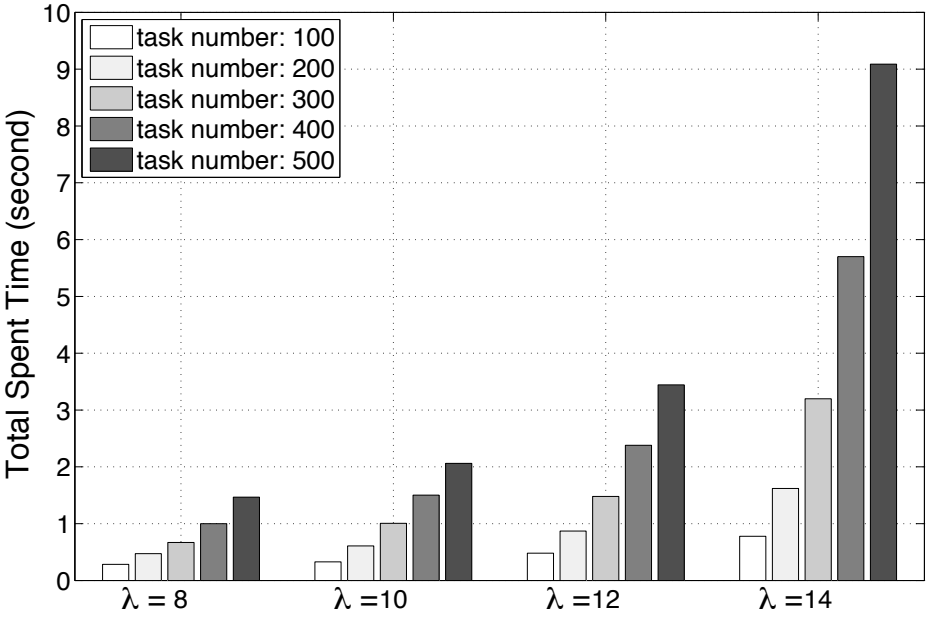

Figure 3.6: Total spent time of RSMT

In a well-defined system, usually the length of system overload time is not long, and the degree of system overload is not serious. If a system is under overload condition for a long time or the degree of system overload is very serious, it means the capability of the

system is not enough to handle its work. Based on this observation, the values of λ are set as $8, 10, 12$, and 14 to represent different degrees of system overload conditions. The input total number of tasks are set as $100, 200, 300, 400$, and 500 to represent different lengths of system overload time. All the simulations are run on a 64bit 4-core 2.5 GHz Intel Xeon E3 PC with 32GB memory.

3.5.2 Evaluation

The simulation results are shown in Figure 3.4, Figure 3.5, and Figure 3.6. The values shown in the figures are the average values of running simulation 100 times. As the performance of EDF, SRTF, and LLF are almost the same in terms of success ratio, only the results of EDF are shown in Figure 3.4.

For success ratio, through the analysis in section 3.4, RSMT can achieve the optimal result[5]. As shown in Figure 3.4, the values of success ratio for RSMT are larger than the baseline algorithms under all the values of λ, it means RSMT performs better than the baseline algorithms in terms of success ratio.

For SMT solver computing time, as shown in Figure 3.5, with increasing value of λ and task number, the value of computing time is increasing. This is because, with bigger value of λ, more tasks wait to be executed on the processor at a time. As described in section 3.3, the scheduling constraint *Constraint on processor* guarantees that no overlap of the execution time of any two tasks on a processor. The increasing number of waiting tasks will make the underlying SMT solver need much more calculation to return a solution model. Similarly, with increasing value of task number, the total number of tasks that need to be scheduled is increasing. This will also increase the calculation time for the underlying SMT solver to return a solution model.

For total spent time of RSMT, from Figure 3.6 we can see that RSMT needs several seconds to generate the desired scheduling table, which means RSMT is capable to handle the scheduling problem.

[5]Only when all the scheduling constraints are defined in the SAT model. More details will be discussed in section 6.2.

3.6 Adapting to Other Scheduling Targets

In the constructed SAT model, there are two kinds of scheduling constraints: system constraints and target constraints. This kind of design makes RSMT flexible to design scheduling for other objectives by just modifying the target constraints. In this section, two examples are given to show how to apply RSMT to different design objectives.

3.6.1 Maximizing Effective Processor Utilization

Effective Processor Utilization (EPU) measures the fraction of time that the processor spends on executing tasks which are successfully completed before their deadlines. For a firm real-time system, if customers pay for the usage of the processor only when their tasks have been successfully completed, EPU may be a reasonable measure [4].

Constraints on scheduling target

A successfully completed task τ_i should be completed before its deadline. As mentioned in section 3.3.5, since all the fragments contained in a task run sequentially, this constraint can be interpreted as: the last fragment of a successfully completed task should be completed before its deadline. Let symbol e denote effective processor time (time that the processor spends on executing tasks which are successfully completed before their deadlines), and its initial value is set to 0.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &\texttt{if } (s_{i,e} + e_{i,e} \leq d_i) \\ &\quad e := e + e_i \\ &\texttt{end} \end{aligned} \tag{3.7}$$

Let symbol $ss_{i,e}$ denote the maximum value of $s_{i,e}$ for tasks in $\mathcal{T}$ that have been successfully completed. The effective processor utilization epu can be calculated as:

$$epu = \frac{e}{ss_{i,e} + e_{i,e}} \tag{3.8}$$

Let symbol $sepu$ denote the maximum value of effective processor utilization, and obviously, $sepu \in [0 \;\; 1]$. The constraints on scheduling target can be expressed as:

$$epu \geq sepu \tag{3.9}$$

Algorithm 3 Schedule synthesis for maximizing effective processor utilization

Input: task set $\mathcal{T}$

Output: schedule $\mathcal{S}$

```
1: A := Assert(T, 1)
2: M := CallSMTSolver(A)
3: if M ≠ ∅ then
4:     return S based on model M
5: end if
6: start := 0, end := 1
7: while true do
8:     mid := start + (end − start)/2
9:     A := Assert(T, mid)
10:    M := CallSMTSolver(A)
11:    if M = ∅ then
12:        end := mid − step
13:    else
14:        A' := Assert(T, mid + step)
15:        M' := CallSMTSolver(A')
16:        if M' ≠ ∅ then
17:            start := mid + step
18:        else
19:            return S based on model M
20:        end if
21:    end if
22: end while
```

Schedule Synthesis

The process of schedule synthesis is summarized in Alg. 3. This process of schedule synthesis is quite similar to the process described in Alg. 2. Function $\texttt{Assert}(\mathcal{T}, 1)$ (line 1) interprets the system constraints defined in section 3.3 and target constraints descried in section 3.6.1 as assertions with $sepu := 1$ in *constraints on scheduling target*. If this expectation can be satisfied, it means overload problem does not happen, model $\mathcal{M}$ will be returned by function $\texttt{CallSMTSolver}(\mathcal{A})$ (line 2). Based on $\mathcal{M}$, the schedule $\mathcal{S}$ can be generated (line 4).

When overload problem happens, $sepu := 1$ cannot be satisfied. We need to decrease the set value of *sepu*. Binary search is used to find the maximum value of *sepu* (line 6–22). With the maximum value of *sepu*, a solution model can be returned by function $\texttt{CallSMTSolver}(\mathcal{A})$. Meanwhile, with $sepu := sepu + step$, $\texttt{CallSMTSolver}(\mathcal{A})$ will return an empty model. This is the criterion to judge if the value of *sepu* is the maximum value. Note that, the variable *step* is predefined and can be used to control search space of SMT

solver. Increasing *step* makes the algorithm faster but also reduces the solution space.

3.6.2 Maximizing Obtained Values of Completed Tasks

In a real-time system, when a task is successfully completed, it means a utility of the system has been achieved. In other words, through completing tasks, some values can be obtained by the system.

When tasks are treated as equally important, the values obtained by the system through completing different tasks are the same. Thus, in that case, the design objective maximizing the obtained values of completed tasks is the same as the design objective maximizing total number of tasks that meet deadlines. However, when tasks are treated as with different degrees of importance, these two objectives become different.

For a firm real-time system that contains tasks with different degrees of importance, each task τ_i is a 4-tuple $\tau_i = (r_i, e_i, d_i, v_i)$, where v_i is the value of task τ_i that can be obtained by the system when τ_i is successfully completed.

Constraints on scheduling target

Let symbol v be the obtained values of the completed tasks, and its initial value is set to 0.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} \\ &\texttt{if } (s_{i,e} + e_{i,e} \leq d_i) \\ &\quad v := v + v_i \\ &\texttt{end} \end{aligned} \tag{3.10}$$

Let symbol sv denote the maximum obtained values of completed tasks, and obviously, sv is no less than 0 and no larger than $\sum v_i$ for $\forall \tau_i \in \mathcal{T}$. The constraints on scheduling target can be expressed as:

$$v \geq sv \tag{3.11}$$

Schedule Synthesis

The process of schedule synthesis is summarized in Alg. 4. This process of schedule synthesis is quite similar to the process described in Alg. 2 and Alg. 3.

Algorithm 4 Schedule synthesis for maximizing obtained values of completed tasks

Input: task set $\mathcal{T}$

Output: schedule $\mathcal{S}$

```
1: A := Assert(T, ∑ v_i)
2: M := CallSMTSolver(A)
3: if M ≠ ∅ then
4:     return S based on model M
5: end if
6: start := 0, end := ∑ v_i
7: while true do
8:     mid := start + (end − start)/2
9:     A := Assert(T, mid)
10:    M := CallSMTSolver(A)
11:    if M = ∅ then
12:        end := mid − min_v(T)
13:    else
14:        A' := Assert(T, mid + min_v(T))
15:        M' := CallSMTSolver(A')
16:        if M' ≠ ∅ then
17:            start := mid + min_v(T)
18:        else
19:            return S based on model M
20:        end if
21:    end if
22: end while
```

Function $\texttt{Assert}(\mathcal{T}, \sum v_i)$ (line 1) interprets the system constraints defined in section 3.3 and target constraints descried in formula 3.10 as assertions with $sv := \sum v_i$ in *constraints on scheduling target*. If this expectation can be satisfied, it means overload problem does not happen, model $\mathcal{M}$ will be returned by function $\texttt{CallSMTSolver}(\mathcal{A})$ (line 2). Based on $\mathcal{M}$, the schedule $\mathcal{S}$ can be generated (line 4).

When overload problem happens, $sv := \sum v_i$ cannot be satisfied. We need to decrease the set value of sv. Binary search is used to find the maximum value of sv (line 6–22). With the maximum value of sv, a solution model can be returned by function $\texttt{CallSMTSolver}(\mathcal{A})$. Meanwhile, with $sv := sv + \texttt{min\_v}(\mathcal{T})$, $\texttt{CallSMTSolver}(\mathcal{A})$ will return an empty model, where symbol $\texttt{min\_v}(\mathcal{T})$ represents the minimum value of v_i for $\forall \tau_i \in \mathcal{T}$. This is the criterion to judge if the value of sv is the maximum value.

3.7 Summary

In this chapter, RSMT is applied to uniprocessor real-time systems to solve the overload problem. Design guidelines for many practical requirements that widely studied in real-time scheduling domain are given. Three scheduling targets that are reasonable under the overload condition are considered.

In RSMT, the scheduling constraints contained in the constructed SAT model are divided into two parts: system constraints and target constraints. This kind of design makes RSMT flexible to design scheduling for other objectives. When adapting to different scheduling targets, little modification on target constraints and schedule synthesis needs to make, and the system constraints can be totally reused.

Chapter 4

Scheduling for Multiprocessor Real-Time Systems

4.1 Introduction

Currently, many practical real-time systems are equipped within multiple processors, for which the schedule synthesis to make sure that all the tasks can be completed before their deadlines is known to be an NP complete problem [23]. Even in the simpler case of identical multiprocessor systems, finding a feasible schedule is NP-complete in the strong sense [42] (see an example in [53]).

Many researches (e.g., [24, 25, 26, 27]) have been conducted to challenge the problem of scheduling multiprocessor systems. But, even for the simpler problem of scheduling identical multiprocessor systems, there still remain many problems. A comprehensive survey of scheduling for identical multiprocessor real-time systems can be found in [13]. For the harder problem of scheduling heterogeneous multiprocessor systems, where we have to face an awkward reality that considering a relatively practical application scenario, there is still no solution to efficiently schedule heterogeneous multiprocessor systems.

To solve these problems, in this chapter, RSMT is applied to design scheduling for heterogeneous multiprocessor real-time systems. As scheduling identical and uniform multiprocessor systems can be treated as special cases of scheduling heterogeneous multiprocessor systems, it means RSMT can also be applied to schedule identical and uniform multiprocessor systems. Various kinds of systems (soft, firm, hard, and mixed critical

real-time systems) and scheduling targets have been considered.

Organization of this chapter. In section 4.2, system model is presented. Scheduling constraints defined in the SAT model are given in section 4.3. The process of generating the desired schedule is explained in section 4.4. Simulations and performance evaluation are shown in section 4.5. In this section, based on simulations, problem scope in which RSMT can be applied is also shown. Meanwhile, a promising method which can expand the scope is described. In section 4.6, the design guidelines for adapting RSMT to different scheduling targets are given. Summary is given in section 4.7.

4.2 System Model

4.2.1 Function Set

Function set defines set of the functions that can be achieved by a real-time system. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ denote the function set. Functions contained in the set are independent with each other. Each function $F_i \in \mathcal{F}$ is achieved by a series of tasks, represented as *poset* $(T_i, \prec)$, where $T_i \neq \emptyset$ denotes the set of the corresponding tasks, and symbol $\prec$ denotes the dependency relation of the tasks in T_i (detail of the poset will be explained in the next subsection). For real-time systems, when functions are triggered at system time instant rf_i, they are required to be completed before *deadline*, represented by df_i. Based on above analysis, the function is defined as $F_i = ((T_i, \prec), rf_i, df_i)$.

Note that, unlike conventional researches on real-time scheduling which set deadline to task (in Chapter 2 and 3, deadline is also set to task), in this chapter, the deadline is set to the function level rather than the task level. This setting can better reflect the reality that the deadline requirement is for functions in real-time systems, while a function is achieved by a series of tasks cooperated together. To further illustrate advantage of the function model, an example will be given after all the models are introduced.

4.2.2 Task Poset

For each function, it is achieved by a series of tasks cooperated together. Poset $(T_i, \prec)$ is used to denote such a series of tasks, where $T_i \subseteq \mathcal{T}$ is the task set corresponding to F_i,

and $T_i = \{\tau_1, \tau_2, \ldots, \tau_m\}$, where $\tau_j \in T_i$ is a task, and m is the number of tasks. Symbol $\mathcal{T}$ represents that set of tasks contained in the system. Task $\tau_j \in T_i$[1] is indicated by $\tau_{i,j}$.

If a task $\tau_i \in \mathcal{T}$ is used by multiple functions, for concise expression, multiple copies of the task are defined in the task set $\mathcal{T}$. For example, if task τ_k is used by function F_i and F_j, two tasks (with the same operations of τ_k) are defined in the task set $\mathcal{T}$ to represent τ_k used in function F_i and F_j, respectively.

Symbol $\prec$ indicates the dependency relation between two tasks. That is, $\tau_k, \tau_j \in T_i, \tau_k \prec \tau_j$ indicates that task τ_j can start to run only after task τ_k has been completed. The dependency relation is transitive. That is, $\tau_k \prec \tau_j, \tau_j \prec \tau_l \Rightarrow \tau_k \prec \tau_l$.

Definition (start task) *A start task of* $(T_i, \prec)$ *is such a task* $\tau_j \in T_i$ *that starts earliest of all the tasks in* T_i*, that is,* $\forall \tau_k \in T_i, j \neq k \Rightarrow \tau_j \prec \tau_k$.

Definition (end task) *An end task of* $(T_i, \prec)$ *is such a task* $\tau_j \in T_i$ *that starts latest of all the tasks in* T_i*, that is,* $\forall \tau_k \in T_i, j \neq k \Rightarrow \tau_k \prec \tau_j$.

Without losing generality, for a task poset $(T_i, \prec)$, it is assumed that there is one start task (denoted by τs_i) and one end task (denoted by τe_i)[2]. Each task has two parameters, $\tau_j = (c_j, m_j)$, where j is the index of the task. c_j is the required computation cost which means the number of time slots needed by a unit speed processor to complete task τ_j (can be analogous to task execution time e defined in Chapter 3); and m_j is the required migration cost for task τ_j migrating from a processor to another one. The parameter m_j combined with parameters of network (details will be explained later) is used to calculated the overhead of migrating task.

4.2.3 Processor Set

In multiprocessor real-time systems, processors are used to execute tasks. Symbol $\mathcal{P} = \{p_1, p_2, \ldots, p_l\}$ is used to denote the set of processors, where l is the number of processors. Each processor p_a is a 2-tuple, $p_a = (sp_a, TS_a)$, where a is the index of the processor.

[1]Based on this task poset model, task preemption is not supported. To support task preemption, we can define a task consisting of a series of indivisible fragments (refer to Chapter 3 for more details).

[2]To express a function with many start (end) tasks, we can set a virtual task to be a new start (end) task. Such a special task contains empty operations and starts before (after) all the original start (end) tasks.

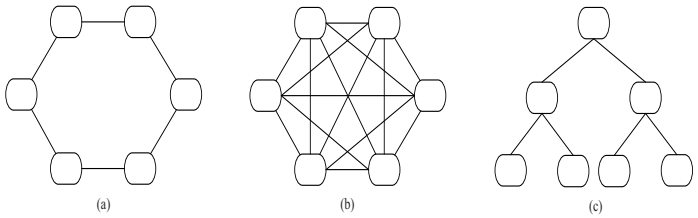

Figure 4.1: Topologies of network channel: (a) Ring, (b) Mesh, (c) Tree

Parameter sp_a is the speed of the processor. When task τ_i running on processor p_a, the number of time slots needed for processor p_a to complete task τ_i, represented by *task completion* tc_a^i:

$$tc_a^i = \frac{c_i}{sp_a} \tag{4.1}$$

TS_a is the task set that can be completed by processor p_a. This parameter is for *heterogeneous* systems in which processors have different architectures, and some tasks can only be executed on some specific processors. If $TS_a = \emptyset$, it means processor p_a cannot be used to execute any task in the system.

Note that, if we set processors for $\forall p_i \in \mathcal{P}$ having the same sp_a and TS_a, the systems will be simplified to identical real-time systems. Similarly, if we set processors for $\forall p_i \in \mathcal{P}$ having the same sp_a, the systems will be simplified to uniform real-time systems.

Processors have independent local clocks[3], they are synchronized with each other in the time domain through synchronization protocol. The maximum difference between the local clocks of any two processors in the networked systems is called *network precision* (also called synchronization jitter) which is a global constant defined in the system [31]. Symbol δ is used to denote the network precision.

4.2.4 Network Channel Set

In multiprocessor real-time systems, processors are connected through network channels. Symbol $\mathcal{N} \subseteq \mathcal{P} \times \mathcal{P}$ is used to denote set of the network channels. This network channel model is generic and can characterize different kinds of network topologies, e.g., bus, ring, mesh, and tree. Moreover, it makes RSMT support all the non-, intra-, fully-migrative scheduling manners as shown in Figure 4.1. Symbol $n_{a \to b} \in \mathcal{N}$ denotes the network

[3]In this research, clock acts as a metronome and is defined in discrete values.

channel from processor p_a to p_b, where $p_a, p_b \in \mathcal{P}, a \neq b$. Symbol $ns_{a\to b}$ represents speed of $n_{a\to b}$.

When the data of the computed results of task τ_i (or the signal issued by task τ_i when it is completed) migrates from processor p_a to p_b[4], the time slots spent on network channel, represented by $tm^i_{a\to b}$, can be calculated as:

$$tm^i_{a\to b} = \frac{m_i}{ns_{a\to b}} \tag{4.2}$$

Based on $tm^i_{a\to b}$, we can get the time instant that processor p_b receives the data of task τ_i migrating from processor p_a through network channel $n_{a\to b}$, represented by $r^i_{a\to b}$, as

$$r^i_{a\to b} = s^i_{a\to b} + tm^i_{a\to b} + \delta \tag{4.3}$$

where, $s^i_{a\to b}$ is the start time of τ_i migrating through network channel $n_{a\to b}$, and δ is the network precision.

Note that, through setting network channel set $\mathcal{N}$, we can specify desired scheduling manners such as non-, intra-, or fully-migrative scheduling. For example, if we want to design a non-migrative scheduling, we can set $\mathcal{N} = \emptyset$. For intra-migrative scheduling, we can set the network channels only existing among processors having the same TS. For fully-migrative scheduling, we can set the network channels existing among all the processors.

4.2.5 Advantage of Function Model

To show the advantage of the function model, an example is shown in Figure 4.2. In this example, a multiprocessor system is equipped with two processors p_1 and p_2 which have processing speed 1 and 2, respectively. There is no network channel connecting these two processors. Two tasks τ_1 and τ_2 with dependency relation $\tau_1 \prec \tau_2$ arrive in the system at $t = 0$. When assigning deadline to task τ_1, as both two processors p_1 and p_2 can execute these two tasks, based on the conventional task model used in Chapter 2, the deadline assignment is in a dilemma. For example, when assigning d_1 based on processor p_1, we can get $d_1 = 2$, while based on processor p_2, we can get $d_1 = 3$. As a task can only have

[4]For conciseness, we say "task τ_i migrates from processor p_a to p_b" to mean "the data of the computed results of task τ_i (or the signal issued by task τ_i when it is completed) migrates from processor p_a to p_b", hereafter.

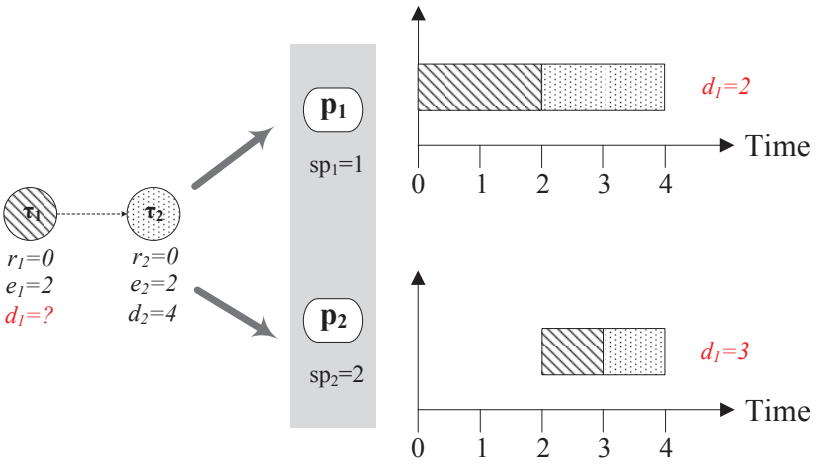

Figure 4.2: Deadline assignment when applying conventional task model

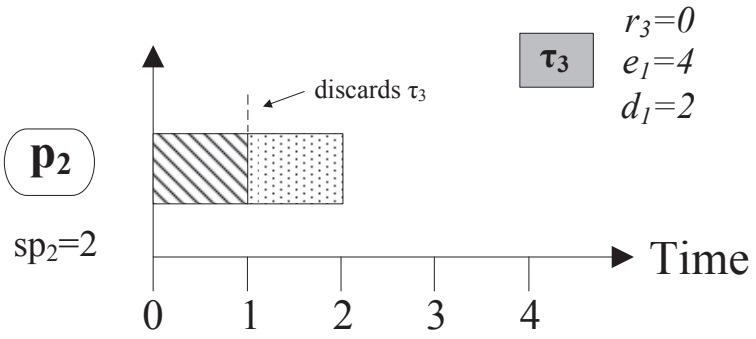

Figure 4.3: Possible execution scenario when assigning $d_1 = 2$

one deadline assignment, we need to make a choice from these two values. If $d_1 = 3$ is chosen, when these two tasks are executed on processor p_1, task τ_2 cannot be completed before its deadline when task τ_1 is completed at its deadline $t = 3$. Therefore, $d_1 = 3$ is not a proper deadline assignment. Now, we try $d_1 = 2$. This assignment will make time slot [2 3) cannot be utilized by task τ_1. A possible execution scenario shown in Figure 4.3 is used to illustrate this defect. Assume, in addition to task τ_1 and τ_2, task τ_3 also arrives in the system at $t = 0$, and processor p_2 is used to execute these three tasks. From the figure, we can see that, under the assignment of $d_1 = 2$, processor p_2 cannot make all the three tasks meet their deadlines. A possible execution sequence will discard τ_3.

Compared to the conventional task model, the proposed function model sets deadline to function level. Let's reuse function model to re-represent this execution scenario. As task τ_1 and τ_2 have dependency relation $\tau_1 \prec \tau_2$, these two tasks can be represented as one function $F_1 = ((T_1, \prec), 0, 4), T_1 = \{\tau_1, \tau_2\}, c_1 = 2, c_2 = 2$, while task τ_3 can be represented

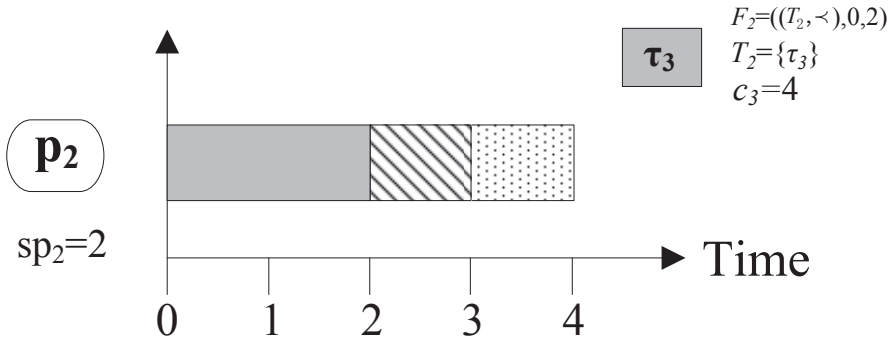

Figure 4.4: Performance by applying function model

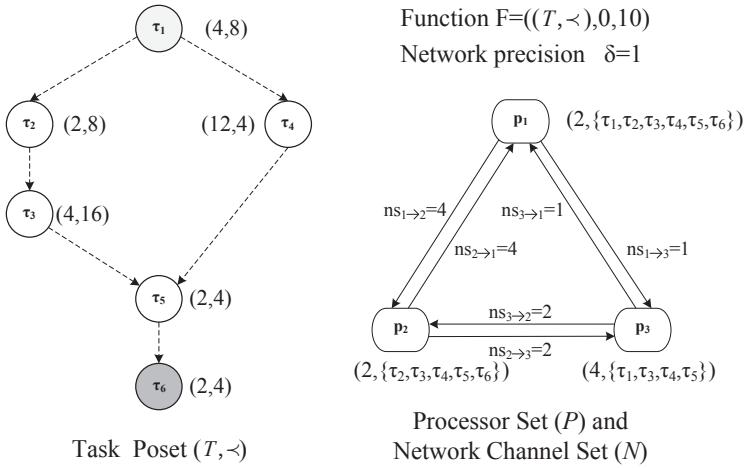

Figure 4.5: An example of scheduling for multiprocessor real-time systems

as function $F_2 = ((T_2, \prec), 0, 2), T_2 = \{\tau_3\}, c_3 = 4$. We can see that, as deadline is assigned to function, there is no deadline assignment dilemma when applying the function model. Figure 4.4 shows the execution sequence by applying the function model. Following the execution sequence, all the three tasks can be successfully completed. This shows advantage of the function model, and also reflects the fact that, for systems equipped with multiple different speed processors, the conventional task model is not capable to model tasks which have dependency relation with each other.

4.2.6 An Example

To illustrate the defined system model, an example of scheduling for multiprocessor real-time systems is shown in Figure 4.5. In this example, there are three processors p_1, p_2, p_3 in the system. These processors are connected with each other through six network channels

$n_{1\to2}, n_{2\to1}, n_{1\to3}, n_{3\to1}, n_{2\to3}, n_{3\to2}$. The network precision δ is 1. In the system, a function $F = ((T, \prec), 0, 10)$ is waiting to be executed on the processors. The task poset of the function is $(T, \prec)$ which comprises of six tasks. Task dependency relations are described in a directed acyclic graph. An edge starting from task τ_i to task τ_j represented by a dotted line denotes a dependency relation $\tau_i \prec \tau_j$.

4.3 Scheduling Constraints

This section describes all the constraints expressed in the SAT model.

System Constraints

4.3.1 Constraint on start execution time of functions

Task set T_i corresponding to function F_i can start to run only after the function is triggered. That is, the start execution time of the start task of the poset $(T_i, \prec)$ should be larger than or equal to the triggered time of function F_i.

$$\begin{aligned} &\forall F_i \in \mathcal{F}, \forall p_a \in \mathcal{P} \\ &s_a^{\tau s_i} \geq r f_i \end{aligned} \tag{4.4}$$

where symbol $s_a^{\tau s_i}$ denotes the start execution time of task τs_i on processor p_a.

4.3.2 Constraint on start time of task migration

If a task τ_i can migrate from processor p_a to processor p_b through network channel $n_{a\to b}$, it implies that *i):* task τ_i has been completed by processor p_a; or *ii):* τ_i has migrated to processor p_a from another processor. For the first case, task τ_i can start to migrate after it has been completed, and for the second case, task τ_i can start to migrate after it has already migrated to processor p_a.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T}, \forall n_{a\to b} \in \mathcal{N}, \exists n_{c\to a} \in \mathcal{N} \\ &(s^i_{a\to b} \geq s^i_a + tc^i_a) \vee (s^i_{a\to b} \geq r^i_{c\to a}) \end{aligned} \tag{4.5}$$

where symbol $s^i_{a\to b}$ denotes the start time of task τ_i migrating through network channel $n_{a\to b}$, s^i_a denotes the start execution time of task τ_i on processor p_a.

4.3.3 Constraint on task dependency

For processor p_a, if $\tau_i \prec \tau_j$, task τ_j can start to run only after τ_i has been completed. Similar to the constraints on start time of task migration, there are two cases. *i):* task τ_i has been completed by processor p_a; *ii):* task τ_i has migrated to processor p_a from another processor. For the first case, τ_j can start to run after τ_i has been completed, and for the second case, τ_j can start to run after τ_i has already migrated to processor p_a.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T}, \forall p_a \in \mathcal{P}, \exists n_{b\to a} \in \mathcal{N} \\ &\tau_i \prec \tau_j \Rightarrow (s_a^j \geq s_a^i + tc_a^i) \vee (s_a^j \geq r_{b\to a}^i) \end{aligned} \tag{4.6}$$

4.3.4 Constraint on processors

A processor can execute only one task at a time. This is interpreted as: there is no overlap of the execution time of any two tasks.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall p_a \in \mathcal{P} \\ &(s_a^i \geq s_a^j + tc_a^j) \vee (s_a^j \geq s_a^i + tc_a^i) \end{aligned} \tag{4.7}$$

4.3.5 Constraint on network channels

A network channel can transfer data of only one task at a time. That is, there is no overlap of the migration time of any two tasks on a network channel.

$$\begin{aligned} &\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall n_{a\to b} \in \mathcal{N} \\ &(s_{a\to b}^i \geq s_{a\to b}^j + tm_{a\to b}^j) \vee (s_{a\to b}^j \geq s_{a\to b}^i + tm_{a\to b}^i) \end{aligned} \tag{4.8}$$

4.3.6 Constraint on heterogeneous processors

In heterogeneous systems, processors have different architectures, some tasks can only be executed on some specific processors. For tasks that cannot be executed on some processors, the start execution time of the tasks in such processors are set to $+\infty$, which means the tasks will never start to run on these specific processors.

$$\begin{aligned} &\forall p_a \in \mathcal{P}, \forall \tau_i \in \mathcal{T} - TS_a \\ &s_a^i = +\infty \end{aligned} \tag{4.9}$$

Target Constraints

4.3.7 Constraint on deadlines of functions

For every triggered function, the desired schedule should make sure that the function can be completed before its deadline. For a hard deadline system, this requirement should be guaranteed throughout the time period during which the system is running. Developers should make such a guarantee when they design the system.

$$\begin{aligned} &\forall F_i \in \mathcal{F}, \exists p_a \in \mathcal{P} \\ &s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i \end{aligned} \tag{4.10}$$

where symbol $s_a^{\tau e_i}$ is the start execution time of task τe_i on processor p_a, and $tc_a^{\tau e_i}$ is the number of time slots needed for processor p_a to complete task τe_i.

4.4 Schedule Synthesis

After all the constraints are defined, we now can employ a SMT solver to generate the desired schedule. The process of schedule synthesis is summarized in Alg. 5. Function $\mathcal{A} := \texttt{Assert}(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N})$ (line 1) interprets the constraints defined in section 4.3 as *assertions* (boolean formulas that can be input into a SMT solver) based on the system model. The variables of these boolean formulas are the start time of task execution on processor, s_a^j, and start time of task migration through network, $s_{b\to c}^j$, for $\forall F_i \in \mathcal{F}, \forall \tau_j \in T_i, \forall p_a \in \mathcal{P}, \forall n_{b\to c} \in \mathcal{N}$. Function $\texttt{CallSMTSolver}(\mathcal{A})$ (line 2) calls a SMT solver to find a solution model $\mathcal{M}$ for $\mathcal{A}$, which contains interpretation for all the variables s_a^j and $s_{b\to c}^j$ defined in the set of assertions $\mathcal{A}$. If the solution model does not exist ($\mathcal{M} = \emptyset$) (line 3), message `UNFEASIBLE` will be returned (line 4), which means there does not exist a schedule can make all the functions meet their deadlines. Otherwise, if the solution model exists ($\mathcal{M} \neq \emptyset$), based on $\mathcal{M}$, function $\texttt{GenSch}(\mathcal{M})$ generates the desired schedule $\mathcal{S}$ which can make all the functions meet their deadlines (line 6).

Alg. 6 describes details of function `GenSch`(). It returns the schedule $\mathcal{S}$ which is a set of variables selected from the solution model $\mathcal{M}$. $\mathcal{M}$ contains the interpretation for all the variables s_a^j and $s_{b\to c}^j$ defined in the set of assertions $\mathcal{A}$. As existential quantifier are used in formula 4.5, 4.6, and 4.10, we need to further select s_a^j and $s_{b\to c}^j$ that can form the desired schedule. For example, in formula 4.10, we should find which processor (i.e., to determine the value of processor index a) can make the logical formula $\forall F_i \in \mathcal{F}, s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i$

Algorithm 5 Schedule synthesis

Input: function set $\mathcal{F}$, task set $\mathcal{T}$, processor set $\mathcal{P}$, network set $\mathcal{N}$

Output: schedule $\mathcal{S}$

1: $\mathcal{A} :=$ `Assert`$(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N})$
2: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
3: **if** $\mathcal{M} = \emptyset$ **then**
4: **return** `UNFEASIBLE`
5: **end if**
6: **return** $\mathcal{S} :=$ `GenSch`$(\mathcal{M})$

Algorithm 6 `GenSch`()

Input: solution model $\mathcal{M}$

Output: schedule $\mathcal{S}$

1: $\mathcal{S} := \emptyset$
2: **for all** $F_i \in \mathcal{F}$ **do**
3: sort task set T_i, such that $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is a permutation with $\tau_{j_1} \not\prec \tau_{j_2}$ for $1 \leq j_1 < j_2 \leq n$, where n is the number of tasks in T_i
4: determine s_a^1 based on formula 4.10, $\mathcal{S} := \mathcal{S} \cup \{s_a^1\}$
5: **for all** $\tau_j \in T_i \backslash \tau_1$ **do**
6: determine s_a^j based on formula 4.5 and 4.6
$\mathcal{S} := \mathcal{S} \cup \{s_a^j\}$
7: **if** τ_j has migrated among processors **then**
8: determine $s_{b\to c}^j$ based on formula 4.5 and 4.6
$\mathcal{S} := \mathcal{S} \cup \{s_{b\to c}^j\}$
9: **end if**
10: **end for**
11: **end for**

be evaluated as `true`. After this judgment, we can determine s_a^j and $s_{b\to c}^j$ that can form the desired schedule. For each function F_i, `GenSch`() first sorts its corresponding task set T_i, such that $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is a permutation with $\tau_{j_1} \not\prec \tau_{j_2}$ for $1 \leq j_1 < j_2 \leq n$ (line 3). Follow this sorted order, the first task τ_1 is the end task τe_i of function F_i. Based on formula 4.10, the value of processor index a can be determined. That is, we can determine s_a^1 (i.e., $s_a^{\tau e_i}$) (line 4). For other tasks in T_i, as $\tau_{j_1} \not\prec \tau_{j_2}$ for $1 \leq j_1 < j_2 \leq n$, after the value of s_a^1 is determined, based on formula 4.5 and 4.6, the values of s_a^j and $s_{b\to c}^j$ (only for tasks which migrates among processors) can be determined (line 5-10). After this, the desired schedule $\mathcal{S}$ can be generated. Notice that, since solution model $\mathcal{M}$ gives interpretations of all the variables defined in the SAT model which can satisfy all the formulas contained in the SAT model, function `GenSch`() can always generate the desired schedule $\mathcal{S}$.

From the schedule synthesis, we can know that RSMT is an *optimal* scheduling

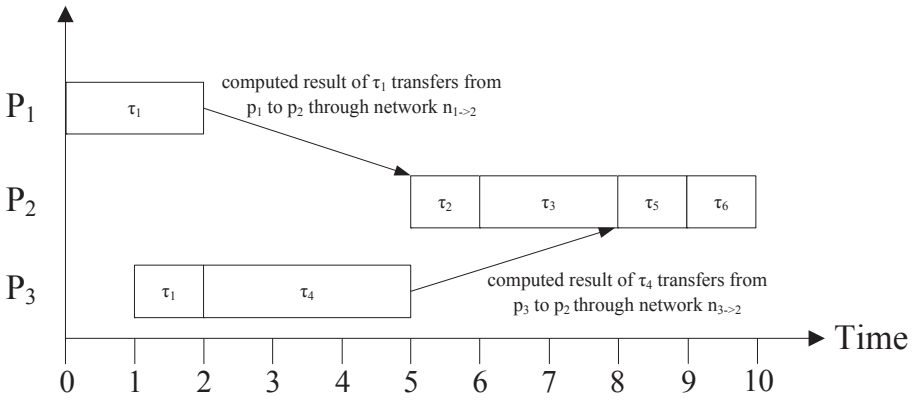

Figure 4.6: Scheduling result for example shown in Figure 4.5 by using RSMT

method[5], that is, if RSMT cannot make all tasks meet their deadlines (return `UNFEASIBLE`), no other scheduling methods can.

4.4.1 Scheduling Results

Recall the example shown in Figure 4.5. Based on the schedule synthesis described in Alg. 5, we can get the solution model $\mathcal{M}$ which defines the values of the start time of task execution on processor, s_a^j, and the start time of task migration through network, $s_{b\to c}^j$, for $\forall F_i \in \mathcal{F}, \forall \tau_j \in T_i, \forall p_a \in \mathcal{P}, \forall n_{b\to c} \in \mathcal{N}$. From the model $\mathcal{M}$, through function `GenSch`() as shown in Alg. 6, we can get the scheduling result as shown in Figure 4.6. It can be seen that, the scheduling sequence can make the function meet its deadline. Some characteristics of this scheduling sequence should be noticed:

- Task τ_1 has been executed on processor p_1 from system time $t = 1$ to $t = 3$, and it has also been executed on processor p_3 from system time $t = 2$ to $t = 3$. This means that RSMT can handle parallel execution of tasks, and can make a task repeatedly run on different processors when such repeated execution is necessary.
- Task τ_2 runs on processor p_2 from $t = 6$ to $t = 7$. Although task τ_2 needs the computed results obtained from completing task τ_1, such computed results not only can be obtained by completing task τ_1 on processor p_2 itself, but also be obtained by transferring the computed results from other processor that has completed task

[5]Only when all the scheduling constraints are defined in the SAT model. More details will be discussed in section 6.2.

τ_1. Specified to this example, at system time $t = 6$, processor p_2 gets the computed results of task τ_1 from processor p_1.

4.5 Simulation, Evaluation, and Limitation

In this section, results of the simulations which are conducted to study the performance of RSMT are presented. A prototype tool for RSMT, based on the system model, constraint formulation, and schedule synthesis described above, has been implemented. The underlying SMT solver employed by the tool is Z3.

4.5.1 Simulation Settings

In order to evaluate the time that RSMT spends on scheduling all the input tasks, the simulation results of the SMT solver *computing* time and the *total spent* time (including interpreting, computing, and generating time) of RSMT are calculated in the simulation studies.

The input functions are generated according to Poisson distribution with arriving rate λ which represents number of the functions that arrive in the system per 100 time units. The number of the tasks within a function varies in [1, 3]. The dependency relation of the tasks is randomly assigned. For each task τ_j, c_j and m_j vary in [1, 7] with exponential distribution. The deadline assignment of a function df_i is according to the formula: $df_i = rf_i + \lfloor sf_i * C_i \rfloor$, where C_i is total computation cost of all the tasks contained in the function, and sf_i is slack factor that indicates the tightness of task deadline. For each function, sf_i varies in [1, 3]. For the network channels, a mesh topology (all the processors are connected with each other through network channels) is considered in the simulation, and the network speed varies in [1, 6]. For the processor set, the speed of processors varies in [1, 6], and the task sets which can be executed on specific processors are randomly assigned.

There are many parameters that can affect the time that RSMT spends on scheduling input tasks. I mainly study the impacts of the number of functions, number of processors, and the parameter λ. The time-out value of SMT solver computing time is set to 1 hour after which the scheduling problem is deemed unfeasible. All the simulations run on a

64bit 4-core 2.5 GHz Intel Xeon E3 PC with 32GB memory.

4.5.2 Evaluation

The simulation results for the performance of RSMT are shown in Figure 4.7, 4.8, 4.9, and 4.10. The values shown in the figures are the average values of running simulation 100 times.

For systems with four processors, the simulation results are shown in Figure 4.7 and Figure 4.8. When $\lambda \geq 20$, the underlying SMT solver Z3 returns the result `unsat`, which means systems are under overload condition with $\lambda \geq 20$.

From Figure 4.7, we can see that with increasing value of λ and function number, the value of the computing time increases. This is because, with bigger value of λ, more functions (means more tasks) wait to be executed on processors at a time. As described in section 4.3, the scheduling constraints *Constraint on processors* and *Constraint on network channels*, guarantees that no overlap of the execution time of any two tasks on a processor and no overlap of the migration time of any two tasks on a network channel, respectively. The increasing number of waiting tasks will make the underlying SMT solver need much more calculation to return a solution model. Similarly, with increasing value of function number, the total number of tasks that need to be scheduled is increasing. This will also increase the calculation time for the underlying SMT solver to return a solution model.

For total spent time of RSMT, Figure 4.8 shows that RSMT needs less than 1 minute to generate the desired scheduling table, which means RSMT is capable to handle the scheduling problem for systems with four processors under the simulation settings.

For systems with five processors, the simulation results are shown in Figure 4.9 and Figure 4.10. When $\lambda \geq 26$, function number = 200, the simulation is time-out, which means the scheduling problem is too complicated for the underlying SMT solver to return a solution model within the limit of time (within 1 hour).

Comparing results shown in Figure 4.7 and Figure 4.9, it can be seen that the problem of scheduling systems with 5 processors is much more complicated than scheduling systems with 4 processors. This is denoted by the results that, under the same values of λ and function number, e.g., $\lambda = 18$, function number = 200, the underlying SMT solver

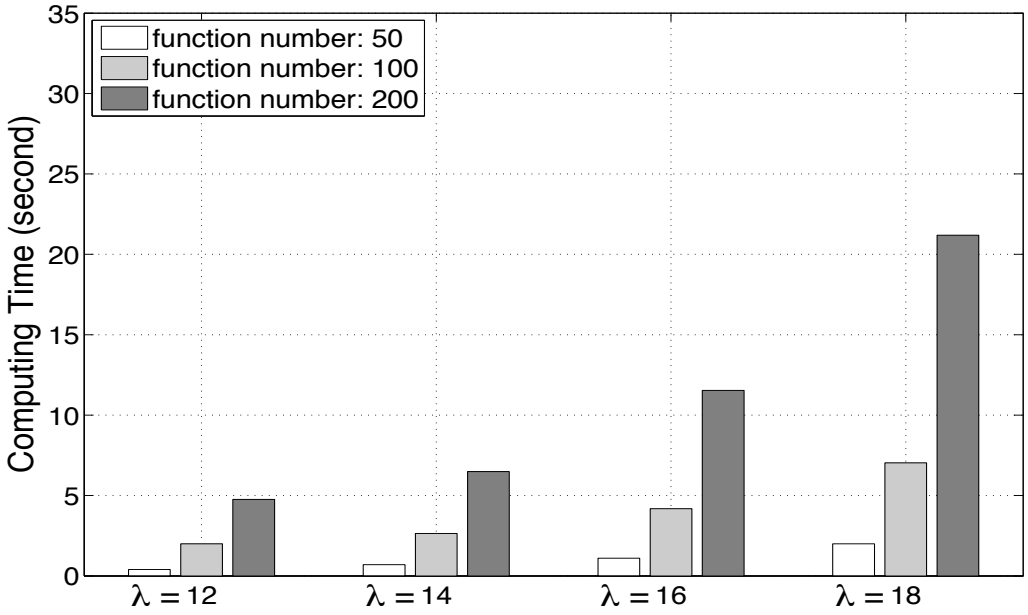

Figure 4.7: SMT solver computing time for system with 4 processors

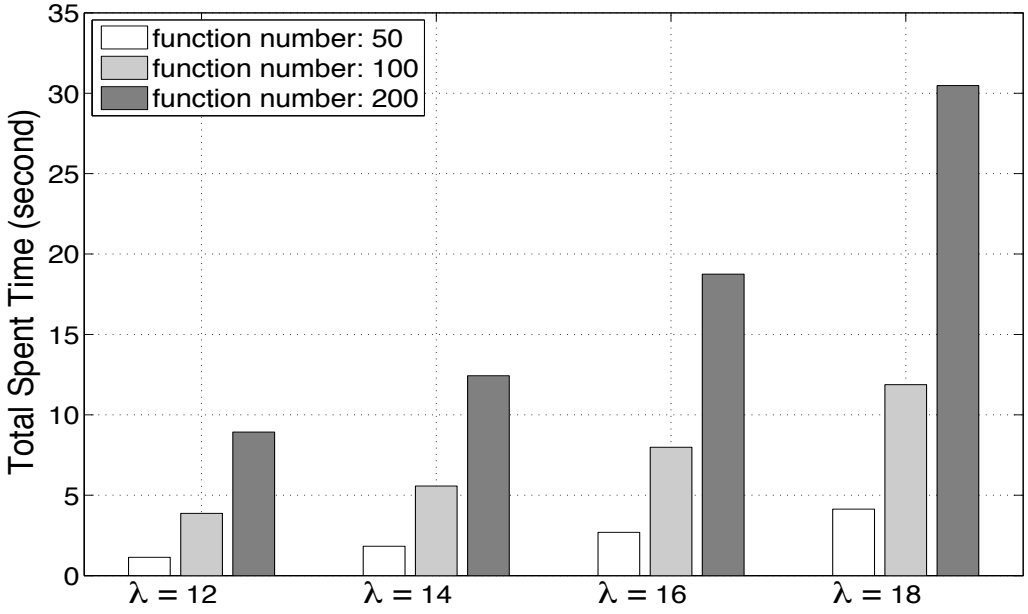

Figure 4.8: Total spent time of RSMT for systems with 4 processors

needs much more time to return a solution model for systems with 5 processors (around 200 seconds) than systems with 4 processors (around 20 seconds). Moreover, from Figure 4.9 and Figure 4.10, we can see that, the computing time occupies vast majority of

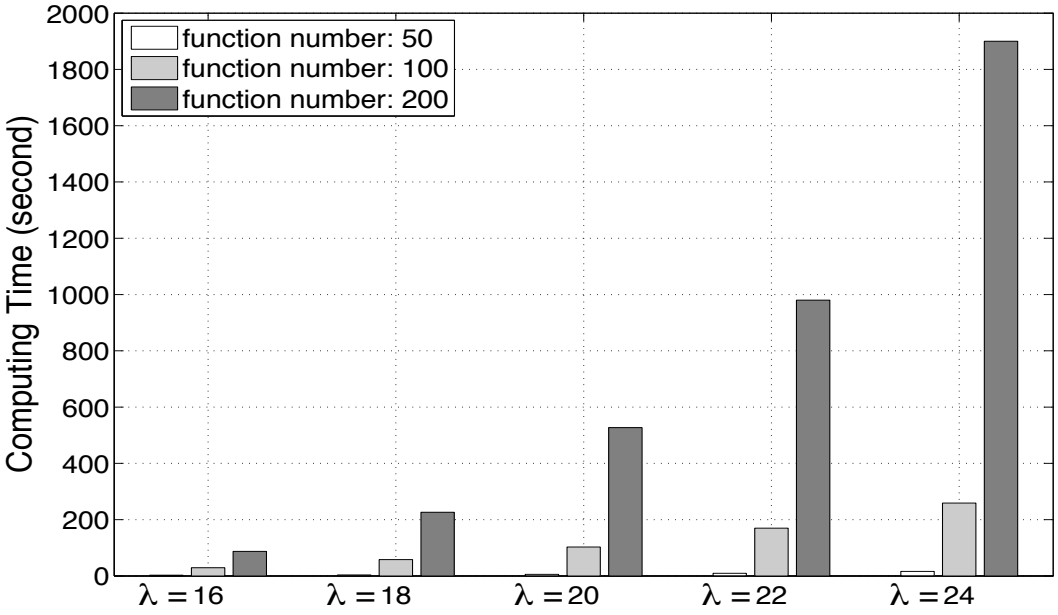

Figure 4.9: SMT solver computing time for system with 5 processors

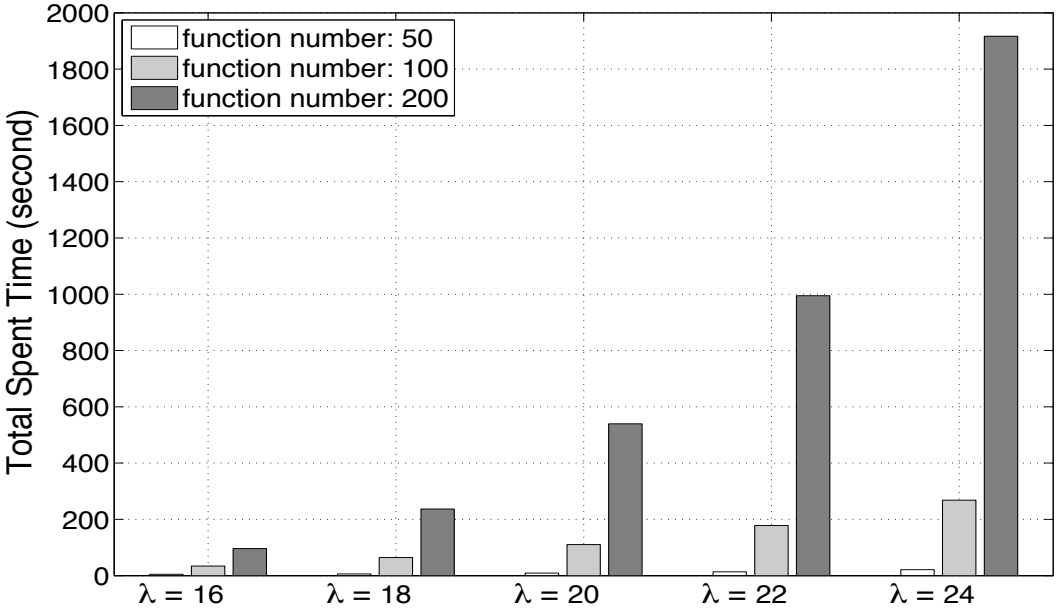

Figure 4.10: Total spent time of RSMT for systems with 5 processors

the total spent time. This phenomenon is much more obvious for scheduling systems with 5 processors than scheduling systems with 4 processors. This confirms the explanation described in section 2.7 that compared to the operation of the interpreting and

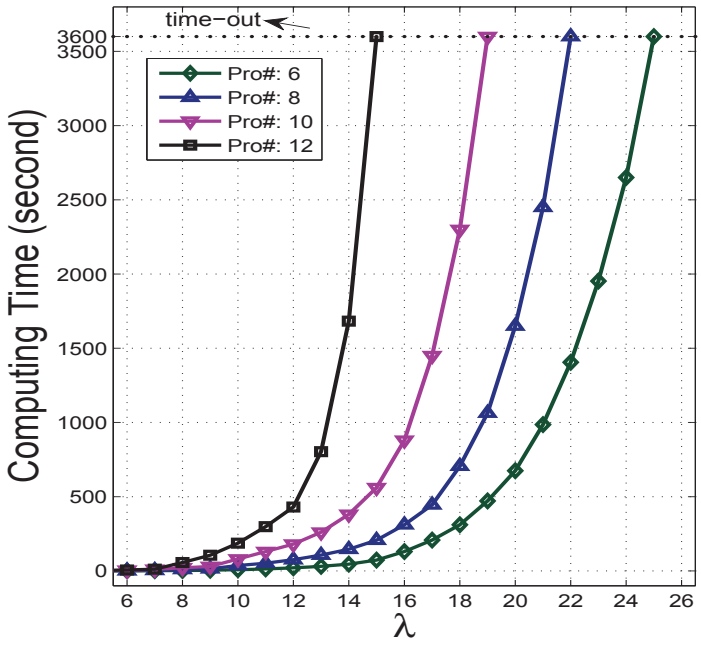

Figure 4.11: Applied scope of RSMT (migrative scheduling)

the generating, the operation of the computing is much more complication (exponential complexity) and becomes major limitation of RSMT. In the next subsection, the applied scope of RSMT is given, and some methods which can expand the scope are discussed.

4.5.3 Applied Scope

In the simulation studies, network channel with mesh topology is considered. Under this setting, tasks are allowed to migrate among all the processors equipped in systems. Such scheduling manner is called *migrative scheduling* (more precisely speaking, fully-migrative scheduling). Under the simulation settings, the scope in which RSMT can be applied is shown in Figure 4.11. Symbol *Pro#* in the figure means processor number. Note that, the results are obtained with function number = 200, which means with function number = 100 or 50, the applied scope will be larger.

From the figure, we can see that, with increasing number of processor, the underlying SMT solver becomes easier to time-out. For example, when processor number = 6, the SMT solver becomes time-out when $\lambda \geq 25$, while when processor number = 12, the SMT solver becomes time-out when $\lambda \geq 15$.

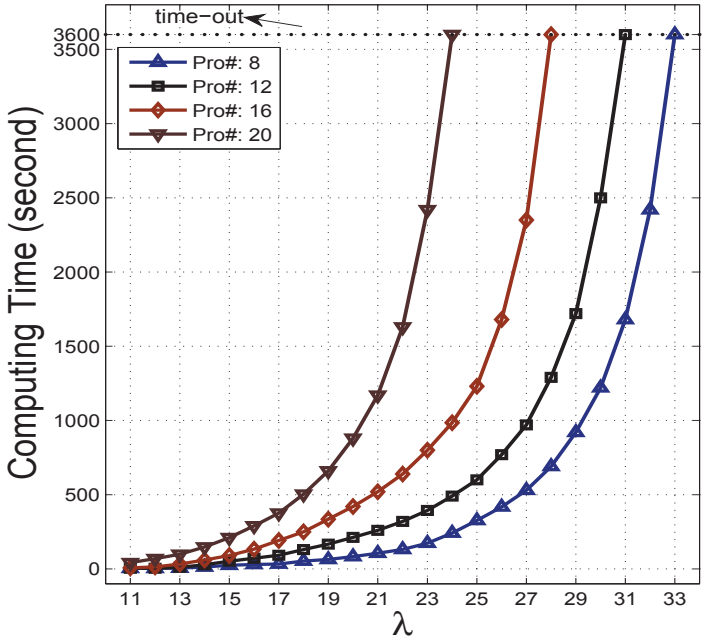

Figure 4.12: Applied scope of RSMT (non-migrative scheduling)

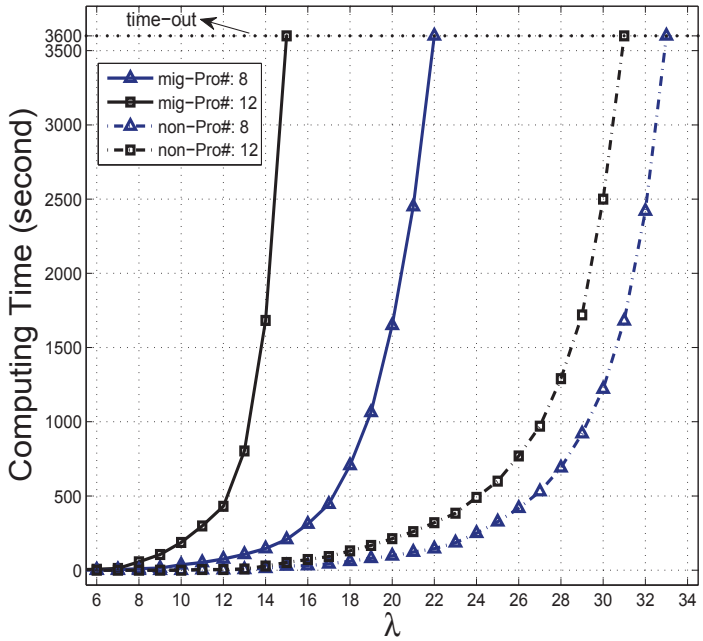

Figure 4.13: Applied scope of RSMT (processor number: 8, 12)

Non-migrative Scheduling

In addition to migrative scheduling, some systems do not allow tasks migrating among processors. Such scheduling manner is called *non-migrative scheduling*. To give the limitation of RSMT in such scheduling manner, in the simulation setting, the network channel

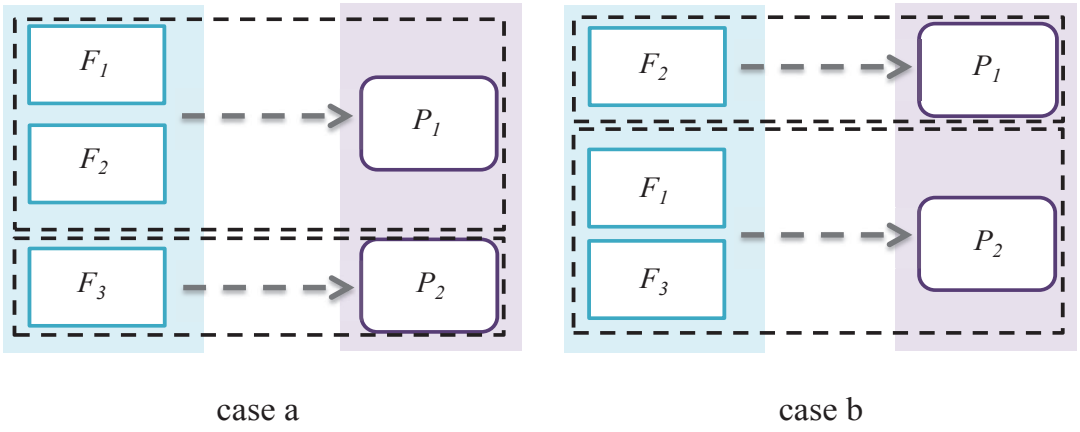

Figure 4.14: Different division

set is set as $\emptyset$. Under this settings, the scope in which RSMT can be applied is shown in Figure 4.12. Symbol *Pro#* in the figure means processor number. Compared to the applied scope for migrative scheduling, we can see that the applied scope of RSMT in non-migrative scheduling manner is much larger than it in migrative scheduling manner. In order to show this comparison result more clearly, in Figure 4.13, the applied scope of RSMT in both migrative and non-migrative scheduling manners for processor number: 8 and 12 are shown together. Symbol *mig-Pro#* and *non-Pro#* in the figure mean processor number in migrative scheduling and non-migrative scheduling, respectively. This is because, when adopting non-migrative scheduling manner, the scheduling constraint *Constraint on network channels* does not need to be considered, which will reduce a lot of calculation for the underlying SMT to return a solution model, consequently the *computing time* of RSMT will reduce.

Divide-And-Conquer

The applied scopes show the limitation of RSMT. As mentioned in section 2.7, such limitation is mainly because the computing operation of RSMT has exponential complexity. When a system configuration is beyond the scope, RSMT cannot be directly applied. In order to overcome such limitation to a certain extent, the idea *divide-and-conquer*[6] sheds some light.

[6]The basic idea of divide-and-conquer is to recursively break down a problem into sub-problems of the same or related type, until these sub-problems become simple enough to be solved directly. The solutions for the sub-problems are then combined to give a solution for the original problem [80].

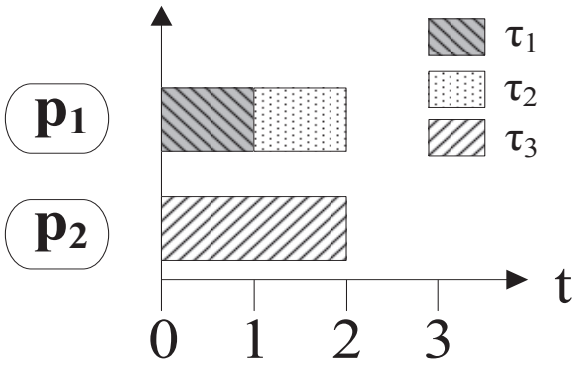

Figure 4.15: Scheduling result under the case without division (same as the result under the division case `a` in Figure 4.14)

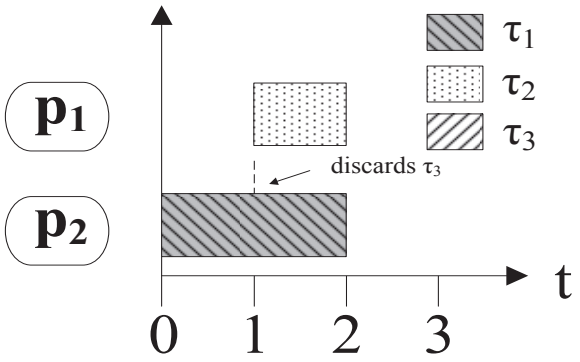

Figure 4.16: Scheduling result under the division case `b` in Figure 4.14

Considering a scenario: three functions are to be executed on two processors. The three functions are: $F_1 = ((T_1, \prec), 0, 2), T_1 = \{\tau_1\}, c_1 = 2$; $F_2 = ((T_2, \prec), 1, 2), T_2 = \{\tau_2\}, c_2 = 2$; $F_3 = ((T_3, \prec), 0, 2), T_3 = \{\tau_3\}, c_2 = 2$, and the two processors are: $p_1 = (2, \{\tau_1, \tau_2\})$, $p_2 = (1, \{\tau_1, \tau_3\})$. If we directly apply RSMT to design scheduling for this scenario, the problem complexity is to schedule three functions onto two processors. Under this case, the scheduling result is shown in Figure 4.15. We can see that, all the three functions can be successfully completed before their deadlines.

Now, we borrow the idea divide-and-conquer. Since task τ_1 can be executed on both processor p_1 and p_2, there are two possible divisions for this scenario, which are shown in Figure 4.14. Under the division case `a`, by applying RSMT, we can get the scheduling result as shown in Figure 4.15, which is the same as the result when directly apply RSMT without division. But the problem complexity becomes scheduling two functions onto one processor plus scheduling one function onto one processor.

From the scheduling constraints *Constraint on processors* which guarantees that there is no overlap of the execution time of any two tasks on a processor, we can know that the problem complexity after division becomes less than it without division. This means some scheduling problems that cannot be solved by directly applying RSMT can be solved by RSMT after division, in other words, the applied scope of RSMT becomes larger after problem division.

However, under the division `b`, by applying RSMT, we can get the scheduling result as shown in Figure 4.16. Under this division, function F_3 cannot be successfully complete before its deadline. This is because, after problem division, the solution space for the underlying SMT solver to return a solution model is reduced, which means RSMT may not be able to find the optimal result as it can find without division.

From this example, we can see that the idea *divide-and-conquer* can, on the one hand, expand the applied scope of RSMT by reducing problem complexity, on the other hand, it will reduce the solution space for the underlying SMT solver to return a solution model, which may result in a non-optimal solution. Further research on how to avoid this disadvantage is an important future work.

4.6 Adapting to Other Scheduling Targets

There are many targets to be considered when designing scheduling for real-time systems. Which objectives are appropriate in a given situation depends, of course, upon the application. In this section, the design guidelines for adapt RSMT to other different scheduling targets are given.

4.6.1 Maximizing obtained values of completed functions

For a firm or soft real-time system, under normal workload conditions, there exists a schedule that can make all the triggered functions meet their deadlines. However, as mentioned in Chapter 3, due to dynamic changes of work environment, in practical environment, system workload may vary widely. Once system workload becomes too heavy so that there does not exist a feasible schedule can make all the functions meet their deadlines, we say the system is *overloaded*. When system is overloaded, one reasonable

scheduling target is to maximize the obtained values of the completed functions. For this scheduling target, a mixed critical system (system with both firm and soft deadline functions) is considered. Note that, scheduling soft (firm) real-time systems can be treated as a special case of scheduling mixed critical systems under which the set of firm (soft) deadline functions contained in the systems is an empty set. This means RSMT can also be applied to firm and soft real-time systems.

Constraints on scheduling target

For this scheduling target, a function with firm deadline is defined as $F_i = ((T_i, \prec), rf_i, df_i, v_i)$, where v_i is the value that system can obtain by completing the function F_i before its deadline df_i. If such a function misses its deadline, it will be useless to the system. Symbol $\mathcal{FH}$ is used to denote the set of firm deadline functions contained in the system. Let symbol v be the obtained values of the completed functions, and its initial value is set to 0. The value that system can obtain by completing the firm deadline functions can be calculated as follows.

$$\begin{aligned} &\forall F_i \in \mathcal{FH} \\ &\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i \\ &\quad v := v + v_i \\ &\texttt{end} \end{aligned} \tag{4.11}$$

For a function with soft deadline, it may still be useful to the system even it has missed deadline. Thus, without losing generality, a soft deadline function is defined as $F_i = ((T_i, \prec), rf_i, df_i, f_i(t))$, where $f_i(t)$ is the coefficient function to indicate the value that the system can obtain by completing function F_i at system time instant t. Symbol $\mathcal{FS}$ is used to denote the set of soft deadline functions contained in the system. The values that system can obtain by completing the soft deadline functions can be calculated as:

$$\begin{aligned} &\forall F_i \in \mathcal{FS} \\ &\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} < +\infty \\ &\quad v := v + f_i(s_a^{\tau e_i} + tc_a^{\tau e_i}) \\ &\texttt{end} \end{aligned} \tag{4.12}$$

Note that, the inequality $s_a^{\tau e_i} < +\infty$ denotes that function F_i has been completed in processor p_a.

Algorithm 7 Schedule synthesis for maximizing obtained values of completed functions

Input: function set $\mathcal{F}$, task set $\mathcal{T}$, processor set $\mathcal{P}$, network set $\mathcal{N}$

Output: schedule $\mathcal{S}$

```
1: A := Assert(F, T, P, N, max)
2: M := CallSMTSolver(A)
3: if M ≠ ∅ then
4:     return S := GenSch(M)
5: end if
6: start := 0, end := max
7: while true do
8:     mid := start + (end − start)/2
9:     A := Assert(F, T, P, N, mid)
10:    M := CallSMTSolver(A)
11:    if M = ∅ then
12:        end := mid − step
13:    else
14:        A := Assert(F, T, P, N, mid + step)
15:        M' := CallSMTSolver(A')
16:        if M' ≠ ∅ then
17:            start := mid + step
18:        else
19:            return S := GenSch(M)
20:        end if
21:    end if
22: end while
```

Let symbol sv denote the maximum obtained values of the completed functions, and obviously sv is no less than 0 and no larger than the sum of the values of the firm deadline functions ($\sum v_i$ for $\forall F_i \in \mathcal{FH}$) and the values of the soft deadline functions when completed before deadlines ($\sum f_i(t_i)$, for $\forall F_i \in \mathcal{FS}$, where t_i is the completed time instant of task τ_i, and $t_i < df_i$). Symbol `max` is used to represent the sum of the values. The constraints on the scheduling target can be expressed as:

$$v \geq sv \tag{4.13}$$

Schedule Synthesis

Similar to the schedule synthesis described in Chapter 3 for overload problem of the uniprocessor systems, the process of schedule synthesis is summarized in Alg. 7.

Function `Assert`$(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N}, \texttt{max})$ (line 1) interprets the system constraints defined in section 4.3 and target constraints descried in section 4.6.1 as assertions with $sv :=$ `max` in formula 4.13. If this expectation can be satisfied, it means overload problem does not

happen, model $\mathcal{M}$ will be returned by function `CallSMTSolver`($\mathcal{A}$) (line 2). Based on $\mathcal{M}$, schedule $\mathcal{S}$ can be generated by function `GenSch`() (line 4), where function `GenSch`($\mathcal{A}$) is described in Alg. 6.

When overload problem happens, if we set $sv :=$ `max`, constraint on scheduling target cannot be satisfied. We need to decrease the set value of sv. Binary search is used to find the maximum value of sv (line 6–22). With the maximum value of sv, a solution model can be returned by function `CallSMTSolver`($\mathcal{A}$). Meanwhile, with $sv := sv + step$, `CallSMTSolver`($\mathcal{A}$) will return an empty model. This is the criterion to judge if the value of sv is the maximum value. Note that, variable *step* is predefined and can be used to control the search space of SMT solver. Increasing *step* makes the algorithm faster but also reduce the solution space.

4.6.2 Making firm deadline functions meet deadlines first

Constraint on scheduling target

Since firm deadline functions usually play important roles in a real-time system, when the system is under overload condition, a reasonable scheduling target is to first make sure that all the firm deadline functions meet their deadlines, then maximize obtained values of the completed soft deadline functions. To make firm deadline functions meet deadlines, we can get

$$\begin{aligned} &\forall F_i \in \mathcal{FH}, \exists p_a \in \mathcal{P} \\ &s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i \end{aligned} \tag{4.14}$$

To maximize the obtained value of the completed soft deadline functions, the formula is similar to it for the previous scheduling target. Let symbol v be the obtained values of the completed functions, and its initial value is set to 0. The values that the system can obtain by completing the soft deadline functions can be calculated as:

$$\begin{aligned} &\forall F_i \in \mathcal{FS} \\ &\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} < +\infty \\ &\quad v := v + f_i(s_a^{\tau e_i} + tc_a^{\tau e_i}) \\ &\texttt{end} \end{aligned} \tag{4.15}$$

Algorithm 8 Schedule synthesis for making firm deadline functions meet deadlines first

Input: function set $\mathcal{F}$, task set $\mathcal{T}$, processor set $\mathcal{P}$, network set $\mathcal{N}$

Output: schedule $\mathcal{S}$

1: $\mathcal{A} :=$ `Assert`$(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N},$ `max_soft`$)$
2: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
3: **if** $\mathcal{M} \neq \emptyset$ **then**
4: **return** $\mathcal{S} :=$ `GenSch`$(\mathcal{M})$
5: **end if**
6: $start := 0, end :=$ `max_soft`
7: **while** `true` **do**
8: $mid := start + (end - start)/2$
9: $\mathcal{A} :=$ `Assert`$(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N}, mid)$
10: $\mathcal{M} :=$ `CallSMTSolver`$(\mathcal{A})$
11: **if** $\mathcal{M} = \emptyset$ **then**
12: $end := mid - step$
13: **else**
14: $\mathcal{A} :=$ `Assert`$(\mathcal{F}, \mathcal{T}, \mathcal{P}, \mathcal{N}, mid + step)$
15: $\mathcal{M}' :=$ `CallSMTSolver`$(\mathcal{A}')$
16: **if** $\mathcal{M}' \neq \emptyset$ **then**
17: $start := mid + step$
18: **else**
19: **return** $\mathcal{S} :=$ `GenSch`$(\mathcal{M})$
20: **end if**
21: **end if**
22: **end while**

Let symbol sv denote the maximum obtained values of the completed soft deadline functions, and obviously sv is no less than 0 and no larger than the values of the soft deadline functions when completed before deadlines ($\sum f_i(t_i)$, for $\forall F_i \in \mathcal{FS}$, where t_i is the completed time instant of task τ_i, and $t_i < df_i$), represented as `max_soft`. The constraints on scheduling target can be expressed as:

$$v \geq sv \tag{4.16}$$

Schedule Synthesis

The process of schedule synthesis is described in Alg. 8 which is quite similar to the process described in Alg. 7. The only modification is just replacing the `max` appearing in Alg. 7 as `max_soft` (line 1 and 6) in Alg. 8. This is because of the target constraint described in section 4.6.2 that only needs to maximize obtained values of the completed soft deadline functions as firm deadline functions has been guaranteed to be completed before their deadlines.

4.7 Summary

In this chapter, RSMT is applied to design scheduling for heterogeneous multiprocessor real-time systems. As scheduling identical and uniform multiprocessor systems can be treated as special cases of scheduling heterogeneous multiprocessor systems, it means RSMT can also be applied to schedule identical and uniform multiprocessor systems.

Various kinds (soft, firm, hard, and mixed critical) of systems and various types of scheduling manners (non-, intra-, fully-migrative scheduling) have been considered. Design guidelines for several practical requirements, and various kinds of scheduling targets that widely studied in real-time scheduling domain are given.

As mentioned in Chapter 3, dividing scheduling constraints into system constraints and target constraints makes RSMT flexible to design scheduling for other objectives by just modifying the target constraints. Design guidelines for three scheduling targets that are reasonable under the corresponding application scenarios are given. Adapting a system to different scheduling targets, little modification on target constraints and scheduling synthesis needs to make, and the system constraints can be totally reused.

Chapter 5

Use of Combining Offline and Online Scheduling

With increasing of system complexity, many real-time systems for industrial applications contain mixed sets of functions including both time-triggered functions (triggered by system clock) and event-triggered functions (triggered by event). Since events are usually initiated by outside environment in which systems operate, the time at which events are initiated is highly randomized. As mentioned in Chapter 1, RSMT performs in offline scheduling paradigm, it requires arriving time (i.e., triggered time) of functions known a priori. Such requirement limits the capability of RSMT to be applied to systems containing event-triggered functions. In order to overcome this limitation, a method of combining RSMT and online scheduling algorithm is given in this chapter. The target systems are heterogeneous multiprocessor real-time systems. Through this method, RSMT shows capability to design scheduling for systems containing event-triggered functions[1]. After the method is introduced, through a case study on a running car, the usage of this combination method in practical applications is shown. Moreover, from this case study, a method for generating task dependency relation based on SOFL (*structured-object-oriented-formal*) specification is also provided.

Organization of this chapter. In section 5.1, the method of combining offline and

[1]Event-driven systems can be treated as a special case of systems containing event-triggered functions, since all the functions contained in event-driven systems are event-triggered functions.

online scheduling is provided. To explain this method, section 5.1.1 introduces systems models and assumptions. Details of the method is described in section 5.1.2. Some discussions are conducted in section 5.1.3. To show the usage of the method, a case study on a running car is conducted in section 5.2. There are two parts included in the case study. First, through a detailed study on cruise control system, a method for generating task dependency relation based on SOFL specification is provided. Details of this method is described in section 5.2.1. After task dependency relation is generated, the combination method is applied to design scheduling for the running car. Details are described in section 5.2.2. At last, section 5.3 summarizes this chapter.

5.1 Combining Offline and Online Scheduling

5.1.1 System Model and Assumption

In order to model time-triggered and event-triggered functions, the idea from the conventional periodic and sporadic task models which are used to model time-triggered and event-triggered tasks [1] is borrowed. Let's first introduce the periodic and sporadic task models. In both task models, a task gives rise to a potentially infinite sequence of executions (called *instants* of task). In the periodic task model, task instants arrive strictly periodically, separated by a fixed time interval which is called the period of the periodic task. Compared to the periodic task model, in the sporadic task model, task instants may arrive at any time once a minimum inter-arrival time has elapsed since the arrival of the previous instant of the same task, this minimum inter-arrival time is called the period of the sporadic task.

Analogy to periodic and sporadic task models, the periodic and sporadic function models are given as follows. In both function models, a function gives rise to a potentially infinite sequence of executions (called *instants* of function). A periodic function set is denoted by $\mathcal{PF} = \{PF_1, PF_2, \ldots, PF_n\}$, where n is the number of periodic function. Each function $PF_i \in \mathcal{PF}$ is characterized by $PF_i = ((T_i, \prec), rd_i, p_i)$, where i is the function index, $(T_i, \prec)$, $T_i \neq \emptyset$ is the task poset, rd_i is the relative deadline, and p_i is the period. Task poset $(T_i, \prec)$ has the same definition as it used in the function model described in section 4.2. Meanwhile, relative deadline rd_i denotes that when a function

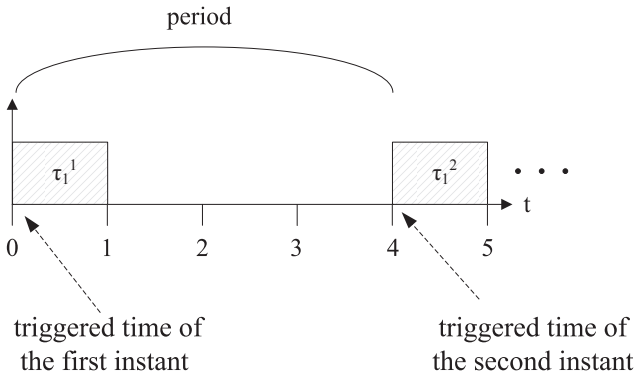

Figure 5.1: Periodic function PF_1

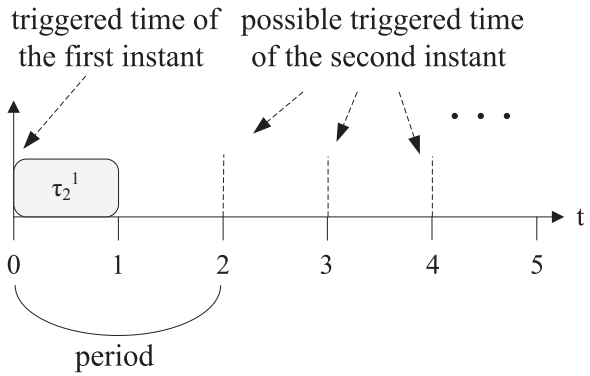

Figure 5.2: Sporadic function SF_2

instant PF_i^j is triggered at system time t, the specific time before which it required to be completed is $t + rd_i$, where symbol PF_i^j represents the j-th instant of function PF_i. Period p_i denotes that when a function instant PF_i^j is triggered at system time t, the next instant of the same function, represented by PF_i^{j+1}, will be triggered at $t + p_i$. Figure 5.1 shows a possible execution of a periodic function $PF_1 = ((T_1, \prec), 1, 4)$, $T_1 = \{\tau_1\}, c_1 = 1$, on a unit speed processor, where c_1 is the computation cost of τ_i. Symbol τ_i^j in the figure denotes the j-th instant of task τ_i (this notation is used hereafter). It shows that when the first instant of the function PF_1 is triggered at $t = 0$, the triggered time of the subsequent instant is determined ($t = 4$). This determinacy is the essential characteristic of time-triggered function.

Symbol $\mathcal{SF} = \{SF_1, SF_2, \ldots, SF_m\}$ is used to represent a sporadic function set, where m is the number of sporadic function. Note that, a function cannot be belong to both periodic function set and sporadic function set, i.e., $\mathcal{PF} \cap \mathcal{SF} = \emptyset$. Each function $SF_i \in \mathcal{SF}$ is characterized by $SF_i = ((T_i, \prec), rd_i, p_i)$, where i is the function index, $(T_i, \prec)$, $T_i \neq \emptyset$

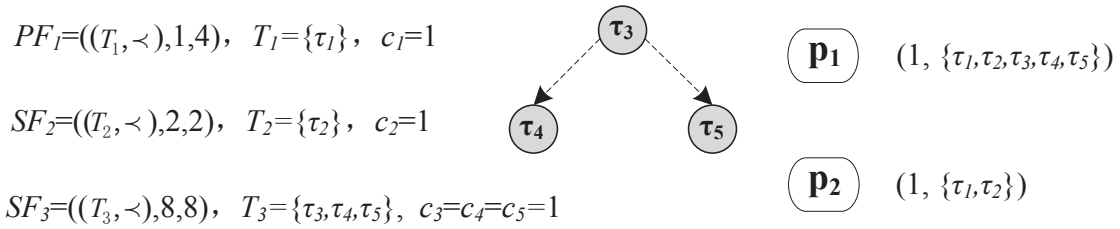

Figure 5.3: Example of system containing both periodic and sporadic function

is the task poset, rd_i is the relative deadline, and p_i is the period. The definition of task poset, relative deadline is the same as they are for periodic function model, and the only difference is the definition of period. In sporadic function model, period p_i denotes that when a function instant SF_i^j is triggered at system time t, the next instant of the same function, represented by SF_i^{j+1} can be triggered at any time at or after $t + p_i$. Figure 5.2 shows possible executions of a sporadic function $SF_2 = ((T_2, \prec), 2, 2)$, $T_2 = \{\tau_2\}, c_2 = 1$, on a unit speed processor. It shows that when the first instant of the function SF_2 is triggered at $t = 0$, the triggered time of the subsequent instant is not determined. The only information is that the triggered time of the subsequent instant will be later than $t = 2$. This non-determinacy reflects the randomness of event-triggered function.

Unlike periodic and sporadic task models that set deadline to task, in periodic and sporadic function models, deadline is set to function, while a function is achieved by a series of tasks cooperated together. The advantage of this setting, especially for scheduling heterogeneous multiprocessor systems, can refer section 4.2.5. In addition to periodic and sporadic function models, the task, processor, and network channel models proposed in Chapter 4 can be used to model the other parts of a heterogeneous multiprocessor system.

Based on these system models, it is assumed that the first instant of each periodic function is triggered at the time when system starts up (i.e., $t = 0$). The dependency relation among functions, task migration among processors, and the overload problem are not considered. Meanwhile, task preemption is allowed at any time.

5.1.2 Combination Method

When task migration is not considered, the procedure of scheduling multiprocessor real-time system can be divided into two phases. The first phase is to assign functions to processors, and the second phase is to allocate processor time slots to tasks. As in the

second phase, the allocating operation performs on each single processor, how to allocate is a problem of scheduling uniprocessor system. In the combination method, function-to-processor assignment conducts offline, while slot-to-task allocation performs online. To explain the combination method, with an example depicted in Figure 5.3, the method is described step by step. In the example, there are three functions: PF_1, SF_2, SF_3. Functions PF_1 and SF_2 are the corresponding functions used in the previous section, while the possible execution sequences of the sporadic function SF_3 on a unit speed processor is denoted in Figure 5.4.

Offline Function-to-processor Assignment

In the offline phase, for a sporadic function $SF_i = ((T_i, \prec), rd_i, p_i)$, due to the non-determinacy, the actually triggered time of function instants cannot be known. However, we do know its maximum triggered frequency, i.e., triggered once every p_i time units. In order to guarantee that all the functions (including both periodic and sporadic functions) can meet their deadlines, it is wondered that if there exists a worst-case arriving pattern. That is, if we can make sure that all the functions can meet their deadlines under the worst-case arriving pattern, regardless of when the sporadic functions are actually triggered, deadlines of all the functions can be guaranteed at system run-time. Fortunately, when scheduling uniprocessor system by online scheduling algorithm EDF, such worst-case arriving pattern does exist, i.e., all the sporadic functions triggered with their maximum frequency (**Lemma 1**, refer [81]). Some studied examples about such worst-case arriving pattern can also be found in [81]. Meanwhile, as mentioned in Chapter 3, EDF is an optimal scheduling algorithm on uniprocessor, in the sense that if there is any scheduling method can guarantee deadlines of all the functions in a function set, EDF also can (**Lemma 2**, refer [33]). Based on these results, the key idea of the combination method comes out.

Theorem. *If we can apply RSMT to decide function-to-processor assignment following which all the function deadlines can be guaranteed by RSMT under the worst-case arriving pattern, then applying EDF algorithm on each processor to decide slot-to-task allocation online can make sure that all the function deadlines can be guaranteed at system run-time.*

Proof. When functions are triggered following the worst-case arriving pattern, on each processor, as deadlines of the functions that are assigned to the processor can be guar-

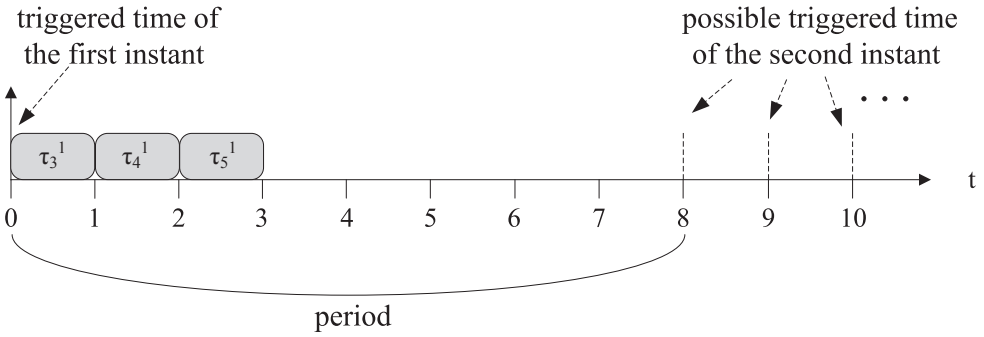

Figure 5.4: Sporadic function SF_3

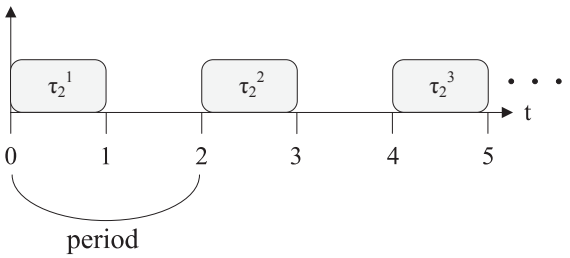

Figure 5.5: Sporadic function SF_2 triggered with maximum frequency

anteed by RSMT, according to Lemma 2, EDF also can guarantee the deadlines of the functions. Furthermore, according to Lemma 1, when using EDF to perform scheduling on uniprocessor, if deadlines of functions can be guaranteed under the worst-case arriving pattern, regardless of when the sporadic functions are actually triggered, deadlines of all the functions can be guaranteed at system run-time. □

Let's first assume that all the sporadic functions are triggered with their maximum frequency. Possible execution sequences of function SF_2 and SF_3 are shown in Figure 5.5 and 5.6, respectively. From these figures, it can be observed that if a sporadic function $SF_i = ((T_i, \prec), rd_i, p_i)$ is triggered with its maximum frequency, it behaves exactly like a periodic task with period p_i. For example, in Figure 5.5, when sporadic function $SF_2 = ((T_2, \prec), 2, 2)$ is triggered with its maximum frequency, i.e., triggered once every 2 time units, it behaves exactly like a periodic task $PF_2 = ((T_2, \prec), 2, 2)$. The similar result has also been observed in [82] (for conventional sporadic task model). Since under the worst-case arriving pattern, sporadic functions can be treated as periodic functions, and the triggered time of function instants can be known a priori, it means RSMT can be applied. We now adopt RSMT to generate scheduling table for the example shown in Figure 5.3 under the worst-case arriving pattern.

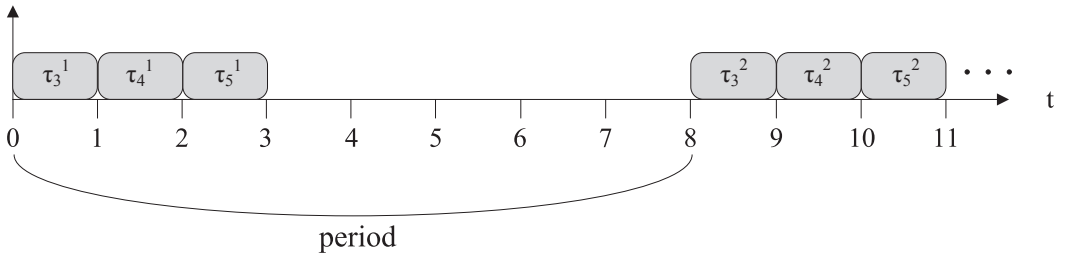

Figure 5.6: Sporadic function SF_3 triggered with maximum frequency

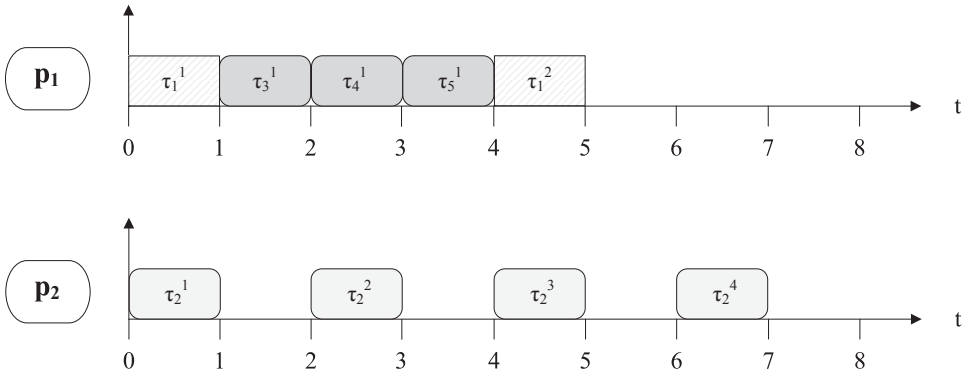

Figure 5.7: Scheduling table deciding function-to-processor assignment

From the periodic function model, it can be known that instant of a periodic function $PF_i = ((T_i, \prec), rd_i, p_i)$ is repeatedly triggered every p_i time units. Therefore, for a periodic function set $\mathcal{PF} = \{PF_1, PF_2, \ldots, PF_n\}$, when the first instants of all the functions are triggered at the same time (based on the system assumptions, the first instants of all the functions are triggered at $t = 0$), the triggered pattern of function instant will repeat every lcm time units, where lcm denotes the least common multiple of period p_i for all the functions in set $\mathcal{PF}$. This means, if all the function instants triggered within time interval [0 lcm) can meet their deadlines, the deadlines of all the future triggered function instants can also be guaranteed.

Specific to the example shown in Figure 5.3, as lcm equals to 8 ($p_1 = 4, p_2 = 2, p_3 = 8$), we only need to make sure that all the function instants triggered within time interval [0 8) can meet their deadlines. These function instants are: PF_1^1, PF_1^2, SF_2^1, SF_2^2, SF_2^3, SF_2^4, SF_3^1, where PF_i^j (SF_i^j) denotes the j-th instant of function PF_i (SF_i). As function instant can be modeled by task poset, triggered time, and deadline, e.g., PF_1^1 can be modeled as $((T_1, \prec), rf_i, df_i)$, where task poset $T_1 = \{\tau_1\}$, triggered time $rf_i = 0$, and deadline $df_i = 1$, function instant can be analogous to the "function" proposed in section

4.2. Moreover, as the task, processor, and network channel models (defined as $\emptyset$, as task migration is not considered) proposed in section 4.2 are used to model the other parts of the system, based on the method described in Chapter 4, we now can apply RSMT to generate scheduling table for the function instants PF_1^1, PF_1^2, SF_2^1, SF_2^2, SF_2^3, SF_2^4, SF_3^1. The result is shown in Figure 5.7.

The scheduling table shows two types of information. The first type of information is function-to-processor assignment. From the Figure 5.7, it can be seen that function PF_1 and SF_3 are assigned to processor p_1, while function SF_2 is assigned to processor p_2. The second type of information is slot-to-task allocation. That is, all the task instants have been allocated corresponding time slots. For example, the second instant of task τ_2, represented by τ_2^2, has been allocated time slot $[2\ \ 3)$ on processor p_2. It can be seen that, under the function-to-processor assignment, deadlines of the function instants can be guaranteed following the slot-to-task allocation.

For the information of slot-to-task allocation, since the scheduling table is specific to the system under the worst-case arriving pattern, when system really runs, sporadic functions may not be triggered with their maximum frequency, and the actually triggered time of sporadic functions can only be known at system run-time. This means the slot-to-task allocation denoted by the scheduling table cannot be used at system run-time, while it should be decided online.

Online Slot-to-task Allocation

According to the theorem proposed before, after the function-to-processor assignment is decided, we now can apply EDF algorithm on each processor to decide slot-to-task allocation online. By this way, all the function deadlines can be guaranteed at system run-time regardless of when the sporadic functions are actually triggered. Figure 5.8 shows the execution sequence generated by EDF in time interval $[0\ \ 8)$ on processor p_1 when the first instant of sporadic function SF_3 is triggered at $t = 2$.

Note that, the proposed combination method deals with the issue of deadline guarantee for event-driven systems. For other issues (e.g., scheduling overhead) that may be considered when we design scheduling for event-driven systems, how to properly use the combination idea needs further study.

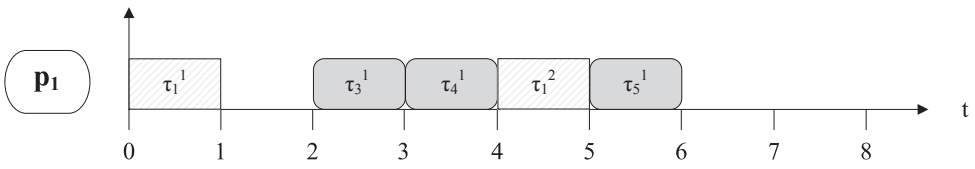

Figure 5.8: EDF deciding slot-to-task allocation (allocation on processor p_1)

5.1.3 Discussion

In the offline phase, in order to guarantee that all the functions can meet their deadlines, RSMT generates scheduling table based on the worst-case arriving pattern, i.e., all the sporadic functions triggered with their maximum frequency. As at system run-time, sporadic functions may not be triggered with their maximum frequency, under this case, processor resources that are reserved for the functions may be wasted. For example, as shown in Figure 5.7, processor p_2 is reserved for sporadic function SF_2. In an extreme case, function SF_2 may never be triggered, which means processor p_2 will never be used.

This kind of waste is mainly because we want to guarantee that all the functions can meet their deadlines. For hard real-time systems, since such guarantee is an essential requirement, this kind of waste cannot be avoided or reduced. But for other kinds of real-time systems (e.g., firm, soft, and mixed critical real-time systems) which allow functions missing deadlines occasionally, this kind of waste can be reduced to a certain extent.

For example, in the offline phase, when performing RSMT to generate the scheduling table, we may assume that some sporadic functions are not triggered with their maximum frequency. Specific to the example shown in Figure 5.3, if in the offline phase, we assume function SF_2 is triggered every 4 time units, only using processor p_1 can handle the execution of all the three functions. The result is shown in Figure 5.9. But of course, when the triggered frequency of SF_2 exceeds the assumed frequency, some function instants may miss deadlines. A possible scenario with a missed deadline function instant is depicted in Figure 5.10. In the possible scenario, function SF_2 is triggered every 2 time units within time interval [0 8). Under this triggered frequency, the function instant SF_2^4 will miss its deadline.

Based on above analysis and examples, we can see that reserving more resources can better guarantee deadline of function, while it may also result in more waste of resources. For firm, soft, and mixed critical real-time systems, in order to reduce possible waste, we

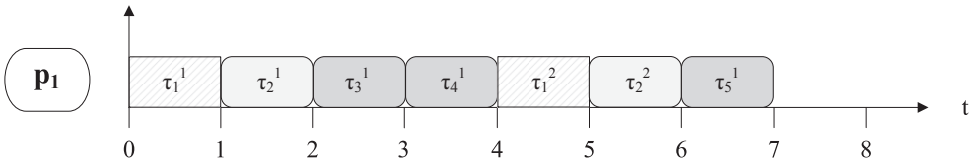

Figure 5.9: Scheduling table generated by RSMT (assuming sporadic function SF_2 triggered every 4 time units)

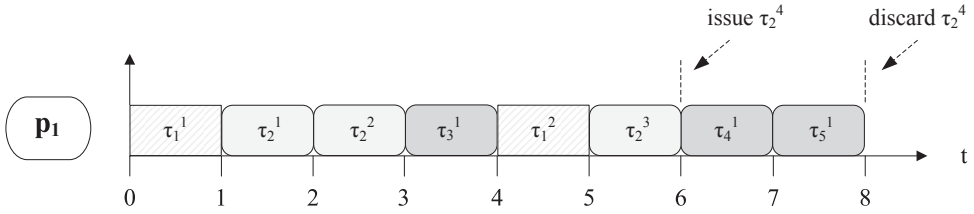

Figure 5.10: Possible scenario with missed deadline function instant

may reserve less resources with a certain degree of sacrifice of function deadline guarantee. This resource reduction should be especially wary as such a real-time system can only tolerate function missing deadline occasionally. If this situation happens too often, it means the reserved resources are not enough to support the normal operation of the system.

We may reserve resources based on observation of the actually triggered frequency of sporadic functions; or we can give different degrees of deadline guarantee to different sporadic functions based on their importance. How to properly deal with the trade-off between function deadline guarantee and resource reservation is an important future work.

5.2 Use Case Study

5.2.1 Task Dependency Relation Generation

In a real-time system, multiple tasks are cooperated together to achieve a function. Usually, in a practical application, there are dependency relations among these tasks. Such dependency relation has been widely considered when designing scheduling for real-time systems. However, works in real-time scheduling domain mainly focus on how to design scheduling to handle the task dependency relations and assume such relations are known a priori. Few works study how to generate the task dependency relation.

In order to generate the dependency relation, a feasible method is based on requirement specification of an application. However, with continuously increasing complexity in development of system, to correctly generate the dependency relation is becoming a challenge. A primary problem is that requirement specification may not be accurately and easily understood by system designers. The major reason causing such a problem is the notations and languages used in the specification lack of precise syntaxes and semantics. These notations and languages inevitably associate ambiguity and may lead to misunderstanding. To solve this problem, formal specification gives a promising solution. With precise constraints of semantics and syntaxes, formal specification can precisely define the behaviors of the system and provide a firm basis for designing the system.

Unfortunately, there exist some difficulties in using formal methods. For example, it requires significant abstraction and mathematical skills; it usually costs more in time and human effort for analysis and design [34]. These difficulties have hindered wide usage of formal methods. To address these difficulties, SOFL, a formal engineering methodology, has been proposed in [34, 35], where SOFL refers to *structured-object-oriented-formal language*. It proposes changes to notation, methodology, and support environments for constructing systems, which makes formal methods more practical and acceptable. Meanwhile, it has precise semantics and syntaxes, which provides a firm foundation for correctly generating the task dependency relation. To show how SOFL can be used to generate task dependency relation, in this subsection, a case study on cruise control system is conducted.

5.2.1.1 Cruise Control System

Cruise control system (CCS) is a servomechanism that can maintain a constant vehicle speed set by a driver. It accomplishes this function by measuring vehicle speed, comparing it to set speed, and automatically adjusting throttle according to a control algorithm. It is usually used for long drives across highways. By using the CCS, drivers do not need to control the throttle pedal to maintain the speed of vehicles, which can alleviate the fatigue of drivers. Meanwhile, it can reduce unnecessary change of speed, which usually results in better fuel efficiency. With these advantages, CCS has now been widely equipped in various brands of automobiles, such as BMW, Audi, and Volkswagen.

Figure 5.11: Function buttons on the control lever of a cruise control system (figure is from the home page of Audi)

CCS developed by different automobile manufacturers usually have different auxiliary functions. Figure 5.11 shows the control lever of a CCS equipped in an Audi automobile. A driver can activate different functions by pressing the function buttons on the control lever. Button ON and OFF are to turn on and turn off the system, respectively. After the system turns on, when the button SET is pressed, if the speed of the vehicle is within a specific speed interval which is supported by the CCS, the system will start to maintain current vehicle speed until the driver presses button OFF, or CANCEL, or steps on brake. SPEED+ and SPEED- is used to adjust the set speed when the system keeps on maintain current vehicle speed. When SPEED+ is pressed, the set speed will be increased, and the system will increase current speed to the set speed and maintain the speed of vehicle at that level. The button CANCEL can temporarily turn off the system, meanwhile the button RESUME can resume the system to the moment at which the system is temporarily turned off.

5.2.1.2 Requirements Specification

As the objective is to investigate how to generate task dependency relation from SOFL specification, rather than develop a fully functional system, for simplicity, only parts of the functions are considered in the case study. Moreover, to the consideration of safety, a new CONFIRM button is provided.

After the system turns on[2], the primary functions required by a CCS are as follows.

[2]As function button SET in Figure 5.11 is not considered, in the case study, system turns on means

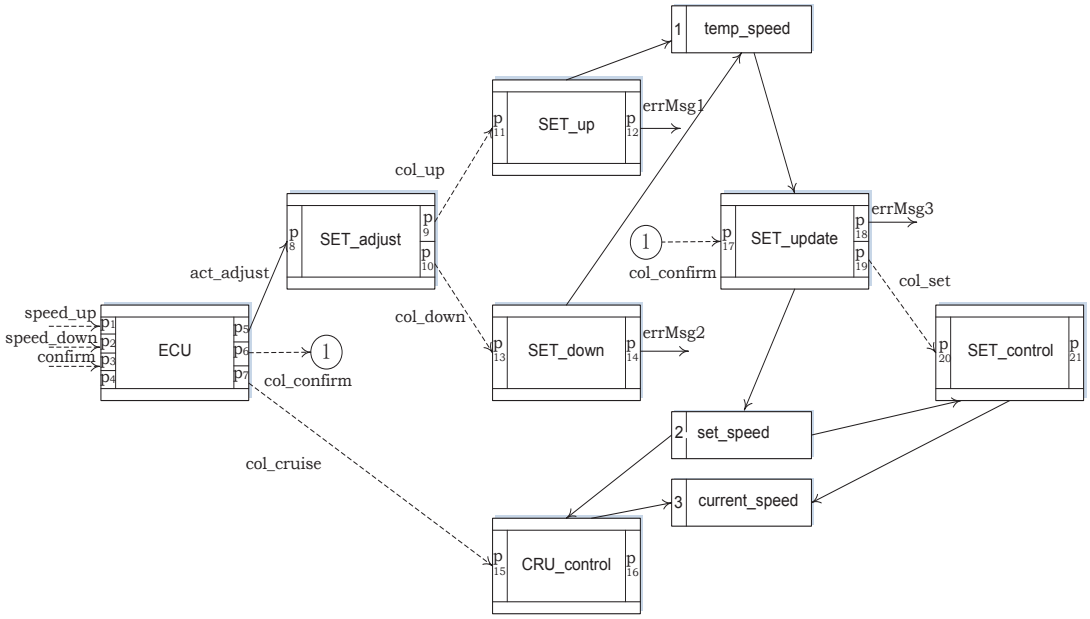

Figure 5.12: Condition data flow diagram (CDFD) for the cruise control system

1. Let the driver increase and decrease the value of the set speed. The set speed is required within a speed interval supported by the CCS. To the consideration of safety, a confirmation operation is needed to confirm the setting.

2. Keep on maintaining the vehicle speed at the set value.

Although above specification is very simple, it still may cause misunderstanding. For example, the sentence "a confirmation operation is needed to confirm the setting" does not clearly describe what will happen if the confirmation operation is not performed. A designer may think that if an operation of increasing or decreasing is not followed by a confirmation operation, the operation will be ignored. While another designer may think that the confirmation operation is needed only when the driver has finished the setting (maybe after pressing button SPEED+ and SPEED- many times). In order to avoid any potential misunderstandings, as described above, formal notation can help greatly.

5.2.1.3 SOFL Specification

A SOFL specification is a hierarchical condition data flow diagram (CDFD) that is linked with a hierarchy of specification modules (**modules**) [35]. The CDFD comprises of a set of processes and describes data flows between them, while the associated modules

the system stars to maintain current vehicle speed.

precisely define the functionality of the components (process, data flow, data store) in the CDFD. Each process in the CDFD is associated with a **process** which is defined in the modules and describes functions in terms of **pre** and **post** conditions, within the specific specification context of the module [36]. More details about SOFL specification are described in [34] [35].

CDFD

The CDFD of the CCS is shown in Figure 5.12, and the associated module is shown in Figure 5.13. In Figure 5.12, each box surrounded by narrow borders denotes a process, such as `SET_adjust()` and `CRU_control()`, which describes an operation. It consumes inputs and produces outputs. Each directed line with a labeled variable name denotes a data flow. A solid line denotes an active data flow, while a dotted line denotes a control data flow. The box with a number and an identifier (e.g., `temp_speed`) is a data store which can be accessed by processes. A directed line from a data store to a process represents the process can read the data from the store, while a directed line from a process to a data store means the process can read, write, and update the data in the store. More details about the components used in the CDFD can refer [35] [37].

The CCS comprises of two primary functions: setting desired vehicle speed (through increasing or decreasing the set speed) and maintaining the vehicle speed at the set value. These two functions are triggered by `ECU()` (electronic control unit) process. Processes with names star with `SET` are for the first functions, and processes `CRU_control()` is for the second function. When the system is running, process `ECU()` keeps on monitoring the inputs of drivers. Different inputs will trigger different processes to achieve different functions. The selection of `speed_up, speed_down`, or `confirm` denotes the corresponding function buttons on the system control lever shown in Figure 5.11 is pressed by the driver.

When button `speed_up` or `speed_down` is pressed, process `ECU()` will generate a data flow `act_adjust` to indicate which command has actually been selected, and passes this information to process `SET_adjust()`. Based on the value of `act_adjust`, process `SET_adjust()` will trigger either processes `SET_up()` (`speed_up` is selected) or `SET_down()` (`speed_down` is selected). Process `SET_up()` or `SET_down()` first reads `temp_speed` from the data store, and try to update `temp_speed` by increasing or decreasing it with a con-

stant value, respectively. As the CCS can only run within a designed speed interval, before updating the data, process `SET_up()` and `SET_down()` will first check if the updated value of `temp_speed` is within the interval. If not, an error message will be issued. Data `temp_speed` is a temporary data that can be manipulated by process `SET_up()` and `SET_down()` and only after process `SET_update()` performs, the value of `temp_speed` can be assigned to `set_speed`. After process `SET_up()` or `SET_down()` completes the updates of `temp_speed`, it will send the completion information to process `SET_update()`. Process `SET_update()` will assign data `temp_speed` to `set_speed` only after the `confirm` button is pressed. After process `SET_update()` assigns the value of `temp_speed` to `set_speed`, it will trigger process `SET_control()` to control current vehicle speed `current_speed` to the new set speed `set_speed`.

When no function button is pressed by the driver, it means the driver does not want to adjust the set speed and wants to maintain current vehicle speed, process `CRU_control()` will be triggered by process `ECU()`. Process `CRU_control()` maintains current vehicle speed `current_speed` to the value of `set_speed` based on a control algorithm.

module

Compared to the specification written in natural language given in section 5.2.1, the functional abstraction expressed by the CDFD is obviously more comprehensible, especially the dependency relations among processes can be clearly expressed. However, in order to accurate describe system behaviors, all the components (conditional process, data flows, and data stores) in the CDFD must be precisely defined. To achieve this, the CDFD is associated with a module shown as in Figure 5.13.

In the module, part **const** shows the constant variables used in the module. All the data flow variables, and data stores in the CDFD are defined in the **var** part. Each of them is defined in a specific data type. Keyword **inv** stands for invariant and indicates the properties that must be sustained throughout the entire specification. For example, `min_sp <= set_speed <= max_sp` in part **inv** means the set value of the CCS must be no larger than the maximum value that supported by the system and no less than the minimum value. Function `Controller()` achieves the function of speed control based on a control algorithm. At this level of specification, the control algorithm has not been

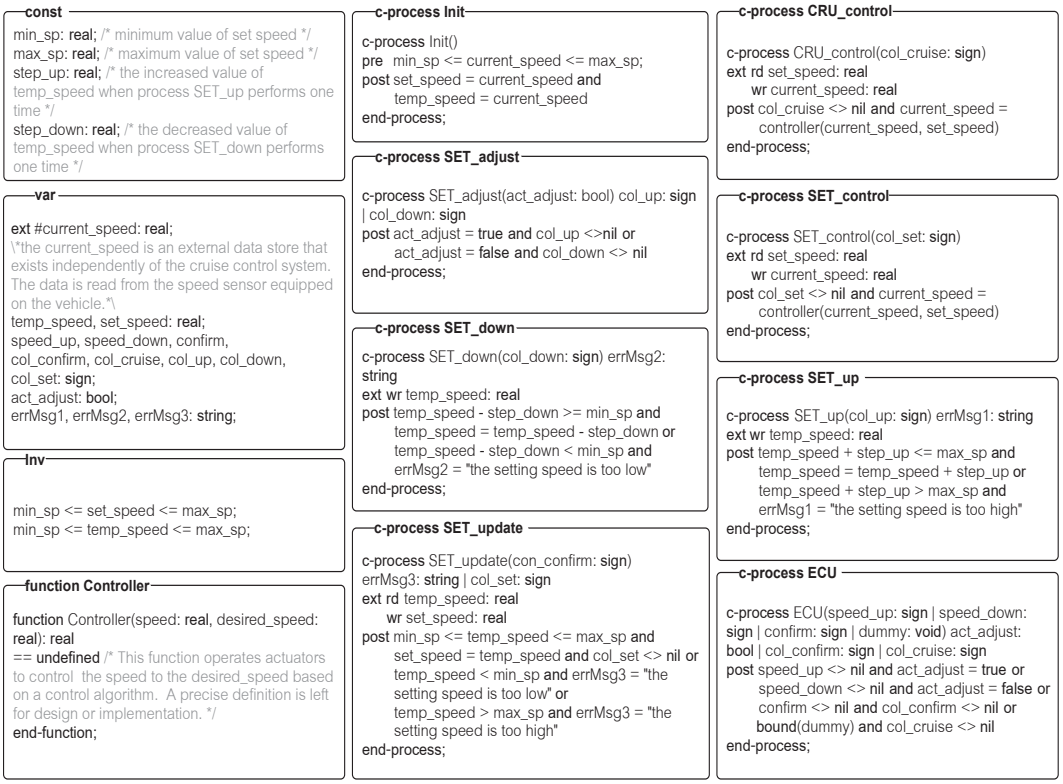

Figure 5.13: Module for the cruise control system

designed.

Process `Init()` is the initial process which performs only one time when the system stars up. We can see that, **pre** condition defined in the process `Init()` requires that `current_speed` should be less than or equal to `max_sp` and larger than or equal to `min_sp`. This ensures that the system can star up only when vehicle is running within the speed interval that supported by the CCS.

Each processes in the CDFD is associated with a **process** specification in the module. It describes functions of the processes in terms of pre and post conditions in which predicate logic is adopted. For example, the post condition in process `SET_adjust()` means: if the value of data flow variable `act_adjust` is true, process `SET_adjust()` will trigger `SET_up()` by generating control signal `col_up`, otherwise `act_adjust` with a false value will make process `SET_down()` be triggered.

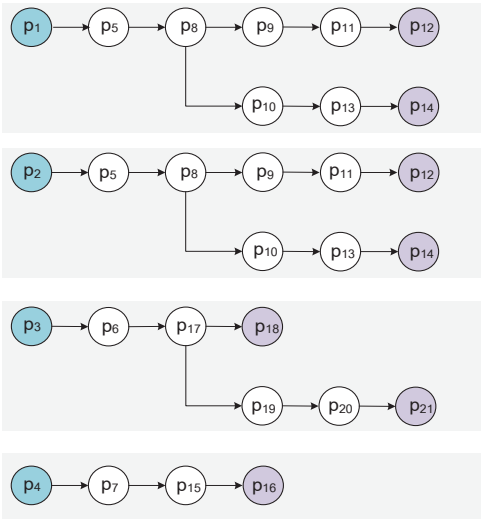

Figure 5.14: Task dependency graph for cruise control system generated from the SOFL specification

5.2.1.4 Generating Task Dependency Relation

A process described in a SOFL specification represents an operation contained in the CCS, and it can be designed as a task (thread) contained in the system. In CCS, as a task, e.g., task `ECU` (from process `ECU()` in the CDFD), can have multiple input and output ports, and signals from different input ports can activate different tasks (trigger different processes). To express dependency relation for such kind of tasks, the dependency relation should be expressed on port level. To express the task dependency relation, a directed graph, called *task dependency graph*, is constructed from the SOFL specification, and its formal definition is as follows.

Definition (task dependency graph) *A task dependency graph is a directed acyclic graph.* $G = (P, E, P_0, P_e)$*, where* P *is set of task ports,* $E \subseteq P \times P$ *is dependency relation (edge) set, with* $(p_i, p_j) \in E$*,* $p_i \neq p_j$*, where* $p_i, p_j \in P$*.* $P_0 \subset P$ *is the start port set, and* $P_e \subset P$ *is the end port set.*

An edge (p_i, p_j) in the task dependency graph means signal generated by port p_j can be issued only after port p_i generates its signal. Symbol $p_i \prec p_j$ is used to illustrate this dependency relation. The dependency relation is transitive. That is, $p_i \prec p_j, p_j \prec p_k \Rightarrow p_i \prec p_k$. A start port $p_i \in P_0$ features that $\forall p_j \in P, \nexists p_j \prec p_i$, while an end port $p_k \in P_e$ features that $\forall p_j \in P, \nexists p_k \prec p_j$.

Based on this definition, from the SOFL specification, we can easily generate the task

dependency graph. From the CDFD shown in Figure 5.12, we can see that the input ports of process `ECU()` are the start port set. By checking the corresponding process `ECU()` in the module shown in Figure 5.13, we can know that port p_5 (port indexes are denoted within each process block in Figure 5.12) can generate signal `act_adjust` only when port p_1 or p_2 accepts its signal `speed_up` or `speed_down`. This means there exist dependency relations between port (p_1, p_5) and (p_2, p_5), that is, there exist edges from p_1 to p_5 and from p_2 to p_5 in the dependency relation graph. By this method, we can get the task dependency graph for the CCS. The result is shown in Figure 5.14[3]. Note that, as either signal from port p_1 or port p_2 can activate port p_5 to generate its signal, we need two disjoint tree structures to represent the dependency relations.

In Figure 5.14, there are four disjoint tree structures in the figures (denoted with shades). Ports p_1, p_2, p_3, p_4 are the start ports. The signals accepted by these ports are generated based on operations of the driver (pressing some function buttons or pressing no function button). Ports $p_{12}, p_{14}, p_{18}, p_{21}, p_{16}$ are the end ports. A function (a response of the CCS to the corresponding operation of the driver) is denoted by a series of ports existing in the graph, which starts from a start port and ends with an end port[4]. Specific to this example, there are seven functions in the system. For example, port series $(p_4, p_7, p_{15}, p_{16})$ denotes a function which expresses the response of the CCS to the situation that no function button is pressed by the driver.

5.2.1.5 Scheduling Problem Modeling

After the task dependency relation is generated, we now can model the scheduling problem based on the models provided in RSMT. As described in section 5.2.1, seven functions are contained in the system. Since these functions are triggered by user input, sporadic function model proposed in section 5.1.1 is adopted to model these functions[5].

- $F_1 = ((T_1, \prec), rd_1, p_1)$, $T_1 = \{\tau_1, \tau_2, \tau_3\}$, where τ_1, τ_2, τ_3 represents task $\texttt{ECU}_1$, $\texttt{SET\_adjust}_1, \texttt{SET\_up}_1$, respectively.

[3]Task dependency relation is transitive. A checking algorithm to check if there exist mistakes in the SOFL specification based on the transitivity of task dependency relation is introduced in research [55].

[4]A function described at this state has been further refined compared to it described in section 5.2.1.

[5]To avoid confusion with notations used in section 5.2.2, each function is denoted by F_i rather than SF_i temporarily, where i is index of function.

- $F_2 = ((T_2, \prec), rd_2, p_2)$, $T_2 = \{\tau_4, \tau_5, \tau_6\}$, where τ_4, τ_5, τ_6 represents task $\texttt{ECU}_2$, $\texttt{SET\_adjust}_2, \texttt{SET\_down}_2$, respectively.

- $F_3 = ((T_3, \prec), rd_3, p_3)$, $T_3 = \{\tau_7, \tau_8, \tau_9\}$, where τ_7, τ_8, τ_9 represents task $\texttt{ECU}_3$, $\texttt{SET\_adjust}_3, \texttt{SET\_up}_3$, respectively.

- $F_4 = ((T_4, \prec), rd_4, p_4)$, $T_4 = \{\tau_{10}, \tau_{11}, \tau_{12}\}$, where $\tau_{10}, \tau_{11}, \tau_{12}$ represents task $\texttt{ECU}_4$, $\texttt{SET\_adjust}_4, \texttt{SET\_down}_4$, respectively.

- $F_5 = ((T_5, \prec), rd_5, p_5)$, $T_5 = \{\tau_{13}, \tau_{14}\}$, where τ_{13}, τ_{14} represents task $\texttt{ECU}_5$, $\texttt{SET\_update}_5$, respectively.

- $F_6 = ((T_6, \prec), rd_6, p_6)$, $T_6 = \{\tau_{15}, \tau_{16}, \tau_{17}\}$, where $\tau_{15}, \tau_{16}, \tau_{17}$ represents task $\texttt{ECU}_6$, $\texttt{SET\_update}_6, \texttt{SET\_control}_6$, respectively.

- $F_7 = ((T_7, \prec), rd_7, p_7)$, $T_7 = \{\tau_{18}, \tau_{19}\}$, where τ_{18}, τ_{19} represents task $\texttt{ECU}_7$, $\texttt{CRU\_control}_7$, respectively.

The task dependency relations in the task posets are expressed in the task dependency relation graph shown in Figure 5.14. In addition, the subscripts of the tasks denote the indexes of the functions in which the tasks are used. For example, $\texttt{ECU}_1$ denotes task ECU used in function F_1. Note that, a task (e.g., ECU) used in different functions has different operations.

After the functions contained in the CCS are modeled by the sporadic function model, the task, processors, and network channel model provided in RSMT can be directly applied to model other parts of the system.

Similar to CCS, for other systems e.g., radar, engine control, we can use this method to generate the task dependency relation, and then models provided in RSMT can be used to model the scheduling problem for the corresponding systems.

5.2.2 Scheduling for a Running Car

To show the usage of RSMT in practical applications, in this subsection, a case study on a running car which is equipped with multiple processors is conducted. Three systems: radar, cruise control, and engine control are considered in the case study. The application scenario is shown in Figure 5.15. When a driver drives a car on a highway, the radar

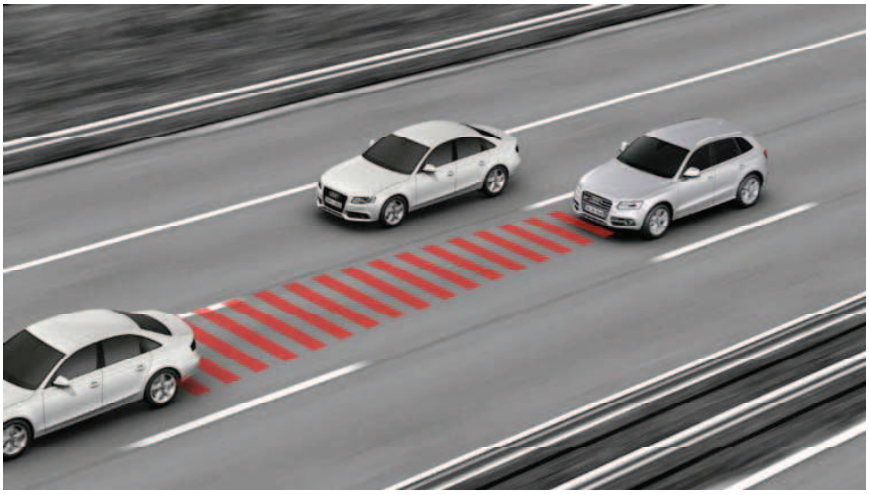

Figure 5.15: Application scenario for a running car (figure is from the Internet)

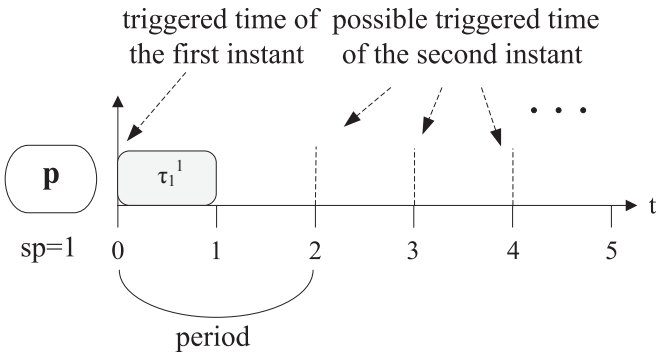

Figure 5.16: Possible execution of function SF_a

system keeps on detect the distance from the front car in order to avoid too close to that car. This detection repeats periodically, which means it can be modeled by the periodic function model. The engine control system controls the operation of the engine of the car. It is triggered by the engine events, such as pulses generated by sensors at the engine crankshaft. Therefore, we can use the sporadic function model to model the engine control system.

We now apply the combination method to design scheduling for the running car. Note that, a system may contain multiple functions. For an event-driven system, at a time, only some specific functions will be triggered to response to the initiated event. For example, there are seven functions contained in CCS. But at a time, only one function will be triggered to response to the initiated event, e.g., function F_1 will be triggered to increase the set speed (if the speed is within the supported speed interval) when drive presses button `speed_up`. However, when designing scheduling for an event-driven system,

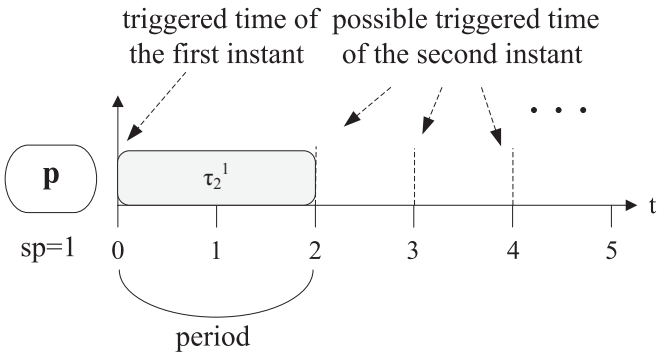

Figure 5.17: Possible execution of function SF_b

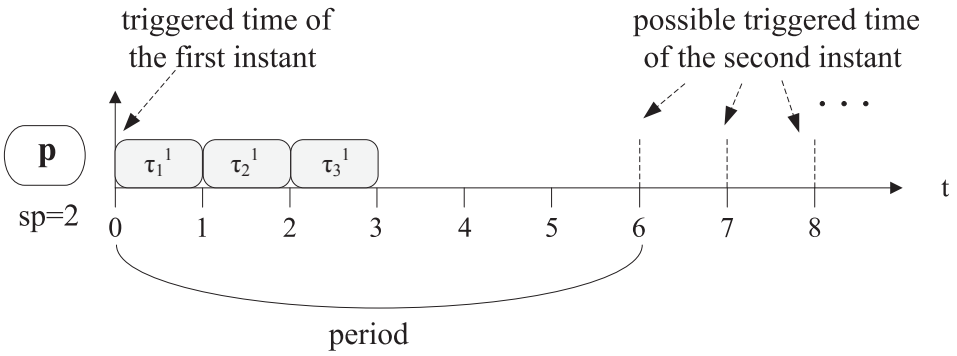

Figure 5.18: Sporadic function SF_1 representing execution of cruise control system

we should make sure that no matter which function is triggered, its deadline can always be guaranteed. Because of this, an interesting phenomenon arises. Assume an event-driven system comprising of two functions, $SF_a = ((T_a, \prec), 2, 2), T_a = \{\tau_1\}, c_1 = 1$ and $SF_b = ((T_b, \prec), 2, 2), T_b = \{\tau_2\}, c_2 = 2$. The possible execution sequences of these two functions are shown in Figure 5.16 and Figure 5.17. Obviously, only using a unit speed processor cannot guarantee the deadlines of these two functions. But if these two functions cannot be triggered at the same time, we can see that if the processor resources can make sure that function SF_b can meet its deadline, then such processor resources can also guarantee the deadline of function SF_a. Therefore, in this case, one unit speed processor is enough to guarantee the deadlines of these two functions.

We call function SF_b as *indicating function* as it can be used to indicate how much resources are needed to guarantee the deadlines of all the functions contained in an event-driven systems. Note that, indicating function may be a function set. For example, if function SF_a and SF_b can only be executed on two different processor p_1 and p_2 respectively, we need to reserve these two processor resources. Under this case, the indicating

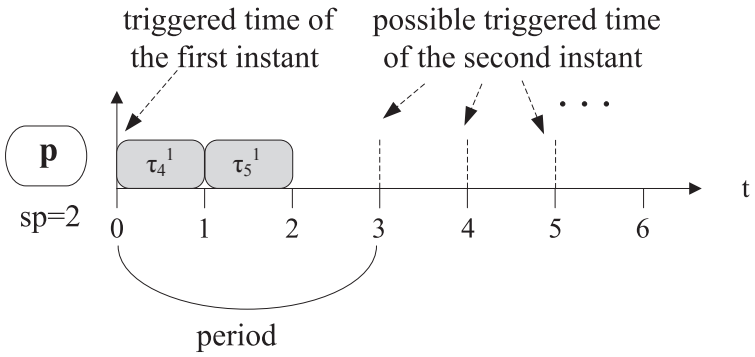

Figure 5.19: Sporadic function SF_2 representing execution of engine control system

function is the function set $\{SF_a, SF_b\}$.

As for different number of functions, the way how the combination method operates is the same, for conciseness, in the case study of the running car, it is assumed that there exist indicating function $SF_1 = ((T_1, \prec), 6, 6)$ and $SF_2 = ((T_2, \prec), 3, 3)$ in cruise control and engine control system, respectively. They are used to represent the executions of these two systems. In addition, periodic function $PF_3 = ((T_3, \prec), 4, 6)$ is used to represent execution of the radar system. The possible executions of these three functions are shown in Figure 5.18, Figure 5.19, and Figure 5.20. For the equipped processors, assume two processors with speed 1 and speed 2 are equipped in the system, and all the functions can be executed on both processors.

Through the combination method proposed in the previous section, in the offline phase, RSMT generates the scheduling table to decide function-to-processor assignment. The result is shown in Figure 5.21. It can be seen that, function SF_1 is assigned to processor p_1, while function SF_2 and PF_3 are assigned to processor p_2. After function-to-processor assignment is decided, EDF is applied to decide slot-to-task allocation online. To see the scheduling performance at system run-time, let's construct some running scenarios.

Running scenario a: *When a drive starts to drive the car, the cruise control system is turned off, while the speed of the car is increasing.*

Under this scenario, function SF_1 will not be triggered, while the triggered frequency of function SF_2 will increase. Figure 5.22 shows a possible execution sequence. As SF_1 is not triggered, there is no task executed on processor p_1. On processor p_2, after the first instant of function SF_2 is triggered at $t = 0$, the second instant is triggered after 5 time

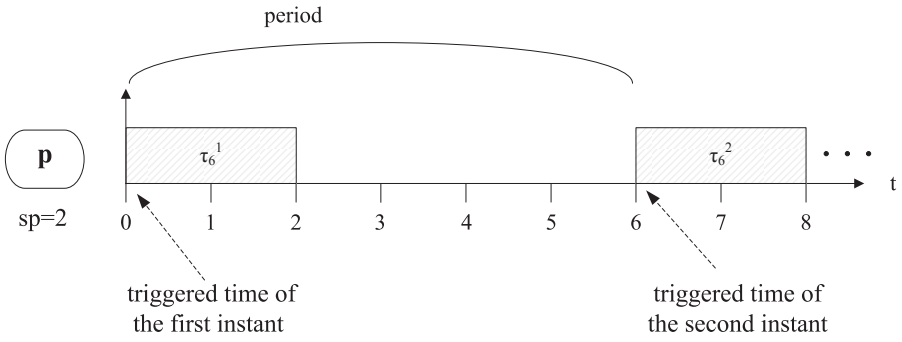

Figure 5.20: Periodic function PF_3 representing execution of radar system

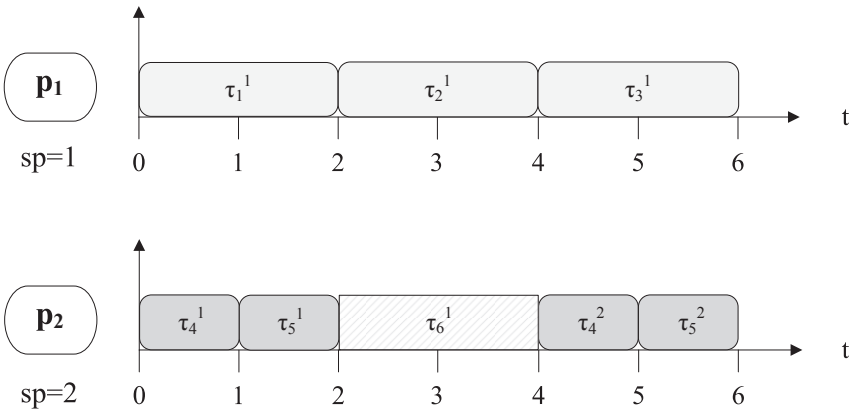

Figure 5.21: Scheduling table generated by RSMT

units. Then at $t = 9$, the third instant of function SF_2 is triggered, which has passed 4 time units after its previous triggered time.

Running scenario b: *At $t = 96$, driver turns on the cruise control system, and the car runs at a constant speed.*

Under this scenario, function SF_1 and SF_2 will be triggered at a fixed frequency. Figure 5.23 shows a possible execution sequence. It can be seen that function SF_1 is triggered every 7 time units, while function SF_2 is triggered every 5 time units.

From the scheduling results for these two running scenarios, it can be seen that the combination method can guarantee the deadlines of all the functions. This shows the effectiveness of the combination method.

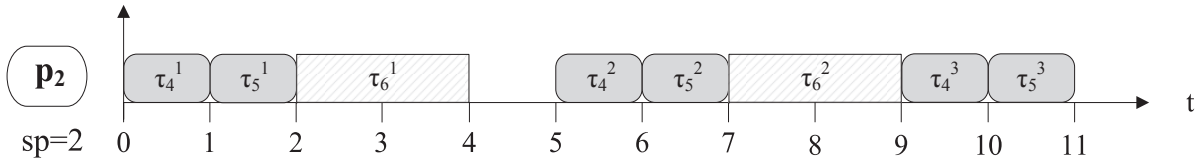

Figure 5.22: Scheduling result for running scenario `a`

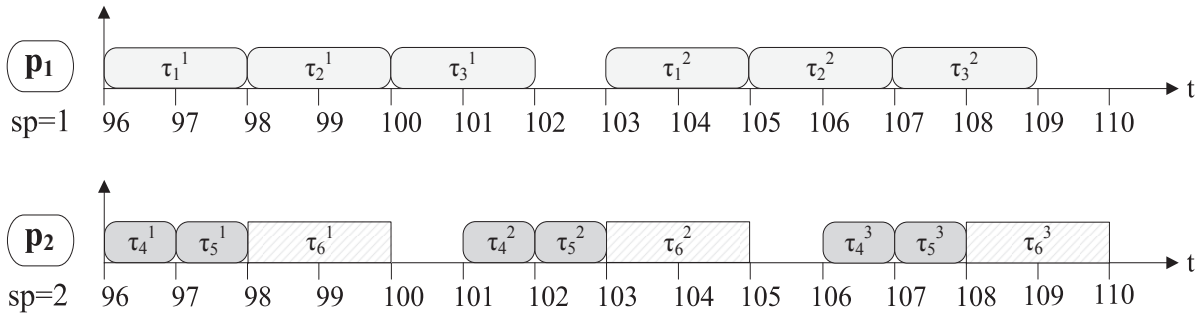

Figure 5.23: Scheduling result for running scenario `b`

5.3 Summary

Since RSMT performs in offline scheduling paradigm, it limits the capability of RSMT to handle event-triggered functions. To overcome such limitation, in this chapter, a method of combining RSMT and the online scheduling algorithm EDF is given. In the method, RSMT performs offline to assign functions to processors, while EDF performs online to allocate processor time slots to tasks. By this way, RSMT shows capability to design scheduling for systems containing event-triggered functions.

To show the usage of this combination method in practical applications, a case study on a running car is conducted. Through envisioned running scenarios, the effectiveness of the combination method is demonstrated. Moreover, from the case study, a method for generating task dependency relation based on SOFL specification is also provided. This method addresses the problem how to generate task dependency relation which is rarely studied in real-time scheduling domain.

Chapter 6

Discussion and Conclusion

6.1 Related Work

For a real-time system, sensitivity to timing is central feature of system behaviors. To guarantee the timing requirements, scientific community has made great efforts. Numerous researches have been conducted in real-time scheduling domain. The most relevant works can be divided into three categories.

6.1.1 Scheduling for Overload

Work in this dissertation

In this dissertation, RSMT is first applied to schedule uniprocessor real-time systems. The overload problem that existing works cannot handle well has been studied [30]. Moreover, when RSMT is applied to schedule multiprocessor real-time systems, the overload problem has also been considered [22].

Other works

In the literature on real-time systems, several scheduling algorithms have been proposed to deal with the overload problem. In [10], the problem of selecting tasks for rejection in an overloaded system is considered. Random criticality values are assigned to tasks. The goal is to schedule all the critical tasks and make sure that the weight of rejected non-critical tasks is minimized. As the values are randomly assigned, the performance of this method cannot be guaranteed. In [2], authors studied some special cases

of overloaded systems. They impose certain constraints on the values of task attributes. For example, under a special case, all the tasks have the same request time instant. An algorithm has been proposed to handle this special case. As the algorithms proposed in [2] can only handle some special cases, it means they are not applicable for most situations.

Other approaches focus on providing less stringent guarantees for temporal constraints. The elastic task model (ETM) proposed in [39] aims at increasing task periods to handle overload in adaptive real-time control systems. In ETM, periodic tasks are able to change their execution rate to provide different qualities of service. Authors in [6] introduced skippable tasks which are allowed to miss deadlines occasionally. Each task is assigned to a skip parameter which represents tolerance of this task to miss deadline. An algorithm was proposed to adjust system workload such that tasks adhere to their timing and skip constraints.

Comparison

All the works in [10, 2, 39, 6] are customized to their own scheduling targets, and many practical requirements (e.g., task dependency relation, tasks with different degrees of importance) cannot be handled. Moreover, none of them has dealt with multiprocessor systems. Compared to these works, because first-order logical formulas are used to formalize scheduling constraints, through various formulas, RSMT can handle many practical requirements and scheduling targets. Moreover, through formulas given in Chapter 4, RSMT can be applied to multiprocessor systems. The main disadvantage of RSMT is that RMST can only perform in offline paradigm. This limits the capability of RSMT to handle overload problem for systems containing event-triggered tasks (or functions). Further research is needed to overcome this limitation to a certain extent.

6.1.2 Scheduling for Multiprocessor Systems

Work in this dissertation

In Chapter 4 and Chapter 5, RSMT is applied to schedule multiprocessor real-time systems [22, 32]. The problem of scheduling heterogeneous multiprocessor systems is addressed, which means the problems of scheduling identical and uniform multiprocessor systems have also been addressed as such problems can be treated as special cases of

scheduling heterogeneous multiprocessor systems.

Other works

Scheduling multiprocessor real-time systems is much more complex than scheduling uniprocessor real-time systems. Most of existing works deal with the relatively simpler problem, scheduling identical multiprocessor systems, e.g., [62, 63, 64, 65, 66, 67].

For scheduling heterogeneous multiprocessor systems, some works have been conducted, e.g., [56, 57, 58, 59, 60, 61, 61, 68]. However, all these works consider the non-migrative scheduling manner. That is, tasks are not allowed to migrative among processors, which is a simpler case compared to full- and intra-migrative scheduling which allow task migrating among processors through network channels.

Few works consider full- or intra-migrative scheduling for heterogeneous multiprocessor systems. In [42], authors considered the problem of assigning implicit-deadline sporadic tasks onto a heterogeneous multiprocessor platform. The platform is limited to comprising two different types of processors (two-type platform). In [69], authors considered intra-migrative scheduling for heterogeneous multiprocessor systems. They study how to find a task-to-processor-type assignment. After the assignment is found, an optimal scheduling algorithm (without considering many practical requirements, e.g., task dependency relation) for identical multiprocessor systems, such as ERfair [25], DP-Fair [70], U-EDF [71], is used to schedule tasks. This procedure makes the algorithms proposed in [69] inevitably introduce all the restrictions (e.g., cannot handle task dependency relation) incorporated in their used algorithms (ERfair, DP-Fair, U-EDF).

Comparison

Similarly, all the works mentioned above cannot deal with many practical requirements. Compared to these works, RSMT can handle many practical requirements. In addition, RSMT can be applied to schedule heterogeneous multiprocessor systems. Moreover, as the usage of network channel model, when designing scheduling for time-driven systems, RSMT can conveniently handle all the non-, intra-, fully-migrative scheduling manners.

When design scheduling for event-driven systems, the combination method introduced

in Chapter 5 is limited to non-migrative scheduling manner. Compared to works in [56, 57, 58, 59, 60, 61, 61, 68], this combination method shows advantages: *i*) less online scheduling overhead, as performing EDF on each processor is much simpler than the other methods; *ii)* requiring weaker assumptions, e.g., no need to assume tasks are independent with each other; and *iii)* can usually achieve better scheduling result, since the underlying SMT solver searches solution space completely (when scheduling problems are within the applied scopes), while other methods mainly depend on heuristic approaches which can get good results only when proper configuration parameters for the scheduling problems are found.

The main limitation of RSMT is that it has exponential complexity. As mentioned before, how to properly deal with this complexity is an important future work.

6.1.3 Scheduling Based on SMT/SAT

Work in this dissertation

Based on SMT, RSMT is proposed. First, the focus of RSMT is on time-driven systems [22, 30, 32]. Then, through combination with EDF, RSMT shows capability to design scheduling for event-driven systems.

Other works

Some existing works have considered designing scheduling based on SMT. In [9], authors gave a simple example on using SMT for designing scheduling. They considered a scenario that scheduling three tasks (called jobs in [9]) onto two processors (called machines in [9]). Meanwhile, each task consists of two individual fragments (called tasks in [9]). In this example, they assume fragments of all the tasks are pre-assigned to specific processors before designing scheduling. Under this assumption, the scheduling problem becomes: deciding start execution time of fragments on their assigned processors.

This is much simpler than the problem of scheduling multiprocessor systems studied in Chapter 4. As the studied problem includes: (1) assigning functions to processors; (2) deciding start execution time of tasks on their assigned processors; (3) assigning tasks to network channels, if the tasks need to migrate among processors; (4) deciding start migration time of tasks on their assigned network channels. Moreover, a task may be

assigned to multiple processors and network channels, which is not considered in the example given in [9]. Thus, RSMT can also be applied to the scenario studied in [9], as the scheduling problem is one sub problem (problem (2)) considered in RSMT.

In [7, 23], authors studied the problem of scheduling time-triggered networked systems (one of multiprocessor real-time systems). Similar to the research in [9], they assume that fragments of all the tasks are pre-assigned to specific processors and network channels (task migration is considered in these works, but not considered in [9]).

In [8], authors studied the problem of scheduling distributed real-time systems (one of multiprocessor real-time systems). Their work is limited to bus network topology. Compared to this work, due to the generality of the network channel model provided in RSMT, RSMT can be applied to various kinds of network topologies (e.g., bus, ring, mesh, and tree).

Comparison

Scheduling problems studied in [9, 7, 23, 8] are simpler than the problems studied in RSMT. In addition, works in [9, 7, 23] are limited to time-driven systems, while RSMT can be applied to design scheduling for both time- and event-driven systems. Also, all the works mentioned above are customized to their own scheduling targets, while when design scheduling for time-driven systems, RSMT can be applied to various kinds of scheduling targets.

6.2 Validity of RSMT

When applying RSMT to design scheduling for real-time systems, the validity of RSMT mainly relies on the constructed SAT model. Such a SAT model comprises of a series of first-order logic formulas which define scheduling constraints that a desired optimal schedule should satisfy. This design makes RSMT valid only when all the scheduling constraints are defined in the SAT model. If a scheduling constraint which is not defined in the SAT model while required by a system, it will make RSMT invalid. For example, in Chapter 3, a constraint called *constraint on task dependency* defines the task dependency relation in the system. If this constraint is not included in the SAT model, it means the task dependency relation is not considered when applying RSMT to design scheduling.

Meanwhile, if a system requires such task dependency relation, RSMT will be invalid.

The constraints given in Chapter 3 and Chapter 4 are the most common constraints that widely exist in many real-time systems. For some systems with their own characteristics, additional constraints should be given. These constraints are mainly from the analysis of the system specification, just as described in section 5.2.1, we get the constraint on task dependency relation from the SOFL specification. To ensure the validity of RSMT, we need to make sure that all the constraints are included in the constructed SAT model. Moreover, as several logical formulas are contained in the SAT model, it is also important to make sure that no contradiction exists in the model.

To meet these requirements, some verification techniques (e.g., testing, model checking) should be applied to verify the constructed SAT model. Further research on how to apply these verification techniques is needed.

6.3 Considering Critical Resources

When applying RSMT to design scheduling for real-time systems, processor is the mainly considered computation resource. Actually, when other critical resources (e.g., printer) are considered, RSMT can be easily applied. To achieve this, we only need to add a scheduling constraint when constructing the SAT model. Based on the model defined in Chapter 4, the constraint can be expressed as:

$$\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall r_\alpha \in \mathcal{R}$$
$$(s_i^\alpha \geq s_j^\alpha + tc_j^\alpha) \vee (s_j^\alpha \geq s_i^\alpha + tc_i^\alpha)$$

Symbol s_i^α represents the time instant at which task τ_i requests resource $r_\alpha \in \mathcal{R}$, where $\mathcal{R}$ is the set of critical resources. Symbol tc_i^α means the time slots that task τ_i needs to occupy resource r_α. By adding this scheduling constraint, RSMT generated schedule can make sure that no multiple tasks access to a critical resource at the same time.

From this example, we can see that, for a newly encountered scheduling problem (e.g., considering other critical resources), through modifying or adding scheduling constraints to the constraints given in Chapter 3 and 4, developers may easily apply RSMT to solve the problem.

6.4 Limitation of RSMT

As mentioned in Chapter 4, because the computing operation of the underlying SMT solver has exponential complexity, when the scale of scheduling problem is very large, the underlying SMT solver may easily be time-out. The preliminary idea of adopting divide-and-conquer may overcome such limitation to a certain extent. But for some scheduling problems, e.g., a huge number of tasks contained in the system, as the solution space is too large, divide-and-conquer will no longer be applicable. Some other sophisticated methods should be applied to deal with such a limitation. From another point of view, for a scheduling problem which will lead to a very large solution space after it is formalized by first-order logical formulas, RSMT is not quite suitable to be applied to solve such a problem. Note that, when applying the periodic and sporadic function models, as RSMT has to consider all the function instants within lcm time units, where lcm denotes the least common multiple of function period for all the functions contained in the system, a small number of functions may also lead the underlying SMT being time-out. For example, in an audio and video system, a common audio sampling frequency is 44.1kHz, and a standard video refresh rate is 60Hz. For two functions with such frequencies, a lot of function instants need to be considered when applying RSMT to design scheduling for the system.

However we should notice that, through constantly improving the performance of SMT solvers by research communities, the capability of the solvers keeps on increasing. Benefit from this, the problem scope that RSMT can support will also keeps on enlarging.

6.5 Accomplishment

Based on RSMT, I give design guidelines for various kinds of scheduling targets and real-time systems, from uniprocessor to multiprocessor systems (including identical, uniform, and heterogeneous systems), from soft to mixed critical real-time systems. Many requirements that are considered in real-time scheduling domain have been taken into account, e.g., task dependency relation, tasks with different degrees of importance, task preemption, and task migration. These design guidelines are given gradually from simple (uniprocessor) to complex (multiprocessor) systems. Through comparisons, it can benefit

readers to solve their own scheduling problems by modifying the design guidelines that have already been given.

With these design guidelines, by applying RSMT, some problems (e.g., overload problem for time-driven systems) have been better solved compared to existing methods, and more important, solutions for some unsolved problems (e.g., scheduling heterogeneous multiprocessor systems) have been obtained.

To show the usage of RSMT in practical applications, a case study on a running car which comprises of radar, cruise control, and engine control is conducted. In the case study, a method for generating task dependency relation based on SOFL specification is given. This method addresses the problem of generating task dependency relation that is rarely studied in real-time scheduling domain.

6.6 Advantage of RSMT

Compared to the conventional design method (i.e., designing scheduling algorithms), RSMT has the following advantages.

- RSMT can be applied to a wide scope in real-time scheduling domain (various systems, various scheduling targets).
- The scheduling constraints defined for an application scenario can be easily extended to and reused for other different application scenarios (e.g., system constraints can be totally reused when adapting a system to a different scheduling target).
- RSMT can handle many complex systems (e.g., heterogeneous systems) and many practical requirements (e.g., task dependency relation).

6.7 Disadvantage and Future Work

From the simulations on applying RSMT to schedule multiprocessor systems, we can know that when a system is very complex (e.g., having many processors), RSMT may be time-out, which means the computation is too complex for the underlying SMT solver to return a solution model under the given time limitation. This situation happens is mainly because SMT solver has exponential complexity. How to properly handle this complexity

is essential for improving the applicability of RSMT. As described in section 4.5.3, a preliminary method which borrows the idea divide-and-conquer shows some potential to deal with this problem. More comprehensive study is needed.

When applying RSMT to design scheduling for event-driven systems through the combination method proposed in Chapter 5, there is a trade-off between function deadline guarantee and resource reservation. That is, reserving more resources can better guarantee function deadline but may result in more resource waste, while reserving less resources can reduce possible waste but may sacrifice a certain degree of function deadline guarantee. How to properly deal with this trade-off is an important future work.

Another future work is to study how to verify the constructed SAT model. As mentioned in section 6.2, such verification is necessary to ensure the validity of RSMT.

6.8 Summary of Scheduling Constraints

The main work for applying RSMT to design scheduling is to design scheduling constraints to construct the SAT model. In this section, all the scheduling constraints provided in Chapter 3 and Chapter 4 are summarized.

These constraints cover a wide scope of problems studied in real-time scheduling domain, but of course, cannot cover all the scheduling problems. For an uncovered problem, I hope developers can solve the problem by modifying the scheduling constraints provided here.

6.8.1 Scheduling for Uniprocessor Real-Time Systems

System Constraints

(in section 3.3)

1. Constraint on start execution time of tasks

$$\forall \tau_i \in \mathcal{T}$$

$$s_{i,1} \geq r_i$$

2. Constraint on start execution time of different fragments

$$\forall \tau_i \in \mathcal{T}, \forall f_a, f_b \in \tau_i$$

$$b > a \Rightarrow s_{i,b} \geq s_{i,a} + e_{i,a}$$

3. Constraint on processor

$$\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall f_a \in \tau_i, \forall f_b \in \tau_j$$

$$(s_{i,a} \geq s_{j,b} + e_{j,b}) \vee (s_{j,b} \geq s_{i,a} + e_{i,a})$$

4. Constraint on task dependency

$$\forall \tau_i, \tau_j \in \mathcal{T}$$

$$\tau_i \prec \tau_j \Rightarrow (s_{j,1} \geq s_{i,e} + e_{i,e}) \wedge$$

$$(s_{i,e} + e_{i,e} > d_i \Rightarrow s_{j,1} = +\infty)$$

Target Constraints

1. Maximizing number of task completion

(in section 3.3)

Let n be the number of successfully completed tasks, and its initial value is set to 0.

$$\forall \tau_i \in \mathcal{T}$$

`if` $(s_{i,e} + e_{i,e} \leq d_i)$

$$n := n + 1$$

`end`

Let symbol sn denote the maximum number of tasks in $\mathcal{T}$ that can be successfully completed. The constraints on scheduling target can be expressed as:

$$n \geq sn$$

2. Maximizing effective processor utilization

(in section 3.6)

Let symbol e be effective processor time, and its initial value is set to 0.

$$\forall \tau_i \in \mathcal{T}$$

$$\texttt{if } (s_{i,e} + e_{i,e} \leq d_i)$$

$$e := e + e_i$$

$$\texttt{end}$$

Let symbol $ss_{i,e}$ denote the maximum value of $s_{i,e}$ for tasks in $\mathcal{T}$ that have been successfully completed. The effective processor utilization epu can be calculated as:

$$epu = \frac{e}{ss_{i,e} + e_{i,e}}$$

Let symbol $sepu$ denote the maximum value of effective processor utilization. The constraints on scheduling target can be expressed as:

$$epu \geq sepu$$

3. Maximizing obtained values of completed tasks

(in section 3.6)

Let symbol v be the obtained values of the completed tasks, and its initial value is set to 0.

$$\forall \tau_i \in \mathcal{T}$$

$$\texttt{if } (s_{i,e} + e_{i,e} \leq d_i)$$

$$v := v + v_i$$

$$\texttt{end}$$

Let symbol sv denote the maximum obtained values of completed tasks. The constraints on scheduling target can be expressed as:

$$v \geq sv$$

6.8.2 Scheduling for Multiprocessor Real-Time Systems

System Constraints

(in section 4.3)

1. Constraint on start execution time of functions

$$\forall F_i \in \mathcal{F}, \forall p_a \in \mathcal{P}$$

$$s_a^{\tau s_i} \geq r f_i$$

2. Constraint on start time of task migration

$$\forall \tau_i \in \mathcal{T}, \forall n_{a\to b} \in \mathcal{N}, \exists n_{c\to a} \in \mathcal{N}$$

$$(s^i_{a\to b} \geq s^i_a + tc^i_a) \vee (s^i_{a\to b} \geq r^i_{c\to a})$$

3. Constraint on task dependency

$$\forall \tau_i, \tau_j \in \mathcal{T}, \forall p_a \in \mathcal{P}, \exists n_{b\to a} \in \mathcal{N}$$

$$\tau_i \prec \tau_j \Rightarrow (s^j_a \geq s^i_a + tc^i_a) \vee (s^j_a \geq r^i_{b\to a})$$

4. Constraint on processors

$$\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall p_a \in \mathcal{P}$$

$$(s^i_a \geq s^j_a + tc^j_a) \vee (s^j_a \geq s^i_a + tc^i_a)$$

5. Constraint on network channels

$$\forall \tau_i, \tau_j \in \mathcal{T}, i \neq j, \forall n_{a\to b} \in \mathcal{N}$$

$$(s^i_{a\to b} \geq s^j_{a\to b} + tm^j_{a\to b}) \vee (s^j_{a\to b} \geq s^i_{a\to b} + tm^i_{a\to b})$$

6. Constraint on heterogeneous processors

$$\forall p_a \in \mathcal{P}, \forall \tau_i \in \mathcal{T} - TS_a$$

$$s^i_a = +\infty$$

Target Constraints

1. Making all functions meet deadlines

(in section 4.3)

$$\forall F_i \in \mathcal{F}, \exists p_a \in \mathcal{P}$$

$$s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i$$

2. Maximizing obtained values of completed functions

(in section 4.6)

Let symbol v be the obtained values of the completed functions, and its initial value is set to 0.

$$\forall F_i \in \mathcal{FH}$$

$$\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i$$

$$v := v + v_i$$

$$\texttt{end}$$

$$\forall F_i \in \mathcal{FS}$$

$$\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} < +\infty$$

$$v := v + f_i(s_a^{\tau e_i} + tc_a^{\tau e_i})$$

$$\texttt{end}$$

Let symbol sv denote the maximum obtained values of the completed functions. The constraints on the scheduling target can be expressed as:

$$v \geq sv$$

3. Making firm deadline functions meet deadlines first

(in section 4.6)

$$\forall F_i \in \mathcal{FH}, \exists p_a \in \mathcal{P}$$

$$s_a^{\tau e_i} + tc_a^{\tau e_i} \leq df_i$$

Let symbol v be the obtained values of the completed functions, and its initial value is set to 0.

$$\forall F_i \in \mathcal{FS}$$

$$\texttt{if } \exists p_a \in \mathcal{P}, s_a^{\tau e_i} < +\infty$$

$$v := v + f_i(s_a^{\tau e_i} + tc_a^{\tau e_i})$$

$$\texttt{end}$$

Let symbol sv denote the maximum obtained values of the completed soft deadline functions. The constraints on scheduling target can be expressed as:

$$v \geq sv$$

Bibliography

[1] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250–1258, 2009.

[2] S.K. Baruah, J. Haritsa, and N. Sharma, "On-line scheduling to maximize task completions," *IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, pp. 228–236, Dec. 1994.

[3] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116–128, 1991.

[4] S.K. Baruah and J.R. Haritsa, "Scheduling for overload in real-time systems," *IEEE Transactions on Computers*, vol. 46, no. 9, pp. 1034–1039, 1997.

[5] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[6] A. Marchand, M. Chetto, "Dynamic scheduling of periodic skippable tasks in an overloaded real-time system," *IEEE/ACS International Conference on Computer Systems and Applications*, Doha, Qatar, pp. 456–464, Apr. 2008.

[7] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," *IEEE Real-Time Systems Symposium*, San Diego, USA, pp. 375–384, Nov. 30 – Dec. 3, 2010.

[8] A. Metzner, et al., "Scheduling distributed real-time systems by satisfiability checking," *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 409-415, Aug. 2005.

[9] L.D. Moura, and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[10] S. Hwang, C.M. Chen, and A.K. Agrawala, "Scheduling an overloaded real-time system," *IEEE Annual International Phoenix Conference on Computers and Communications*, Arizona, USA, pp. 22–28, Mar. 1996.

[11] K. Ramamritham and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[12] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," *IEEE Real-Time Systems Symposium*, Lake Buena Vista, USA, pp. 182–190, Dec., 1990.

[13] R.I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 5, pp. 35:1–35:44, 2011.

[14] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 40–61, 1973.

[15] N. C. Audsley, and et al., "Fixed priority scheduling: an historical perspective," *Real-Time Systems*, vol. 8, no. 3, pp. 173–198, 1996.

[16] L. Sha, and et al., "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 8, no. 2, pp. 101–155, 2004.

[17] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[18] C. Barrett, R. Sebastiani, R. Seshia, and C. Tinelli, "Satisfiability modulo theories," *Handbook of Satisfiability*, vol. 185, 2009.

[19] L.d. Moura N. Bjorner, "Satisfiability modulo theories: an appetizer," *Formal Methods: Foundations and Applications*, LNCS, vol. 5902, pp. 23–36, 2009.

[20] L. Moura and N. Bjorner, "Z3: an efficient SMT solver," *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, pp. 337–340, 2008.

[21] B. Dutertre, "Yices 2.2," *International Conference on Computer-Aided Verification*, Vienna, Austria, LNCS, vol. 8559, pp. 737–744, 2014.

[22] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "A Framework for Scheduling Real-Time Systems," *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, pp. 182–187, Jul., 2016.

[23] S.S. Craciunas and R.S. Oliver, "SMT-based task- and network-level static schedule generation for time-triggered networked systems," *International Conference on Real-Time Networks and Systems*, NY, USA, pp. 45–54, Oct., 2014.

[24] S.K. Baruah, N. Cohen, G. Plaxton, and D. Varvel, "A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[25] J. Anderson and A. Srinivasan, "Early-release fair scheduling," *Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, pp. 35–43, Jun., 2000.

[26] L. Zhang, D. Goswami, R. Schneider, and S. Chakraborty, "Task- and network-level schedule co-synthesis of ethernet-based time-triggered systems," *Asia and South Pacific Design Automation Conference*, Singapore, pp. 119–124, Jan., 2014.

[27] T. Pop, P. Eles, and Z. Peng, "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems," *International Symposium on Hardware/Software Codesign*, Estes Park, USA, pp. 187-192, May, 2002.

[28] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, 2nd edition, Springer, 2004.

[29] J. Liu, Real-Time Systems, 1st edition, Prentice Hall, 2000.

[30] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "Scheduling overload for real-time systems using SMT Solver," *IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Shanghai, China, pp. 189–194, May, 2016.

[31] E.A. Lee, "The past, present and future of cyber-physical systems: a focus on models," *Sensors*, vol. 15, no. 3, pp. 4837–4869, 2015.

[32] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SMT-based scheduling for multiprocessor real-time systems," *IEEE/ACIS International Conference on Computer and Information Science*, Okayama, Japan, pp. 1–7, Jun., 2016.

[33] M.L. Dertouzos, "Control robotics: the procedural control of physical processes," *International Federation for Information Processing Congress*, pp. 807–813, 1974.

[34] S. Liu, Formal Engineering for Industrial Software Development Using the SOFL Method, Springer, 2004.

[35] S. Liu, A.J. Offutt, C. H.-Stuart, Y. Sun, and M. Ohba, "SOFL: a formal engineering methodology for industrial applications," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 24–45, 1998.

[36] X. Wang and S. Liu, " An approach to declaring data types for formal specifications," *International Workshop on SOFL+MSVL*, LNCS, vol. 8332, Springer, 2014,

[37] C. H.-Stuart and S. Liu, "An operational semantics for SOFL," *Asia-Pacific Software Engineering Conference*, pp. 52–61, Dec. 1997.

[38] P. Derler, E.A. Lee, and A. S.-Vincentelli., "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13–28, 2012.

[39] G. Buttazzo, G. Lipari, and L. Abeni,"Elastic task model for adaptive rate control," *IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 286–295, Dec., 1998.

[40] A. Burns and A. Wellings, Real-Time Systems and Programming Languages, 4th edition, Addison Wesley, 2009.

[41] G. Buttazzo, Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications, 2nd edition, Springer, 2005.

[42] G. Raravi, et al., "Task assignment algorithms for two-type heterogeneous multiprocessors," *Euromicro Conference on Real-Time Systems*, Pisa, Italy, pp. 34–43, Jul. 2012.

[43] AMD Accelerated Processing Units, AMD Inc, 2013, http://fusion.amd.com

[44] Apple A5X: Dual-core CPU and Quad-core GPU, Apple Inc, 2013, http://www.apple.com/ipad/specs/

[45] Intel Atom Processor, Intel Corporation, 2013, http://www.intel.com/atom

[46] Intel Core Processor Family, Intel Corporation, 2013, http://www.intel.com/core

[47] Tegra 4: Mobility at the speed of life, Nvidia Inc, 2013, http://www.nvidia.com/object/tegra.html

[48] Snapdragon Processors: All-in-one Mobile Processor, Qualcomm Inc, 2013, http://www.qualcomm.com/

[49] Exynos 5 OCTA Processor, Samsung Inc, 2013, http://www.samsung.com/exynos/

[50] NOVATHOR–Smartphone and Tablet Platforms, ST Ericsson, 2013, http://www.stericsson.com/

[51] OMAP Technologies-OMAP Applications Processors, Texas Instruments, 2013, http://www.ti.com/

[52] Z. Cheng, H. Zhang, Y. Tan, and A.O. Lim, "Greedy scheduling with feedback control for overloaded real-time systems," *IFIP/IEEE International Symposium on Integrated Network Management*, Ottawa, Canada, pp. 934–937, May, 2015.

[53] D. Johnson, Near-Optimal Bin Packing Algorithm, PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[54] Z. Cheng, H. Zhang, Y. Tan, and A.O. Lim, "DPSC: a novel scheduling strategy for overloaded real-time systems," *IEEE International Conference on Computational Science and Engineering*, Chengdu, China, pp. 1017–1023, Dec., 2014.

[55] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SOFL-based dependency graph generation for scheduling," *International Conference on System of Systems Engineering*, Kongsberg, Norway, pp. 1–6, Jun, 2016.

[56] G. Horowitz and S. Sahni, "Exact and approximate algorithms for scheduling non-identical processors," *Journal of the ACM*, vol. 23, no. 2, pp. 317–327, 1976.

[57] S. Baruah, "Partitioning real-time tasks among heterogeneous multiprocessors," *International Conference on Parallel Processing*, pp. 467–474, Aug., 2004.

[58] S. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," *Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, pp. 536-543, Jan., 2004.

[59] J. Correa, M. Skutella, and J. Verschae, "The power of preemption on unrelated machines and applications to scheduling orders," *Mathematics of Operations Research*, vol. 37, no. 2, pp. 379–398, 2012.

[60] J. Lenstra, D. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, pp. 259–271, 1990.

[61] A. Wiese, V. Bonifaci, and S. Baruah, "Partitioned EDF scheduling on a few types of unrelated multiprocessors," *Real-Time Systems*, vol. 46, no. 2, pp. 219–238, 2013.

bibitemJansen1999 K. Jansen and L. Porkolab, "Improved approximation schemes for scheduling unrelated parallel machines," *31st Annual ACM Symposium on Theory of Computing*, New York, USA, pp. 408–417, 1999.

[62] A. Srinivasan and S.K. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," *Information Processing Letters*, vol. 84, no. 2, pp. 93–98, Oct., 2002.

[63] B. Veeravalli, D. Ghose, and T.G. Robertazzi, "Divisible load theory: a new paradigm for load scheduling in distributed systems," *Cluster Computing*, vol. 6, no. 1, pp. 7–17, 2003.

[64] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," *Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, pp. 181–190, 2008.

[65] X. lin, et al., "Real-time divisible load scheduling for cluster computing," *Real-Time and Embedded Technology and Applications Symposium*, Bellevue, WA, pp. 303–314, 2007.

[66] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," *Euromicro Conference on Real-Time Systems*, Dublin, Ireland, pp. 239–248, 2009.

[67] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," *Euromicro Conference on Real-Time Systems*, Dublin, Ireland, pp. 249–258, 2009.

[68] B. Andersson, G. Raravi, and K. Bletsas,"Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," *IEEE Real-Time Systems Symposium*, San Diego, USA, pp. 239–248, 2010.

[69] G. Raravi, Real-Time Scheduling on Heterogeneous Multiprocessors, PhD thesis, Doctoral Programme in Electrical and Computer Engineering, University of Porto, 2014.

[70] G. Levin, and et al.,"DP-FAIR: a simple model for understanding optimal multiprocessor scheduling," *Euromicro Conference on Real-Time Systems*, Brussels, Belgium, pp. 3–13, 2010.

[71] G. Nelissen, and et al., "U-EDF: an unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," *Euromicro Conference on Real-Time Systems*, Brussels, Belgium, pp. 13–23, 2012.

[72] G.E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, 1965.

[73] R.W. Keyes, "The impact of moore's law," *Solid State Circuits Newsletter*, 2006.

[74] D. Liu, X.S. Hu, and et al.,"Firm real-time system scheduling based on a novel QoS constraint," *IEEE Real-Time Systems Symposium*, Cancun, Mexico, pp. 386–395, 2003.

[75] D.D. Niz, K. Lakshmanan and R. Rajkumar,"On the scheduling of mixed-criticality real-time task sets," *IEEE Real-Time Systems Symposium*, Washington, DC, USA, pp. 291–300, 2009.

[76] C.G. Cassandras and S. Lafortune,"Introduction to discrete event systems," *Kluwer Academic Publishers*, 1999.

[77] H. Kopetz,"Event-triggered versus time-triggered real-time systems," *Operating Systems of the 90s and Beyond*, Springer, pp. 86–101, 1991.

[78] A. Albert, R.B GmbH,"Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *Embedded World 2004*, pp. 235–252, 2004.

[79] A. Albert, R.B GmbH,"Evaluation and comparison of the real-time performance of CAN and TTCAN," *International CAN Conference*, pp. 1–8, 2003.

[80] G. Brassard and P. Bratley, Fundamental of Algorithmics, Prentice-Hall, 1996.

[81] D. Isovic, and G. Fohler,"Handling mixed sets of tasks in combined offline and online scheduled real-time systems," *Real-Time Systems*, vol. 43, no. 3, pp. 296–325, 2009.

[82] N.C. Audsley, et al.,"Hard real-time scheduling: the deadline monotonic approach," *Real-Time Programming*, pp. 127–132, 1992.

Publications

Journal

[1] Z. Cheng, Y. Tan, and Y. Lim, "Design and Evaluation of Hybrid Temperature Control for Cyber-Physical Home Systems," *Internation Journal of Modelling, Identification and Control (IJMIC)*, Inderscience Publishers, vol. 26, no. 3, pp. 196–206, 2016.

[2] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SMT-based Scheduling for Overloaded Real-Time Systems," *IEICE Transactions on Information and Systems* (accepted).

[3] Z. Cheng, Y. Tan, and Y. Lim, "SMT-based Scheduling for Multiprocessor Real-Time Systems," *Real-Time Systems*, Springer (submitted).

International Conference

[4] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "A Framework for Scheduling Real-Time Systems," *22th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, USA, pp. 182–187, Jul., 2016.

[5] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SMT-based Scheduling for Multiprocessor Real-Time Systems," *15th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, Okayama, Japan, pp. 1–7, Jun., 2016.

[6] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SOFL-based Dependency Graph Generation for Scheduling," *11th International Conference on System of Systems Engi-*

neering (SoSE), Kongsberg, Norway, pp. 1–6, Jun., 2016.

[7] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "Scheduling Overload for Real-Time Systems using SMT Solver," *17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Shanghai, China, pp. 189–194, May, 2016.

[8] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "A Case Study: SOFL + Model Checking for OSEK/VDX Application," *5th International Workshop on SOFL + MSVL in ICFEM*, Paris, France, pp. 132–146, Nov., 2015.

[9] Z. Cheng, H. Zhang, Y. Tan, and A.O. Lim, "Greedy Scheduling with Feedback Control for Overloaded Real-Time Systems," *14th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, Canada, pp. 934–937, May, 2015.

[10] Z. Cheng, H. Zhang, Y. Tan, and A.O. Lim, "DPSC: A Novel Scheduling Strategy for Overloaded Real-Time Systems," *17th IEEE International Conference on Computational Science and Engineering (CSE)*, Chengdu, China, pp. 1017–1023, Dec., 2014.

[11] Z. Cheng, Y. Tan, and A.O. Lim, "Fitting Method for Hybrid Temperature Control in Smart Home Environment," *6th International Conference on Modelling, Identification and Control (ICMIC)*, Melbourne, Australia, pp. 300–305, Dec., 2014.

[12] Z. Cheng, W.W. Shein, Y. Tan, and A.O. Lim, "Energy Efficient Thermal Comfort Control for Cyber-Physical Home System," *4th IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Vancouver, Canada, pp. 797–802, Oct., 2013.

[13] H. Zhang, Z. Cheng, C. Tian, Y. Lu, and G. Li, "Verifying OSEK/VDX Applications: An Optimized SMT-based Bounded Model Checking Approach," *15th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, Okayama, Japan, pp. 1–6, Jun., 2016.

[14] W.W. Shein, Z. Cheng, Y. Tan, and A.O. Lim, "Study of Temperature Control using Cyber-Physical System Approach in Home Environment," *1st IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, Taipei, Taiwan, pp. 78–83, Aug., 2013.

Domestic Conference

[15] Z. Cheng, Y. Tan, and Y. Lim, "rERA: An Optimization Algorithm of Task Dependency Graph for Scheduling," *IEICE General Conference*, Fukuoka, Japan, Mar., 2016.

[16] Z. Cheng, W.W. Shein, Y. Tan, and A.O. Lim, "Fitting Method for Hybrid Temperature Control in Cyber-Physical Smart Home Environment," *IEICE General Conference*, Niigata, Japan, Mar., 2014.

[17] Z. Cheng, W.W. Shein, Y. Tan, and A.O. Lim, "High Efficient Control Algorithm for Thermal Comfort in Smart Homes Environment using Cyber-Physical Systems," *IEICE Technical Report on Ambient Intelligence and Sensor Networks (ASN)*, Tokyo, Japan, Nov., 2013.

[18] Z. Cheng, Y. Tan, and A.O. Lim, "Study of Temperature Control in Cyber-Physical Home System Environment," *IEICE Technical Report on Information Networks (IN)*, Kyoto, Japan, Apr., 2013.