UNIVERSITÉ PARIS-EST

Thèse

en vue de l'obtention du titre de

Docteur
de l'Université Paris-Est

Spécialité : Informatique
École doctorale : MSTIC

Scheduling of Parallel Real-time DAG Tasks on Multiprocessor Systems

---

Manar Qamhieh

soutenue le 26 Janvier 2015

Jury:

Sanjoy K. Baruah	UNC, USA	Rapporteur
Maryline Chetto	IRCCyN, France	Examinateur et Président du jury
Liliana Cucu-Grosjean	INRIA, France	Examinateur
Laurent George	ESIEE, France	Directeur
Serge Midonnet	UPEM, France	Examinateur
Pascal Richard	ENSMA, France	Rapporteur

UNIVERSITÉ
PARIS-EST
MARNE-LA-VALLÉE

LABORATOIRE D'INFORMATIQUE
GASPARD-MONGE

Sous la co-tutelle de :
CNRS
ÉCOLE DES PONTS PARISTECH
ESIEE PARIS
UPEM • UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

سَلامٌ لِأَرضٍ خُلقتْ للسَّلام وَ ما رأَتْ يَوماً سَلاماً*

محمود درويش (شاعر فلسطيني)

* Peace to a land that was created for peace, and never saw a peaceful day.

Mahmoud Darwish (Palestinian poet)

Acknowledgements

During my PhD studies, I have met many people who affected me personally and scientifically. Their presence was essential for my achievements and success, therefore, I dedicate the following humble words to express my gratitude and to acknowledge their help each in their own way.

I would like to start by thanking my supervisors Laurent George and Serge Midonnet for their expert advice and constant encouragements during the entire period of research work. When our work together began 4 years ago as an internship in the LIGM laboratory, scientific research was a new domain for me. They taught me the proper research methodology and how to write scientific papers. Many thanks to Serge for his logical evaluation of research by questioning every choice and solution. Similarly, many thanks to Laurent for his precious guidance and advice regarding my research, and for being available whenever I needed a meeting with him despite his busy schedule.

I would like to thank the members of the AlgoTR and LRT teams at LIGM for the interesting scientific discussions about various real-time and network research problems. Among these members, I would especially like to thank Frédéric Fauberteau with whom I have worked since my first day in the lab. He was always available to answer my questions, to discuss possible solutions and approaches in my research and to proof-read my thesis manuscript.

Many thanks to Prof. Sanjoy Baruah for welcoming me into his lab at the University of North-Carolina at Chapel-Hill for two months during my thesis. It was a real honor to work under his supervision and to absorb some of his research methodology.

Over the years of my thesis, I shared an office with good colleagues, with whom I became close friends, Younès Chandarli, Fadhela Kerdjoudj and Paul Morel. The office was a comfort zone for me and our various discussions and coffee breaks were a stress reliever from work. Thanks to Fadhela for encouraging my culinary skills, to Younès for being a good travel friend when we attended the same conferences and seminars and to both of them for teaching me the Algerian accent and for being my private French tutors. On the other hand, Paul was one of the most gentle people I ever met, he was always smiling, considerate and pleasant to talk with.

I met many wonderful friends from the lab and the University Paris-Est with whom I interacted and I had various scientific, personal and cultural discussions and activities. Among them are my lunch buddies Ali Marandi, Karel Břinda, Safa Hamdoun, Younes Maaouni and Zakaria Chemli. Many thanks to my friend Rémi Maurice for helping me solve some of my LaTeX problems. Finally, I would like to thank the administration staff in the lab and the university for their constant help.

I am fortunate to have a loving family who supports me all the time and has faith in me. I dedicate this work to my mother who is by far the strongest woman I have ever met and who

taught me how to face challenges with patience and persistence. I am thankful for my father, my brothers and my sister (who is also my best friend) for being always there for me despite the long graphical distance that separated us over the last five years. I learned a lot from this experience and I believe that I could not have succeeded in my work without their support and encouragement.

I am very grateful to my husband Saleh, who I met at the beginning of my thesis, got married in the following year and became inseparable since then. The life of a couple of PhD students is complicated, but he made it possible with his kindness, gentleness and patience. I will never forget his supportive and understanding attitude when I am nervous or overloaded with work. His presence in my life was a great bless.

Finally, I am thankful for Allah the Almighty, my success can only come from Him.

To my family & my husband

for their
love, endless support and encouragement ...

Abstract

Scheduling of Parallel Real-time DAG Tasks on Multiprocessor Systems

The use of multiprocessor systems has been increased recently in industrial applications, and parallel architectures have been introduced for the software to become compatible with the hardware. Respectively, their use has been extended to real-time systems, whose execution is performed based on certain temporal constraints. Thus, the real-time scheduling problem has become more complex and challenging. In multiprocessor systems, a hard real-time scheduler is responsible of allocating active jobs to the available processors of the systems while respecting their timing parameters.

In this thesis, we are interested in studying the hard real-time scheduling problem of parallel Directed Acyclic Graph (DAG) tasks on multiprocessor systems. In this model, a task is defined as a set of dependent subtasks that execute under precedence constraints. The execution order of these subtasks is dynamic, i.e., a subtask can execute either sequentially or in parallel with its siblings based on the decisions of the real-time scheduler. To this end, we analyze two DAG scheduling approaches to determine the execution order of subtasks: the Model Transformation and the Direct Scheduling approaches. We consider global preemptive multiprocessor scheduling algorithms to be used with the scheduling approaches, such as Earliest Deadline First (EDF) and Deadline Monotonic (DM).

The Model Transformation approach aims at converting each DAG task into a collection of independent sequential threads with intermediate timing parameters. The objective of this approach is to facilitate the scheduling process of parallel tasks by avoiding the internal dependencies between subtasks through transformation. We provide a DAG-Str (DAG Stretching) Algorithm to schedule periodic implicit-deadline DAG tasks. The concept of the DAG-Str algorithm is to execute DAGs as sequentially as possible and avoid their parallel execution. After applying the DAG-Str algorithm, at most one thread from each segment in the DAG is forced to execute on two processors. In order to reduce the number of thread migrations, we present the Seg-Str (Segment Stretching) algorithm which is a modified version of the DAG-Str algorithm. Briefly, the Seg-Str algorithm forces a single thread from each stretched job to migrate between processors. We evaluate the schedulability performance of both stretching algorithms by providing

a resource augmentation bound when global EDF scheduling algorithm is used, which computes the least processor speed that guarantees the schedulability of stretched DAG sets.

The second approach is the Direct Scheduling, which aims at scheduling DAG tasks using any real-time scheduling algorithm directly on DAGs without modifying their model or their timing characteristics. The real-time scheduler is aware of the intra-task parallelism of DAG tasks and of the precedence constraints that determine the execution order of their subtasks. We are interested in analyzing the importance of the internal structure on the schedulability analysis of common multiprocessor scheduling algorithms. We study the Direct Scheduling at DAG-Level, in which real-time algorithms take scheduling decisions based on the global timing parameters of DAGs. Then, we analyze the GEDF schedulability condition for DAG tasks while taking into consideration the internal structure of DAGs. As a result, we provide an adapted schedulability condition for DAG scheduling for any work conserving algorithm and GEDF. Then, we propose a Subtask-Level scheduling of DAGs in which real-time algorithms take scheduling decisions based on local timing parameters of subtasks. We also provide interference and workload analyses for this scheduling, and we provide schedulability conditions for any work conserving algorithm and GEDF. Due to the incomparability of DAG scheduling approaches, we use extensive simulations to compare their schedulability performance when global EDF and DM are used.

Résumé

Ordonnancement Temps Réels des Tâches Parallèles sur des Systèmes Multiprocesseurs

Les applications temps réel durs sont celles qui doivent s'exécuter en respectant des contraintes temporelles. L'ordonnancement temps réel a bien été étudié sur mono-processeurs depuis plusieurs années. Récemment, l'utilisation d'architectures multiprocesseurs a augmenté dans les applications industrielles et des architectures parallèles sont proposées pour que le logiciel devienne compatible avec ces plateformes. L'ordonnancement multiprocesseurs de tâches parallèles dépendantes n'est pas une simple généralisation du cas mono-processeur et la problématique d'ordonnancement devient plus complexe et difficile. La responsabilité de l'ordonnanceur multiprocesseur est de trouver une façon de choisir, à chaque instant, quelles taches doivent s'exécuter sur les processeurs en respectant leurs contraintes temporels.

Dans cette thèse, nous étudions le problème d'ordonnancement temps réel de graphes de tâches orientés acycliques parallèles sur des plateformes multiprocesseurs. Dans ce modèle, un graphe est composé d'un ensemble de sous-tâches dépendantes avec des contraintes de précédence. L'ordre d'exécution des sous-tâches est dynamique, c'est-à-dire que les sous-tâches peuvent s'exécuter en parallèle ou en séquence par rapport aux décisions de l'ordonnanceur temps réel. Pour traiter les contraintes de précédence, nous proposons deux méthodes pour ordonnancer les graphes, par transformation du modèle de graphe et par ordonnancement direct des graphes.

Concernant la méthode de transformation du modèle de graphe, nous nous intéressons à l'ordonnancement des graphes périodiques à échéance sur requête. Cette méthode simplifie l'ordonnancement des graphes en transformant le modèle parallèle en un modèle séquentiel de sous tâches indépendantes afin d'éviter les dépendances internes. Nous proposons deux algorithmes de transformations (DAG-Str et Seg-Str) pour forcer l'exécution séquentielle des sous-tâches. Nous proposons une analyse d'ordonnancabilité par calculer le facteur d'expansion (speedup) pour l'ordonnancement EDF.

Concernant la méthode d'ordonnancement direct, nous nous intéressons à l'ordonnancement des graphes sporadiques à échéance contrainte. Cette méthode garde les caractéristiques générales des graphes et elle considère que l'ordonnanceur prend en compte ces contraintes de précédence. Nous nous montrons l'importance de la structure interne des graphes sur l'ordonnancement et l'analyse.

Nous proposons l'ordonnancement direct au niveau des graphes, qui considère les paramètres temporels des graphes comme leurs échéances et leurs périodes. Puis, nous proposons un ordonnancement direct au niveau des sous-tâches, qui considère les paramètres temporels des sous-tâches. Le modèle de graphe ne caractérise pas les sous-tâches par des échéances ou des périodes locales. Nous proposons donc des algorithmes simples pour définir les paramètres temporels locaux des sous-tâches, comme des échéances, des périodes et des gigues d'activation.

Enfin, nous prouvons que les deux méthodes d'ordonnancement de graphes ne sont pas comparables. Nous fournissons alors des résultats de simulation pour comparer ces méthodes en utilisant les algorithmes d'ordonnancement globaux EDF et DM. Nous avons développé un logiciel nommé YARTISS pour générer des graphes aléatoires et réaliser les simulations.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

DAG	**D**irected **A**cyclic **G**raph
DAG-Str	**DAG Str**etching Transformation
DM	**D**eadline **M**onotonic
DP	**D**ynamic **P**riority
EDF	**E**arliest **D**eadline **F**irst
FJP	**F**ixed **J**ob **P**riority
FTP	**F**ixed **T**ask **P**riority
GEDF	**G**lobal **EDF**
LLF	**L**east **L**axity **F**irst
MTS	**M**ulti-**T**hreaded **S**egment task
OPA	**O**ptimal **P**riority **A**ssignment
RM	**R**ate **M**onotonic
Seg-Str	**Seg**ment **Str**etching Transformation
Task-Str	**Task Str**etching Transformation
WCET	**W**orst-**C**ase **E**xecution **T**ime

Symbols

τ	a real-time task set
m	the number of processors in the system
ν	processor's speed
τ_i	the i^{th} task τ
$\overline{\tau}_i$	the i^{th} task from the Multi-Threaded Segment (MTS) model
$\tau_{i,j}$	the j^{th} subtask in DAG task τ_i
$S_{i,j}$	the j^{th} segment in the MTS task τ_i
$\tau_{i,j}^k$	the k^{th} thread in segment $S_{i,j}$ of τ_i
J_i^k	the k^{th} job of task τ_i
$J_{i,j}^k$	the k^{th} job of subtask $\tau_{i,j}$
J_i	any job of task τ_i
C_i	the worst-case execution time of task τ_i
$C_{i,j}$	the worst-case execution time of subtask $\tau_{i,j}$
$c_{i,j}$	the worst-case execution time of any thread in segment $S_{i,j}$
T_i	minimum inter-arrival time / period between successive jobs of task τ_i
D_i	relative deadline of task τ_i
$D_{i,j}$	local relative deadline of subtask $\tau_{i,j}$
O_i	local/intermediate offset of task τ_i
$O_{i,j}$	local/intermediate offset of subtask $\tau_{i,j}$
d_i^k	absolute deadline of task job J_i^k
$d_{i,j}^k$	local absolute deadline of subtask job $J_{i,j}^k$
L_i	critical path length of DAG task τ_i
$Pred(\tau_{i,j})$	the set of predecessor subtasks of subtask $\tau_{i,j}$
$Succ(\tau_{i,j})$	the set of successor subtasks of subtask $\tau_{i,j}$
$Sibling(\tau_{i,j})$	the set of sibling subtasks of subtask $\tau_{i,j}$

Chapter 1

General Introduction

We start this thesis by a general introduction about the main axes of our work. Section 1.1 presents definitions and notations of real-time systems and their basic task models and components. Then, Section 1.2 provides a brief history description of processor development through the years until multiprocessor systems which are the center of our attention. Finally we focus on software development and its compatibility with hardware advancement by the use of parallelism in Section 1.3.

1.1 Real-time Systems

The *Oxford* dictionary defines **real-time** as "the actual time during which a process or event occurs". In computer science, a real-time system is defined as the system whose correctness depends on executing processes correctly within certain timing constraints. Recently, the term *real-time* is widely used to describe many applications and computing systems that are somehow related to time, such as real-time trackers, gaming systems and information services. The following list contains certain examples of practical real-time applications:

- Mobile and communication systems. For example, wireless communication systems in automotive and industrial applications which consist of a large number of nodes that require certain guarantees w.r.t. message passing and delays.
- Multimedia and entertainment systems: multimedia information is in the form of streaming audio and video. The communication between multimedia servers and receivers during information processing can have firm real-time requirements.

- Data distribution systems which notify users of important information in a short delay (few minutes or less). Such systems are found mainly in transport systems to inform passengers of accidents and schedule delays or changes.
- General purpose computing such as in financial and banking systems.
- Medical systems such as peacemakers and medical monitors of treatments or surgical procedures.
- Industrial automation systems such as the ones found in factories to control and monitor production process. For example, sensors collect parameters periodically and send them to real-time controllers, which evaluate the parameters and modify processes when necessary. These systems can handle non-critical activities as in logging and surveillance.
- General control management systems such as the ones found in avionic systems. Real-time engine controllers are responsible of automatic navigation and detection of hardware malfunctions or damages through reading sensors and processing their parameters and react within an acceptable delay. Another example is the air traffic control system which is classified as a critical application.

Based on these examples of real-time applications, we can notice that the criticality of the systems varies. Some applications require a strict respect of timing constraints such as in control and management systems. Whereas media and communication systems, for example, can tolerate timing delays without major consequences. Thus, real-time systems are categorized mainly into two groups, **hard** and **soft** real-time systems. In hard real-time systems, the correctness of their outputs depends on respecting given timing constraints or catastrophic results occur. If such systems fail in performing their tasks within acceptable deadline margins, their results become useless and might lead to catastrophic consequences.

Unlike hard real-time applications, soft real-time systems have flexible timing constraints and they perform less critical activities and tasks. The quality of services provided by soft real-time systems depends on providing results within a minimum delay. If such delay is not respected, the quality degrades but not the correctness of the execution or results.

In general, the most important aspect to be considered in real-time systems is to design applications and guarantee their execution within certain timing constraints before implementation. Also, such applications should have fault tolerance techniques and be reliable and robust against

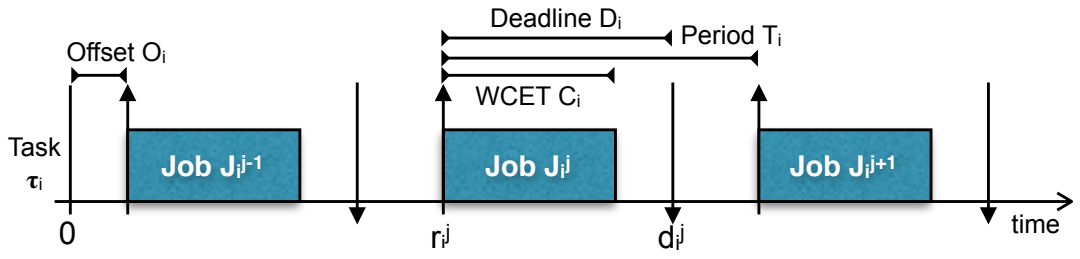


FIGURE 1.1: Timing parameters of an independent sequential real-time task.

unexpected modifications of system parameters. Possible solutions can be considered to guarantee such characteristics as implementing recovery systems, alternative designs and acceptance tests and analyses.

1.1.1 Real-time Task Model

A real-time application is composed of a set of tasks denoted by τ composed of n tasks where $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is assumed to generate one or more identical instances which are called jobs. The default real-time model [85] considers independent sequential tasks, and task τ_i has the following timing parameters:

- **Worst-Case Execution Time (WCET)** C_i which is an estimation of the longest possible execution time of any job of task τ_i, i.e., the actual execution time of a job should never exceed its WCET in any scenario. The evaluation of WCET of tasks is very important for the reliability of real-time systems, and the pessimism of their estimations increases relatively to the criticality of the application.

- **Relative deadline** D_i which is the time interval in which each job executes w.r.t. its release time. In hard real-time systems, jobs of any task must always meet their deadline, whereas execution tardiness of jobs is accepted in soft real-time systems.

- **Period or minimum inter-arrival time** T_i which is the time interval between successive jobs of task τ_i. A *periodic* task generates jobs which are activated periodically with identical time intervals between successive jobs, while a *sporadic* task generates jobs separated by at least T_i time units. If task τ_i generates a single job only, then its period is considered infinite.

- **Offset** O_i is defined as the time delay of the first job of the task w.r.t. a reference time 0.

Let J_i (respectively J_i^j) denotes any job generated by task τ_i (respectively the j^{th} job of τ_i). A job is an instance of the task with the same WCET and it is characterized by an absolute release time $r_i \geq O_i$ and an absolute deadline $d_i = r_i + D_i$. Figure 1.1 shows an example of the traditional real-time task model.

A real-time task is characterized by the relation between its deadline and its period:

- **Implicit-deadline**: task τ_i has a relative deadline equal to its period ($D_i = T_i$).
- **Constrained-deadline**: task τ_i has a relative deadline less than or equal to its period ($D_i \leq T_i$).
- **Arbitrary-deadline**: task τ_i has a relative deadline which can be less, equal or greater than its period.

In the case of implicit and constrained deadline tasks, there is at most one active job at any time instant t, while jobs having an arbitrary-deadlines may overlap and more than one job of such tasks can be active at any time t. The **slack time** of a task is defined as the time difference between the relative deadline of the task and its WCET. While the **laxity** of a job at time t is defined as the amount of time remaining for the active job to execute at time t. Also, the **response time** of a job is defined as the time required for it to be executed relative to its release time. The response time of a task is the maximum response time of all jobs. In order to calculate the response time of a task, **busy-period** approach can be used Lehoczky [78] defined this approach as follows:

Definition 1.1 (Level-i busy period [78])**.** A level-i busy period is a time interval $[a, b]$ within which jobs with priorities higher than τ_i are processed throughout $[a, b]$ but no jobs of priorities higher than τ_i execute in $(a - \epsilon, a)$ or $(b, b + \epsilon)$ for sufficiently small $\epsilon > 1$.

A task set is referred to as **synchronous** or **asynchronous** based on the first activation scenario of its tasks. A synchronous task set is defined as the task set whose first job of its tasks are activated at the same time. While the first jobs of an asynchronous task set are activated at different time.

The **processor utilization** of task τ_i is defined as the task's processor usage and it is denoted by $U_i = \frac{C_i}{T_i}$. The utilization $U(\tau)$ of a task set τ is the sum of utilization of its tasks, where $U(\tau) = \sum_{i=1}^{n} U_i$. The **density** δ_i of task τ_i is denoted by $\frac{C_i}{\min(D_i, T_i)}$. The density of an implicit-deadline task is equal to its utilization because its period is equal to its deadline. On a processing

platform of m processor, a necessary feasibility condition based on system utilization $U(\tau)$ is defined as follows:

$$U(\tau) \leq m \tag{1.1}$$

The workload of real-time tasks can be characterized by the **Demand Bound Function** and the **processor load** which are defined as follows:

Definition 1.2 (Demand Bound Function (DBF)[22])**.**
The Demand Bound Function (DBF^i) of a sequential task τ_i in a time interval $[0, t]$ for any $t > 0$ is defined as the sum of execution time of all jobs of τ_i that have both their arrival time and deadline in $[0, t]$.

$$DBF^i(t) = \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right)^+ * C_i \tag{1.2}$$

where $(x)^+ = \max(0, x)$.

Consequently, the processor load [21, 22] of task set τ is defined as:

$$load(\tau) = \max_{\forall t > 0} \left(\frac{\sum_{i=1}^{n} DBF^i(t)}{t} \right) \tag{1.3}$$

1.1.2 Real-time Scheduling

A real-time scheduling is defined as the process that defines the execution order of tasks on a platform of processors. An example of such systems is the Real-Time Operating System (RTOS) which is responsible for choosing which jobs to execute on which processor and at what time. The main objective of a real-time scheduler is to guarantee the correctness of the results while respecting the timing constraints of the tasks (no deadline miss). It is important to clarify that real-time scheduling does not necessarily mean executing tasks as soon as possible, but taking scheduling decisions that guarantee their timing constraints.

Based on the decisions of the scheduler, a real-time task can be in one of the following states:

- **Ready state**: the task is activated and it is available for execution, but it is not currently selected by the scheduler to execute on a processor.
- **Running state**: the task is assigned to at least one processor (according to its model of parallelism) and it is actually executing.

- **Blocked state**: if the task is waiting for an event to happen such as an I/O event, it remains blocked and cannot be scheduled until the event happens. Then the task moves to the ready state.

The different states of tasks are shown in Figure 1.2. Moreover, a real-time scheduler controls the transitions between the ready and running states of tasks, but it has no control over the external events that block the execution of tasks.

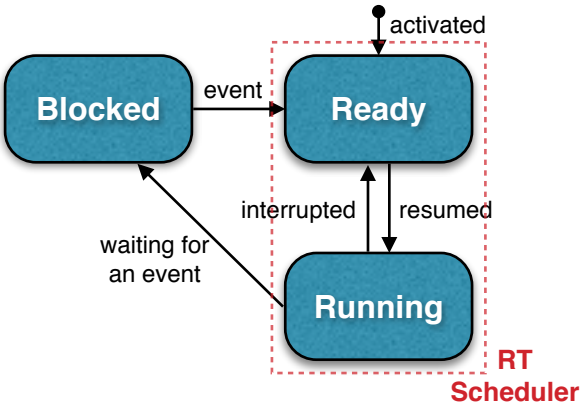


FIGURE 1.2: Different states and transitions of a real-time task.

Furthermore, a real-time scheduling is divided into two categories based on the scheduling decisions and when they are taken:

- **Offline scheduling**: a real-time system is scheduled based on a scheduling table that contains all scheduling decisions of the system and the activation times of all tasks. Hence, the scheduling decisions are taken prior to the running of the system and they rely on a knowledge of the process behavior.
- **Online scheduling**: scheduling decisions are taken during run time of the system based on certain priority assignment rules defined by the scheduling algorithm. It is used usually in dynamic systems where jobs may arrive or leave the system at any time, or when the scheduling table cannot be stored in the system (e.g. embedded systems).

1.1.3 Real-time Algorithms

A RTOS, which schedules tasks on a platform of processors, consists of certain scheduling algorithms (policies) that define the priority assignment of tasks and jobs and choose which

jobs to execute at which processor at what time. The scheduling algorithms are divided into three main categories:

- **Fixed Task Priority (FTP)**: each task in the set is assigned a fixed priority based on its timing parameters. In this category, all jobs of the same task inherit the same priority of their task. Examples of such algorithms is the Deadline Monotonic (DM) and the Rate Monotonic (RM) scheduling algorithms.
- **Fixed Job Priority (FJP)**: when a job is activated, it is assigned a fixed priority according to its timing parameters. Hence, priorities are assigned based on the activation events of jobs. As a result, various jobs of the same task may have different priorities. Example of this category is the Earliest Deadline First (EDF) algorithm.
- **Dynamic Priority (DP)**: the assigned priorities to jobs may vary as a function of time. Hence, the priority of a job may change during its execution. Example of such category is the Least Laxity First (LLF) scheduling algorithm.

Real-time scheduling is categorized based on the execution behavior of high priority tasks as follows:

- **Preemptive scheduling** in which running jobs are interrupted by higher priority jobs for some time, and they are allowed to continue their execution when all high priority jobs terminate their execution. An effect due to preemption by a higher priority job is referred to as *preemption effect*.
- **Non-preemptive scheduling** in which running jobs cannot be interrupted and the activation of a higher priority task can be delayed by the execution of at most one job of lower priority if this job is executing when the higher priority job is released. The effect due to execution blocking of high priority job by a lower priority one is referred to *blocking effect*.

A real-time scheduling algorithm is said to be **work-conserving** if it schedules ready jobs on available processors and it does not delay them if there are idle processors in the system.

Real-time Feasibility and Schedulability

For a given scheduling algorithm $\mathcal{A}$, a task is referred to as $\mathcal{A}$-**schedulable** if all of its jobs respect their deadline when scheduled with algorithm $\mathcal{A}$. Similarly, a task set is said to be $\mathcal{A}$-schedulable if all of its tasks are $\mathcal{A}$-schedulable. A task set is said to be **feasible**, if there is at least one scheduling algorithm that can schedule the task set while meeting all task deadlines. Additionally, a scheduling algorithm $\mathcal{A}$ is said to be **optimal** if all feasible task sets are $\mathcal{A}$-schedulable.

In general, a scheduling algorithm defines the scheduling decisions of a system which are necessary for its execution. Therefore, schedulability conditions can be defined to determine the status of the task set and whether it is schedulable or not using a given scheduling algorithm $\mathcal{A}$ before its implementation. There are three types of schedulability conditions w.r.t. algorithm $\mathcal{A}$:

- **Sufficient condition**: if the condition is true then the task set is deemed $\mathcal{A}$-schedulable, otherwise the schedulability of task set is undetermined, it can be schedulable or not.
- **Necessary condition**: if the condition is not true, then the task set is unschedulable using algorithm $\mathcal{A}$. Otherwise, the schedulability of the task set is undetermined, it can be schedulable or not.
- **Exact condition**: it is the combination of sufficient and necessary conditions. If the condition is true, then the task set is $\mathcal{A}$-schedulable. Otherwise, it is definitely unschedulable.

Comparability of Scheduling Algorithms

Real-time scheduling algorithms are compared with each other based on the scheduling results of task sets. For scheduling algorithms $\mathcal{A}$ and $\mathcal{B}$, their schedulability relationship can be as follows:

- **Domination**: $\mathcal{A}$ dominates $\mathcal{B}$ if all $\mathcal{B}$-schedulable task sets are also $\mathcal{A}$-schedulable but there is at least one task set that is schedulable by $\mathcal{A}$ and not schedulable by $\mathcal{B}$.
- **Incomparability**: if there is at least one task set that is $\mathcal{A}$-schedulable and it is not schedulable by $\mathcal{B}$. At the same time, there is at least one task set that is not schedulable by $\mathcal{A}$ but schedulable by algorithm $\mathcal{B}$.

- **Equivelance**: if all $\mathcal{A}$-schedulable task sets are also $\mathcal{B}$-schedulable. The same is applied on unschedulable task sets.

In order to compare the performance of real-time scheduling algorithms, Kalyanasundaram and Pruhs [73] introduced the **resource augmentation factor** (or speedup factor) as a comparison method. The resource augmentation factor of an algorithm $\mathcal{A}$ measure the required increase of processor speed for a feasible task set to become $\mathcal{A}$-schedulable on a system of the same number of processors.

Definition 1.3 (from [73])**.** For a given task set τ that is feasible on m unit-speed processors using an optimal scheduler, it is schedulable using $\mathcal{A}$ scheduling algorithm on m processors that are ν times faster. The minimum speedup factor of processors speed is the resource augmentation bound of scheduler $\mathcal{A}$.

Sustainability of scheduling algorithms

A scheduling algorithm is said to be **sustainable** [18] w.r.t. a task model, if and only if schedulability of any task set compliant with the model implies schedulability of the same task set modified by at least one parameter: (i) decreasing execution times, (ii) increasing periods or inter-arrival times and (iii) increasing deadlines. These timing changes are considered as positive changes of task set which should decrease the execution time of jobs or decrease their processor demand. Hence, negative scheduling results are not always intuitive.

Definition 1.4 (Scheduling anomaly)**.** A scheduling anomaly occurs when a positive change in task set parameters results in a counter-intuitive effect on schedulability.

Thus, a scheduling algorithm or a schedulability test that have no scheduling anomalies is called *sustainable*. This property is important to consider in scheduling design because scheduling analysis is done usually while considering the worst-case execution time of tasks, while in reality, jobs hardly execute up to this value. A system may be considered unreliable if a task set ceases to be schedulable after a reduction in the execution time or any other parameters (described above).

1.2 Development of Computer Processors

1.2.1 History of Processor Development

The concept of algorithms and modern computers is defined by Turing [115] who presented the *Turing Machine*. He proved that mathematical computations can be performed by machines if they are represented by an algorithm. Turing machines are considered as the central concept of modern computers.

The history of computer systems can be summarized into the following main generations:

- Vacuum Tube (1939-1954): at that time, computers were implemented using electron tubes (valves) of glass in which internal gas has been removed so as to control electron flow. The first working electro-mechanical programmable computer is the *Z3* which was designed by Konrad Zuse in 1939. Then, electronic programmable computers replaced the electro-mechanical one. This started by the design of the *Atanasoff–Berry Computer (ABC)* machine in 1942 by John Vincent Atanasoff and Clifford Berry.
- Transistors (1953-1958) : a transistor is defined as a semiconductor device which is used to amplify and switch electronic signals and electrical power. Transistors, which were invented in 1947, replaced vacuum tubes in the design of computer systems, because they are smaller than vacuum tubes and they consume less power. The first computer based on transistors was designed by Tom Kilburn in 1953 but it used also vacuum tubes in the design.
- Integrated Circuits (ICs) (1958-1971): they referred to chips on which set of electronic circuits are placed. The idea of ICs was presented by Geoffrey W.A. Dummer in 1952, but the first practical implementation of the idea was done by Jack Kilby in 1958. Modern computer systems are based on ICs, and they are divided mainly into two categories based on the number of available processing units: uniprocessor and multiprocessor systems.

1.2.2 Uniprocessor Systems

A uniprocessor platform is a system which consists of a single Central Processing Unit (CPU). All computations are done on this unit sequentially and a single task is executing at any time. Hence, no real (physical) parallelism is performed on such systems due to the hardware restriction,

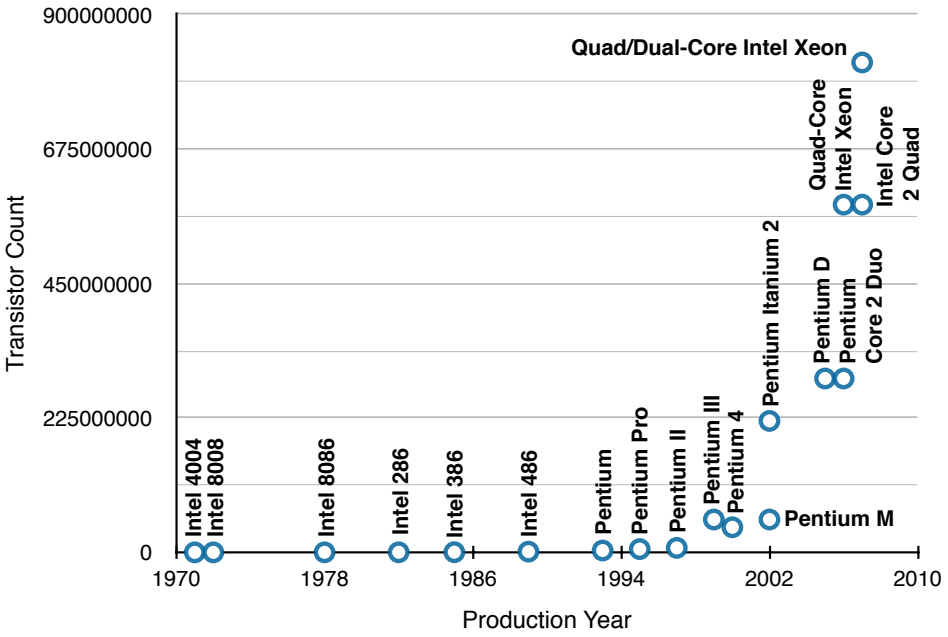


FIGURE 1.3: Evolution of *Intel* processor development showing their transistor count.

however, multitasking architecture is supported in such systems. Multitasking architecture in uniprocessor systems means that more than one task can be allocated to the processor and execute on time sharing basis.

Uniprocessor systems have been used in industry for decades and the performance of such systems was sufficient for the designed applications at that time. Even in real-time systems, scheduling on uniprocessor platforms was thoroughly studied since the 70s, and many algorithms and schedulability analyses were provided for such systems. However, software and applications are getting more and more complicated, and they require more powerful platforms to execute on.

In 1965, Moore [92] provided an observation regarding the industrial manufacturing of chips and ICs which is well known as the *Moore's Law*. Moore's observation states that the number of transistors in an IC doubles each two years, hence the performance of chips doubles as well. Although this observation was not presented as a physical law, it proved its correctness for more than forty years. For example, we can notice that the Moore's law is approximately followed by the development of *Intel* processors. The number of transistors integrated in their processors has exponentially increased during the years, as shown in Figure 1.3.

Based on Moore's law, the performance of processors grows exponentially by time due to doubling the number of transistors in ICs, which increases clock speed of processors and makes them execute tasks faster. In 1974, Robert Dennard [49] observed a relation between the size of

transistors and their power density. From what is later known by Dennard's scaling, the power density of ICs was expected to remain constant through the years if the size of transistors keep scaling down. Around 2005-2007, the Dennard's scaling seemed to break down due to energy leakage from transistors of small sizes. Due to these physical constraints of IC manufacturing, multicore/multiprocessor solution was presented instead of reducing transistor sizes and trying to maintain Dennard's scaling. Since clock rates of processors are not going faster in the same previous rate, platform performance is enhanced by duplicating the number of CPUs.

1.2.3 Multiprocessor Systems

Multiprocessor systems are defined as the systems which consist of more than one processing unit, hence, computations can be done faster and in parallel. Until recently, uniprocessor systems dominated the industrial execution platforms rather than the multiprocessor systems. It is either because system designers were afraid of the risk resulted from changing their previous stable uniprocessor designs, or that uniprocessor performance was sufficient for most embedded systems applications. However, with the increased demand for processing power and effectiveness for massive computations and applications, the shifting from uniprocessor to multiprocessor systems is on the way. Figure 1.4 shows the result of a study conducted by the VDC Research[1] in 2011. This study observed the percentage of industrial applications using single processor or more. We notice that in 2011, the percentage of uniprocessor applications was more than 30% of total applications. However, in few years this percentage is expected to drop to 15% in favor of multiprocessor platforms.

Consequently, personal computers (PCs) have been designed with multicore and/or multiprocessor chips in the last few years. The same has happened in the recent manufacturing of smart-phone systems.

Differently from multitasking architecture, multiprocessor systems support physical parallelism in hardware. Hence, concurrent tasks and processes can execute simultaneously on different processing units. Multiprocessing systems can be categorized as follows, based on their structure:

- **Multiprocessor system**: it consists of more than one processing unit that shares system memory through a communication bus. Applications and executing software have

[1] http://www.vdcresearch.com/

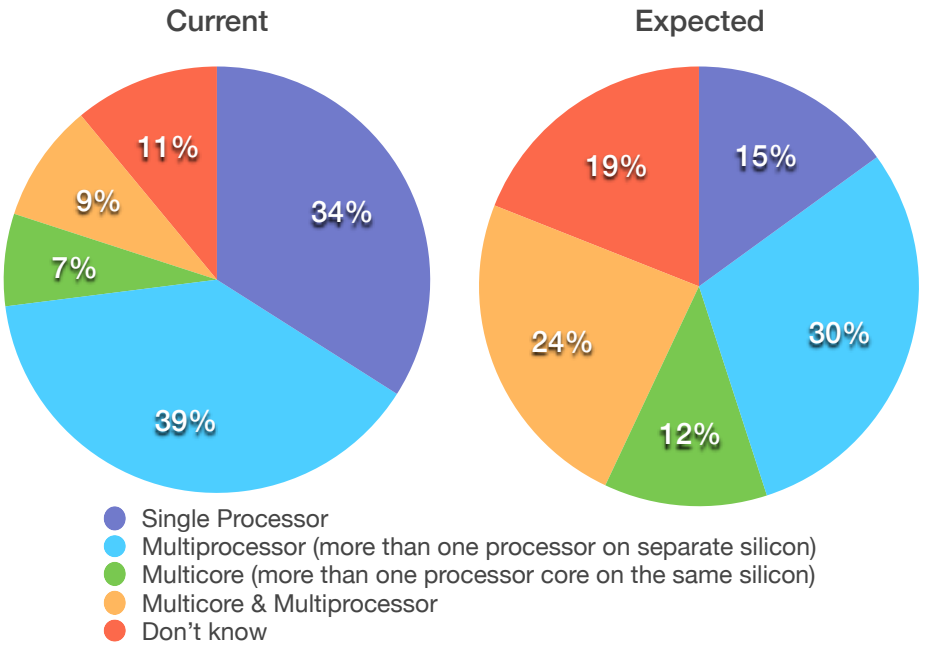


FIGURE 1.4: Shifting from uniprocessor to multiprocessor systems in industrial applications [64].

the choice of getting advantage of the performance boost from this architecture. In other words, sequential applications can choose to execute on such systems sequentially without the need of changing their programming techniques. Parallelism can be achieved by splitting processes/threads of an application if possible on different processors of the system to gain more execution speed.

- **Multicore system**: the core of a system consists of more than one logical unit. These units (usually between 2 to 8 cores) have their own individual memory cache and another level of cache memory which is shared between the different cores of a processor. The difference between multiprocessor and multicore systems lies in application programming. In order for an application to scale up its performance while executing on multicore systems, it has to change its sequential architecture. Multicore systems are found usually in personal computers (PCs) and smart-phones. Examples are Intel, AMD and ARM processors.

- **Many-core system**: this system has the same technical structure of multicore systems and it is used usually to refer to massive multicore systems. It consists of a large number of cores which is between dozens and hundreds in a single processor. Such systems implement parallel architecture, and software has to be adapted to such systems so as to run efficiently

and get advantage of hardware capabilities. For example, the CSX700[2] processor which is released by ClearSpeed in 2008 and has 192 cores, and the TILE-GX[3] 72-core processor from Tilera released in 2009.

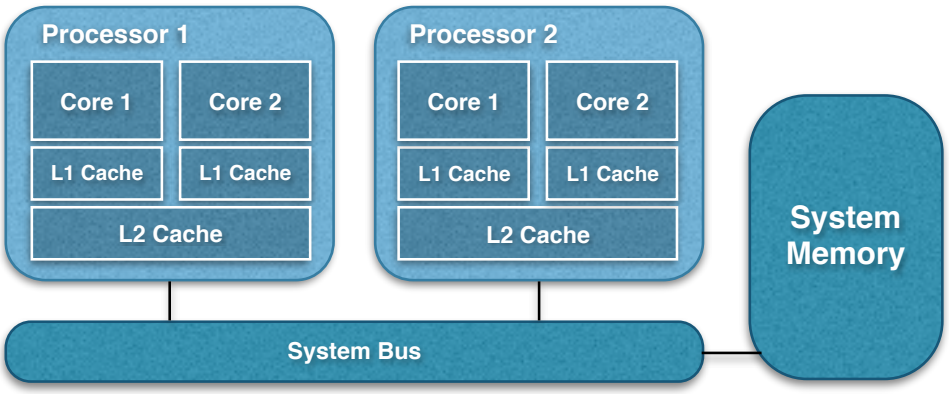


FIGURE 1.5: A multicore-multiprocessor system architecture.

The combination of the multicore and multiprocessor architectures forms a system which is referred to as a multicore multiprocessor system. In such systems, a processor consists of more than one core that shares cache memory, and there is more than one processor that shares the system memory by a bus. Figure 1.5 shows an example of such architecture.

Multiprocessor systems can be divided into categories based on the type of processors as follows:

- **Homogeneous or Symmetric Multi-Processors (SMP)**: all processors of the system are identical and the execution rate of tasks is the same on all processors.
- **Uniform Processors**: each processor is characterized by a speed or computing capacity, which determines the execution rate of a task.
- **Heterogeneous Processors**: Processors of the same system have different speed, and the execution rate of a task depends on the type of processor and on the task itself.

In the remainder of this thesis, we use the term "multiprocessor system" to refer to any platform which contains more than one CPU, which might be multicore, many-core or multiprocessor systems. Further more, we consider execution platforms of homogeneous unit-speed processors.

[2]CSX700: `http://www.clearspeed.com/products/csx700.php`
[3]TILE-GX: `http://www.tilera.com/products/processors/TILE-Gx_Family`

1.2.4 Real-time Multiprocessor Systems

The problem of scheduling real-time applications on multiprocessor systems is more complicated and challenging than real-time scheduling on uniprocessor systems. It is because there are more decisions to be taken in the case of multiprocessor scheduling and more issues to be considered. The responsibility of a real-time scheduler is not limited to decide which job to execute at what time, but also to decide on what processor to execute. Moreover, the multiprocessor scheduling problem can be seen as two parts, (i) allocating tasks/jobs to processors and (ii) assigning them priorities to be used by the scheduler.

Real-time multiprocessor scheduling is divided into the following categories based on migration level of jobs between processors:

- **Global scheduling** in which job migration is completely allowed at any time and for all jobs. Hence, a given job can start its execution on one processor and migrate to another during its execution. In such systems, overheads due to job migration can be costly (e.g. additional communication loads, context switching and cache misses).

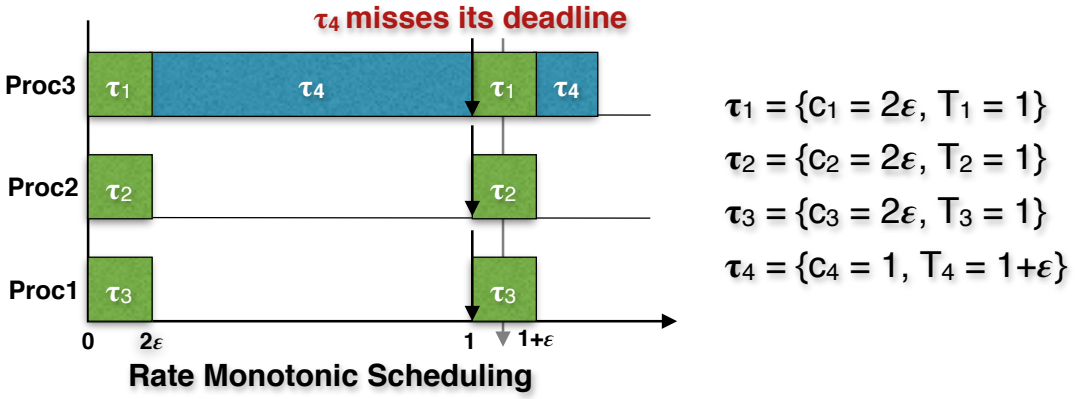


FIGURE 1.6: Example of the Dhall Effect on multiprocessor systems (From Dhall and Liu [51]).

Following our previous remark regarding the complexity of multiprocessor scheduling when compared to uniprocessors, Dhall and Liu [51] described a particular problem of multiprocessor scheduling which is called the *Dhall effect*. It shows that task sets can be unschedulable using common scheduling algorithms such as RM, DM and EDF regardless of the number of used processors, as shown in the example in Figure 1.6. Thus, the scheduling algorithms lose their scheduling properties and their optimality when used on multiprocessor systems. The Dhall effect discouraged researchers from considering multiprocessor systems until Phillips et al. [95] showed that Dhall effect was related to high utilization

tasks. Furthermore, Fisher [58] proved the impossibility of an online optimal scheduling algorithm for sporadic tasks on multiprocessor systems. This result is extended also to consider more general task models that include dependencies such as the generalized multiframe model and the recurring task model[4]. Global scheduling has many advantages such as acceptable overheads and context switches and avoiding the complexity of job assigning on processors.

- **Partitioned scheduling** allocates tasks to processors before scheduling and all of their jobs are forced to execute without the possibility of migration. The problem of multiprocessor scheduling is transformed into multiple uniprocessor scheduling problems on each processor of the system. The main concern regarding partitioned scheduling is that processor capacity can be fragmented and not fully used. The maximum utilization bound of such systems can reach 50% of total processing capacity for some cases (Dhall effect). Also, allocation process of tasks on processors is an NP-Hard problem and it requires bin-packing heuristics such as First Fit, Next Fit, Best Fit. However, classical uniprocessor scheduling algorithms can be used on each processor individually as a result.

- **Restricted-migration scheduling** allows tasks to migrate between processors at job boundaries (activation time of jobs) and never during the execution of jobs.

- **Semi-partitioned scheduling** in which most tasks are fixed to specific processors to reduce number of migration, while a few tasks migrate across processors to improve processor utilization.

1.3 Parallel Applications

A parallel application is defined as an application that performs its computations simultaneously on multiple processors. Parallelism is important so as to get advantage of hardware advancement of processor architecture such as multiprocessor and many core systems. It is important to notice that increasing the performance of execution platforms by duplicating their processing units is useless if designed software is not compatible with the hardware. As in the case of industrial projects, increasing the number of workers of a project does not accelerate the development of the project instantly. It is done by modifying management strategies so as to distribute tasks between workers. The same concept can be applied to software design so as to consider

[4]For more details, please refer to Subsection 1.3.3 "General Parallel Real-time Task Models"

parallelism and avoid sequential design whose performance does not scale up by the use of multiprocessors.

Software parallelism is divided into the following categories:

- **Inter-task parallelism** in which tasks execute in parallel. A set of tasks which executes on multiple processors is an example of such parallelism.
- **Intra-task parallelism** (inter-subtask parallelism): A parallel task consists of portions (subtasks) which execute in parallel.
- **Intra-subtask parallelism**: A subtask of a parallel task consists of a set of threads that execute in parallel. Hence, a subtask needs more than one processor to execute.

However, there is a limitation of software parallelism represented by the *Amdahl's law* which states that the performance of an application cannot be enhanced infinitely when executed in parallel. Amdahl [3] described the relationship between the speedup of parallel execution of an application relative to the serial execution. The speedup of a program using multiple processors in parallel computing is limited by the time needed by the sequential fractions of the program to execute. For example, if an application needs x time units to execute sequentially on a single processor, and among this execution, there is a strictly-sequential portion of code that needs y time units (where $y \leq x$), then this application requires at least y time units to execute even on unlimited number of processors.

1.3.1 Parallelism in Uniprocessor Systems

As stated earlier, a uniprocessor system consists of a single processing unit (CPU). Hence, there is no actual physical parallelism applied on such systems and only a single process can execute at any time instant. However, this does not mean that only a single application or task executes in a given time interval on uniprocessor systems. Multiple applications can run at the same time on uniprocessor systems by supporting multitasking architecture. This is done based on time sharing and by allocating some resources to each application. Hence, multiple processes seem to be executing at the same time but actually only one process is executing on the available processor of the system at every time slot. Multiple processes are queued and wait for their turn to be executed on CPU, but it appears as if they were running in parallel. The switching between executing processes is done fast enough to be transparent for a human, and thus it

appears that the system is running all processes simultaneously. So multitasking is a software technique related to the operating system rather than physical parallelism on hardware.

1.3.2 Parallelism in Real-time Systems

In this document we assume that a parallel real-time task consists of a set of subtasks. A *subtask* is an execution portion of the task which is characterized by a WCET. Similarly, a *thread* is defined as an execution portion of a parallel task (or even a subtask). Threads of a task execute usually within a segment and they are all activated at the same time and have to terminate their execution at the same time. All threads of a segment have the same WCET.

Moreover, a real-time job/task is said to be:

- **Rigid**, if the number of processors assigned to this job/task is specified a priori and does not change throughout its execution.
- **Moldable**, if the number of processors assigned to this job/task is determined by the scheduler and does not change throughout its execution.
- **Malleable**, if the number of processors assigned to this job/task can be changed by the scheduler during its execution.

1.3.3 General Parallel Real-time Task Models

There are different models of real-time parallel tasks which are shown in Figure 1.7:

- **Independent task model** which has been defined by Liu and Layland [85] and is considered as the basic model of tasks where each one consists of a single vertex.
- **Multiframe model** proposed by Mok and Chen [91]: this task model is a generalization of the independent task model. A multiframe task generates an infinite succession of frames which are separated by a minimum separation time. The WCET of the task is not constant and it is defined according to a cyclic pattern.
- **The Generalized MultiFrame (GMF) model** presented by Baruah et al. [25]: it is a generalization of the Multiframe model. The deadlines of frames are allowed to differ from the minimum separation time. Frames have different deadlines and minimum separation times.

- **Recurring branching** [14]: this model allows selection points to determine the execution behavior of a given task instance, such as conditional statements "if-then-else" and "case". The corresponding task graph is a directed tree.
- **Recurring (Directed Acyclic Graphs)** [15]: it defines a task as a Directed Acyclic Graph (DAG) which is the most general model of parallel tasks and the other parallel models can be represented as special cases of the DAG model.

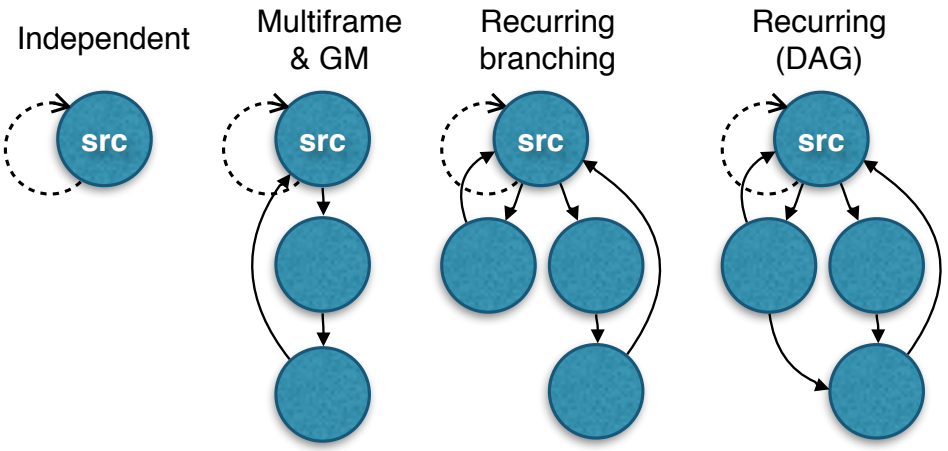


FIGURE 1.7: Different models of parallel tasks [119].

1.3.4 Directed Acyclic Graph (DAG)

In this thesis we concentrate on the common model of parallelism which is the DAG model from the recurring task model. A DAG task consists of a set of subtasks (execution portions) with precedence constraints. Let τ_i be the i^{th} DAG task in the set τ where $1 \leq i \leq n$. Task τ_i consists of a set of subtasks under precedence constraints that determine their execution flow, and it is characterized by $(\{\tau_{i,j} | 1 \leq j \leq n_i\}, G_i, O_i, D_i, T_i)$, where the first parameter represents the set of subtasks of τ_i and n_i is their number. Each subtask is denoted by $\tau_{i,j}$ where $1 \leq j \leq n_i$. Parameter G_i is the set of directed relations between these subtasks, O_i is the offset of the DAG, D_i is τ_i's relative deadline and T_i is its period (or minimum inter-arrival time between jobs). Each subtask $\tau_{i,j}$ is characterized by a specific WCET $C_{i,j}$. A job J_i of DAG task τ_i consists of a set of subtask jobs based on the same inter-subtask parallelism of the DAG. Let $J_{i,j}$ (respectively $J^k_{i,j}$) denotes a job of subtask $\tau_{i,j}$ (respectively, the k^th job of subtask $\tau_{i,j}$) and it is characterized by an absolute release time $r_{i,j} \geq O_i$ (respectively $r^k_{i,j}$) and an absolute deadline $d_{i,j} \leq D_i$ (respectively $d^k_{i,j}$).

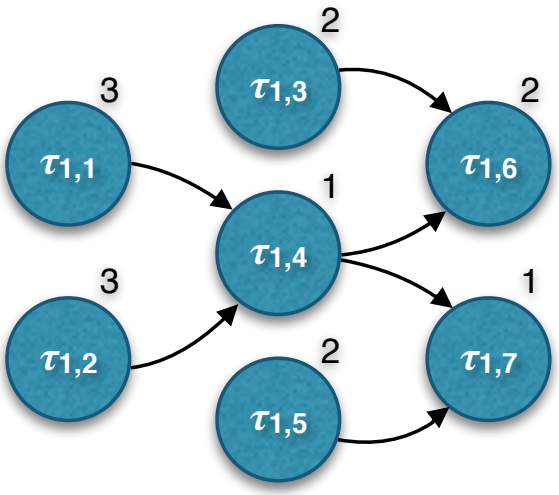


FIGURE 1.8: An example of a DAG task τ_1 which consists of 7 subtasks.

The activation of subtasks of a DAG depends on its structure. A **source** subtask has no inputs and it is activated by the activation of its DAG. A **sink** subtask is an ending subtask in the DAG which does not generate outputs used by other subtasks. When subtask $\tau_{i,u}$ uses the outputs of subtask $\tau_{i,v}$ as its inputs, then subtask $\tau_{i,v}$ is considered as its **parent** subtask while subtask $\tau_{i,u}$ is a **child** subtask. More generally, a **predecessor** subtask is an indirect parent, while a **successor** subtask is an indirect child. Let $Pred(\tau_{i,u})$ (respectively $Succ(\tau_{i,u})$) be the set of all predecessor subtasks of $\tau_{i,u}$ (respectively, the set of all of its successor subtasks). We assume that a DAG task has at least one source subtask and one sink subtask.

The dependencies between subtasks are due to the inter-subtask parallelism of DAG tasks and they determine their execution flow. However, parallelism is not mandatory and it is based on the decisions of the real-time scheduler. A DAG task may be executed sequentially as a chain if necessary.

The global timing characteristics of a DAG task are the same as the common sequential task model. We suppose that a DAG generates an infinite number of jobs separated by a minimum (or exact) inter-arrival time and a deadline to be respected by each job. The WCET of a DAG task is the sum of WCET of all of its subtasks, where $C_i = \sum_{j=1}^{n_i} C_{i,j}$.

An example of a DAG task is shown in Figure 1.8 which shows the structure of the DAG model. In this example the circles in the figure represent the subtasks of the DAG task while the arrows represent their precedence constraints and the execution behavior of the DAG task. The number on the upper right side of each subtask represents its WCET.

Determining the topology of a DAG task is important in real-time scheduling so as to choose the scheduling order of subtasks. There are many techniques and algorithms to traverse DAG tasks such as the number of children or parent subtasks or based on their timing characteristics. However, the most common technique used in real-time scheduling is based on the critical path of the DAG.

Definition 1.5 (DAG's critical path)**.** For a given DAG task, its critical path is defined as the path of subtasks with the longest sequential execution time among all other paths of the DAG when it executes on a system of unlimited resources (e.g. number of processors).

Let a critical subtask be any subtask in the critical path of the DAG. Based on the definition above, a DAG task can be seen as a sequence of critical subtasks executing sequentially, while non-critical subtasks execute in parallel with it. Moreover, the critical path length L_i is considered as the minimum response time of the DAG. For example, the critical path of task τ_1 from Figure 1.8 is defined as $\{\tau_{1,1}, \tau_{1,4}, \tau_{1,6}\}$ with length equal to 6. The choice between subtask $\tau_{1,1}$ and $\tau_{1,2}$ is done arbitrarily.

A DAG task is said to be feasible if of all of its subtask jobs respect their absolute deadlines. For any given DAG set, there are two trivial necessary feasibility conditions. A DAG set τ is deemed unfeasible when scheduled using any scheduling algorithm on m unit-speed processors if, at least, one of the following conditions is false:

- The total utilization $U(\tau)$ of a task set τ, which is the sum of utilization of its DAGs, should be less than the number of processors in the system ($U(\tau) \leq m$).
  Regarding the case of identical unit-speed processors, the system's capacity within an interval of length t cannot exceed $(m * t)$ execution time units. Hence, it is impossible for any task set to be feasible on a system of unit-speed processors if its processing demand exceeds this value and if processors are overloaded.

- The deadline of any DAG task should be at least as long as its critical path length ($\forall \tau_i \in \tau : L_i \leq D_i$).
  The critical path length of a DAG task represents the minimum response time of the DAG task, and it is impossible for a DAG task which executes on unit-speed processor systems to finish its execution earlier than this value. In order for a DAG to be feasible, it should finish its execution not later than its deadline, hence, its relative deadline should be at least as long as its critical path length.

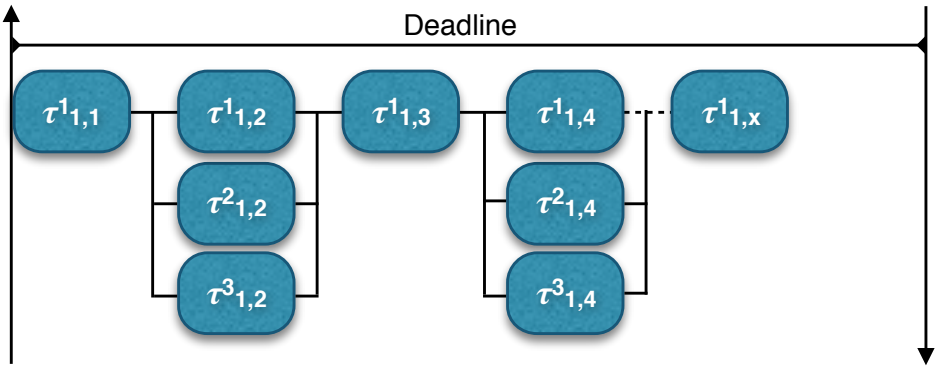


FIGURE 1.9: Fork-Join model of parallel real-time tasks.

By default in this document, we consider global preemptive scheduling of periodic implicit-deadline DAG sets which execute on homogeneous unit-speed processors.

1.3.4.1 Special Parallel Model: Fork-Join Tasks

The Fork-Join (FJ) model of parallel tasks is well known as the execution model of the famous *OpenMP* [1] framework and Cilk programming language[5]. In this model, a task is defined as an alternative sequence of sequential and parallel segments. It consists mainly of a master thread that forks into multiple threads of the same WCET to form a parallel segment. When all threads terminate their execution they merge again into the master thread.

Figure 1.9 shows an example of a fork-join model. In this document, we consider that all parallel segments consist of the same number of threads. All threads of the same segment have the same WCET. The master thread of the FJ task represents its critical path and its length is the minimum sequential execution time of the task.

1.3.4.2 Special Parallel Model: Multi-Threaded Segment Tasks

Another model of parallel tasks is the Multi-Threaded Segment (MTS) model, which is a more general version of the FJ model but still a special case of the Directed Acyclic Graphs. Instead of having alternative sequence of sequential and parallel segments as in the Fork-Join model, a MTS task consists of a sequence of parallel segments, where each one consists of at least one thread. A segment is activated when all threads of the predecessor segment terminate their

[5] `https://software.intel.com/en-us/intel-cilk-plus`

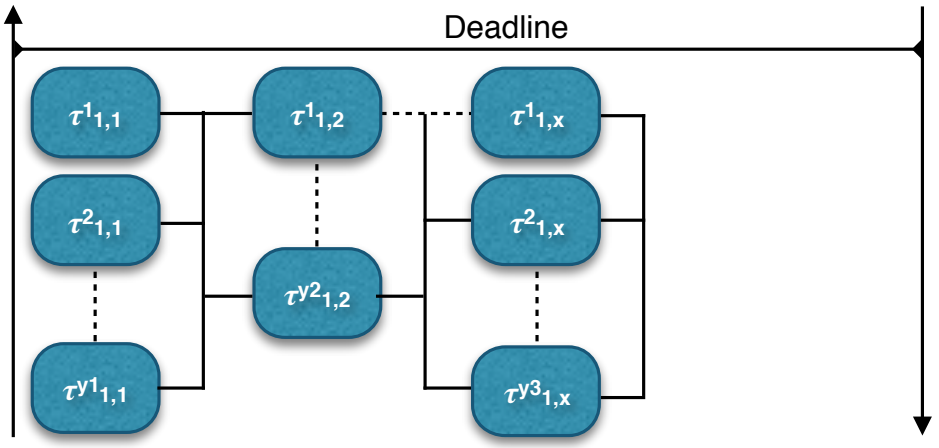


FIGURE 1.10: Multi-Threaded Segment model of parallel real-time tasks.

execution, and similarly, its successor segment becomes active after it terminates its execution. A segment that consists of a single thread is considered as a sequential segment.

1.4 Problem Description and Contributions

In this thesis, we are interested in studying the problem of global preemptive scheduling of parallel real-time DAG tasks on homogeneous multiprocessor systems. Based on the DAG model, the execution order of their subtasks is dynamic, i.e., a subtask can execute either sequentially or in parallel with its siblings based on the decisions of the real-time scheduler. To this end, we analyze two DAG scheduling approaches to determine the execution order of subtasks: the Model Transformation and the Direct Scheduling approaches. The Model Transformation converts parallel tasks into other models to simplify the real-time scheduling, and the Direct Scheduling adapts the scheduling algorithm and feasibility analysis of the DAG model so as to take into consideration the dependencies and the precedence constraints to schedule DAGs directly.

List of Contributions & Thesis Outline

- Chapter 2 (Related Work).
- Chapter 3 (Scheduling of Parallel Tasks using Model Transformation):
  Concerning the Model Transformation approach, we propose a DAG Stretching (DAG-Str) algorithm which is an extension of a stretching algorithm for the parallel FJ task model studied in [77]. The general structure of the DAG model makes stretching transformation

more challenging. The concept of DAG-Str algorithm is to force parallel tasks to execute as sequentially as possible and avoid their parallel structure whenever it is possible. We analyze our DAG-Str algorithm by showing its resource augmentation bound in the case of global preemptive Earliest Deadline First scheduling algorithm.

Moreover, we prove that global preemptive EDF scheduling of DAG tasks when DAG-Str is used has a resource augmentation bound equal to $\frac{3+\sqrt{5}}{2}$ for all task sets with $n < \varphi.m'$, where n is the number of tasks in the task set, φ is the golden ratio[6] and m' is the number of available processors in the system after stretching. Otherwise, the resource augmentation bound is equal to 4. Recently, the same resource augmentation bound of GEDF has been proved in [82] in the case of Direct Scheduling of DAG tasks.

- Chapter 4 (Direct Scheduling Approach of Parallel DAG Tasks):
  In the case of Direct Scheduling, many researches from the state-of-the-art have been interested in this problem but they mainly consider DAG scheduling without including their internal structure in the scheduling. In this work, we show that including extra information about the internal structure of DAGs and adding extra timing parameters of their subtasks enhance the scheduling process of DAGs and helps real-time scheduler to take better scheduling decisions adapted to the DAG model.

  In this work, we divide Direct Scheduling into two categories, DAG-Level scheduling and Subtask-Level scheduling. The DAG-Level scheduling executes DAGs based on their global timing parameters without considering the parameters of their subtasks. Our contribution lies in proposing a scheduling analysis that takes into consideration the internal structure of DAGs, and we provide a sufficient schedulability condition based on the workload analysis which is aware of the internal structure of DAGs.

  In Subtask-Level scheduling of DAGs, a real-time scheduler takes decisions based on the timing parameters of subtasks and not DAGs. Before scheduling, we assign extra local timing parameters to subtasks that are extracted from the global parameters of DAGs. We provide a scheduling analysis of global EDF at a Subtask-Level and we provide sufficient schedulability conditions based on interference and workload analyses.

- Chapter 5 (Experimental Analysis of DAG Task Scheduling):
  We start this chapter by proving the incomparability of DAG scheduling approaches while using global preemptive multiprocessor scheduling while using Earliest Deadline First and Deadline Monotonic scheduling algorithms. We provide DAG scheduling examples so as

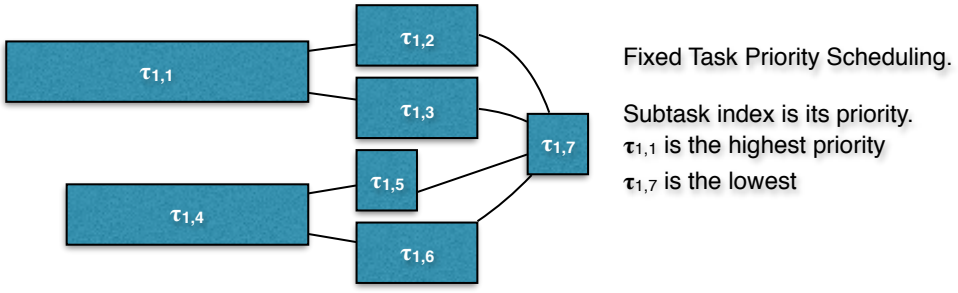
[6] The value of the golden ratio is $\frac{1+\sqrt{5}}{2}$

to prove that these approaches are not comparable, i.e., there exist task sets that are schedulable using one scheduling approach and they are unschedulable using the other one, and vice versa.

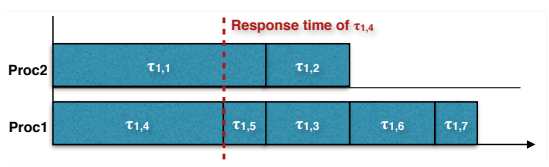
Due to the incomparability of DAG scheduling approaches, we use extensive simulations to compare their schedulability performance. To this end, we present *YARTISS* which is a general multiprocessor simulation tool written in Java and developed at our *LIGM* research Lab. *YARTISS* is designed specifically in a way that enables users to extend it and add modules to it easily. Using *YARTISS*, we generated DAG sets randomly and we performed extensive simulations to compare schedulability performance of DAG scheduling approaches.

Anomalies in DAG scheduling

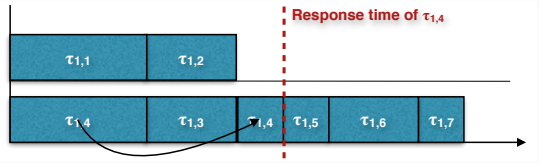
In this thesis, we consider that each real-time task/subtask execute up to its WCET at all times, because DAG scheduling suffers from execution anomalies. DAG scheduling algorithms can be unpredictable when the actual execution time of a task is less than its WCET. Ha and Liu [66] proved that global preemptive fixed task priority scheduling algorithms for independent sequential tasks on multiprocessor systems are predictable. However, this is cannot be extended to the case of DAG scheduling. An example is shown in Figure 1.11 which presents the scheduling



(a) Example of a DAG task τ_1. The index of a subtask $\tau_{1,j}$ represents its priority where $j = 1$ is the highest priority and 7 is the lowest.



(b) The scheduling of the first job of τ_1 when all jobs execute up to their WCET.



(c) Subtask $\tau_{1,1}$ job executes for 3 time units instead of 5.

FIGURE 1.11: An example showing anomaly of DAG scheduling when Fixed Task Priority scheduling algorithm is used.

of a single DAG task τ_1 on 2 processors using preemptive fixed priority scheduling algorithm. The DAG task consists of 7 subtasks and their priorities are the numbers inside each subtask in the figure. Inset 1.11(b) shows the scheduling of a DAG task τ_1 when its WCETs are considered. The response time of the DAG is equal to 10. However, when the first subtask executes for 3 time units instead of 5, this causes successor subtasks to be activated earlier and this leads to increase the response time of the scheduled DAG. Hence, a positive change in the execution time of the DAG has negative results. The scheduling is not sustainable w.r.t. execution time.

Due to these scheduling difficulties and the special characteristics of DAG tasks, we concentrate our work in this thesis on including knowledge about the internal structure of DAGs in the scheduling process. Unlike DAG scheduling methods and approaches found in literature, our method aims at extracting extra information about subtasks that are not provided by the model and include them in the scheduling process. We believe that by doing so, we can enhance the scheduling of DAG tasks on multiprocessor systems to obtain a sustainable scheduling.

Chapter 2

Related Work

In this chapter, we briefly present the main algorithms and results regarding real-time scheduling in uniprocessor and multiprocessor systems which are found in the state-of-the-art. Section 2.1 concentrates on uniprocessor systems and we review the main scheduling algorithms which are proved to be optimal. Section 2.2 describes the effect of multiprocessor scheduling on the optimality of uniprocessor algorithms and their performance in term of schedulability. Then, we present some multiprocessor algorithms (optimal and variations of optimal uniprocessor algorithms). Finally, Section 2.3 focuses on parallelism in real-time systems and the main scheduling algorithms and approaches proposed to overcome the challenge due to parallelism. The section is divided into two subsections to present parallelism in the case of uniprocessor and multiprocessor systems.

2.1 Real-time Scheduling of Uniprocessor Systems

Most of real-time scheduling research began in 1973 when Liu and Layland [85] analyzed the preemptive scheduling problem of hard real-time systems on uniprocessor platforms. Among their contributions, the authors provided optimal scheduling algorithms based on Fixed-Task (FTP) and Fixed-Job (FJP) Priority assignment for independent sequential tasks (sporadic or periodic). They proved the optimality of Rate Monotonic (RM) FTP scheduling algorithm for sporadic and synchronous periodic implicit-deadline tasks and provided the following schedulability condition for a task set τ of n tasks:

$$U(\tau) \leq n(2^{1/n} - 1)$$

So in the general case, RM scheduling algorithm can meet all the deadlines in task set τ if its utilization $U(\tau)$ is not greater than $ln2 \approx 0.69$ (when the number of tasks n tends towards infinity). In the case of FJP class, Liu and Layland proved the optimality of Earliest Deadline First (EDF) scheduling algorithm for arbitrary-deadline tasks for all sets with utilization less than 1.

It is interesting to note that uniprocessor FTP scheduling algorithms gained great practical importance when compared to EDF despite the dominance in schedulability performance of the latter. However, the advantage of FTP algorithms is that they are generally easier to implement and to analyze. Furthermore, they cause less system overheads (mainly due to job preemption and context switches), and system schedulability can be enhanced by simply modifying priority assignment of tasks without modifying the scheduler.

In 1982, Leung and Whitehead [80] proved that Deadline Monotonic (DM), which is a FTP algorithm, is optimal for sporadic and synchronous periodic uniprocessor sets with constrained-deadline tasks (deadlines are not greater than periods). Moreover, the authors proved that DM is not optimal in the case of asynchronous periodic task sets. A while later in 1990, Lehoczky [78] proved that DM is not optimal also in the case of arbitrary-deadline tasks.

Oppositely to the above-described FTP algorithms, Audsley [8, 9] presented an optimal fixed priority scheduling algorithm which is known by *OPA*. It stands for **O**ptimal **P**riority **A**ssignment and it is proved to be optimal for asynchronous periodic arbitrary-deadline tasks. The authors proved that OPA can determine priority ordering of tasks in a given set in polynomial time which is not greater than $n(n+1)/2$ schedulability tests.

Regarding scheduling algorithms which are based on Dynamic Priority (DP) assignment (e.g., Least Laxity First algorithm), they are not usually used in uniprocessor systems because their implementation is complicated when compared to FTP and FJP classes, and they can cause higher system overheads. The use of other simpler optimal scheduling algorithms such as EDF is a better choice and it avoids such development and execution complications.

2.2 Real-time Scheduling of Multiprocessor Systems

Originally, the concept of multiprocessor platforms has been presented a long time ago. The SIMD (Single Instruction, Multiple Data) model for instance, which is a class of parallel computers, was firstly presented in 1958 by Unger [117]. The idea of increasing processing performance

by duplicating processors was found interesting and appealing by many researchers as stated by Bouknight et al. [37]:

> The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of re-centralizing one of the three major components ... Centralizing the [control unit] gives rise to the basic organization of [an] ... array processor ...

However, the need for hardware development was not pressing at that time which delayed its evolution and allowed uniprocessor systems to dominate industrial projects and platforms. The same observation can be applied in real-time systems, it has been identified a long time ago by Liu [84] that real-time multiprocessor scheduling problem is challenging. In addition to the problem of deciding which job to execute and at what time, real-time schedulers have to decide on which processor these jobs have to execute as well. This concern was expressed by Liu as follows:

> ... 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...

As predicted by Liu, many optimal uniprocessor scheduling algorithms were proved to lose their optimality when used on multiprocessor systems due to their scheduling complications such as the Dhall's effect [51]. For instance, in the case of fixed priority assignment, Dhall and Liu proved that scheduling performance of Rate Monotonic and Deadline Monotonic degrades on partitioned multiprocessor systems and they fail at low utilization. A significant amount of research has been done to enhance the performance of such scheduling algorithms by proposing hybrid priority assignment techniques in which tasks are divided into groups that are assigned priorities based on different assignment rules. Andersson et al. [7] proposed RM-US (Rate Monotonic with Utilization Separation) which assigns highest priorities to tasks whose utilization is above a given threshold and the remaining tasks are scheduled using RM. Similarly, Bertogna et al. [31] provided DM-DS (Deadline Monotonic with Density Separation) which uses DM instead of RM.

Similarly, FJP scheduling algorithms suffer from the same problem of FTP class. For instance, EDF scheduling algorithm loses its optimality when extended to multiprocessor systems, but it remains as an interesting multiprocessor scheduling algorithm and many variations have been proposed to enhance its performance. In 2002, Srinivasan and Baruah [112] provided EDF-US (Earliest Deadline First with Utilization Separation) which schedules a group of tasks with static priorities and the remaining tasks using EDF. Bertogna [28] used task densities instead of utilization in EDF-DS (Earliest Deadline First with Density Separation) to schedule sporadic task sets with constrained and arbitrary deadlines.

In 2004, Baruah [16] provided a fixed priority scheduling algorithm of periodic tasks on a platform of m identical processors, which assigns highest priority to the first $(m-1)$ tasks having utilization greater than one half, and schedules the remaining ones with EDF. The algorithm leads to a utilization at most equal to $(m+1)/2$ which is the maximum utilization that can be achieved by fixed priority algorithms. The same utilization bound has been proved for global multiprocessor FJP scheduling with implicit-deadlines in [7]. Another algorithm called EDF(k), proposed in [12, 63], assigns highest priorities to the k tasks with the highest utilization and the remaining tasks are scheduled using EDF.

Dynamic priority scheduling algorithms, in which priority of a job changes during its execution, are useful for multiprocessor systems. This priority assignment overcomes multiprocessor scheduling complications such as the Dhall's effect. A number of scheduling algorithms based on dynamic priority (e.g., PFair, RUN and U-EDF) are proved to be optimal in the case of periodic task sets with implicit deadlines.

PFair (**P**roportionate **Fair**ness) is an optimal multiprocessor scheduling algorithm which was introduced by Baruah et al. [24] in 1996 for periodic and sporadic global scheduling of implicit-deadline task sets. PFair utilizes the full capacity of processors by scheduling successfully any task set whose utilization is not greater than the number of processors. It is based on the concept of slots, i.e., the time line is divided into equal quanta. Each quantum is assigned to tasks fairly by the scheduler. The main disadvantage of fairness-based scheduling algorithms lies in the generated overheads of job execution and the huge number of job migrations and context switches forced by the scheduler. There are many variations to enhance its performance and reduce overheads such as PD [23], SA [75], ERFair [4], PD^2 [5], BF [122] and LLREF [43].

Another optimal multiprocessor scheduling algorithm for periodic implicit-deadline tasks called *RUN* was introduced by Regnier et al. [106] in 2011. The RUN algorithm, which stands for

Reduction to **Uni**processors, is a real-time scheduler which reduces the multiprocessor problem into a series of uniprocessor problems which are scheduled using EDF. Unlike other scheduling algorithms, RUN schedules the idle time of tasks rather than their execution time, and it is proved to have an upper bound of $O(\log m)$ average preemptions per job on m processors. Similarly to PFair, it can successfully schedule task sets whose utilization is not greater than the number of processors in the system.

In 2012, Nelissen et al. [94] introduced *U-EDF* which is an optimal multiprocessor scheduling algorithm for sporadic implicit-deadline tasks and for sporadic tasks when their total density is smaller than the number of processors in the system. The concept of this algorithm is summarized into two main steps: when a job is released, (i) U-EDF pre-allocates execution time to active tasks in a horizontal manner (tries to maximize utilization of first processor, then the second one, ...) so as to save enough execution resources for future jobs that might be released and interfere with currently active ones. (ii) Based on these allocations, active jobs execute using EDF scheduling algorithm which is slightly modified so as to prevent parallel execution within jobs.

There are many other scheduling algorithms and solutions which can be found in the state-of-the-art regarding multiprocessor schedyling problem. In 2011, Davis and Burns [48] gathered and summarized the main results regarding this problem in a general survey which is an excellent reference.

2.3 Parallel Real-time Scheduling

The development of multiprocessor systems and their growing importance in industrial applications nowadays encourage researchers to analyze the evolution of software in real-time systems as well. They considered more general task models which contain parallelism and precedence constraints rather than the classical sequential model of real-time tasks. Thus, different formal models have been proposed for representing recurrent tasks; these models differ from one to another in the restrictions they place on the jobs that may be generated by a single task [50, 83–85, 113].

In this section, we briefly present the main scheduling algorithms and their associated schedulability conditions regarding the problem of scheduling parallel dependent real-time tasks on

uniprocessor and multiprocessor systems. We highlight the main scheduling techniques used in the state-of-the-art in order to take into account parallelism in real-time systems.

2.3.1 Parallel Scheduling on Uniprocessor Systems

In general, when parallelism is mentioned we think directly of physical parallelism which is associated with distributed and multiprocessor systems, where multiple processes execute simultaneously on the available processing units of the system. Applications with parallelism and precedence constraints were first implemented on uniprocessor systems due to the practical importance of such systems in industrial embedded systems. In real-time systems, the wealth of scheduling algorithms and analyses of uniprocessor systems makes them a tempting execution platform even in the case of parallel tasks. There had been some researches which proposed to simplify the parallel scheduling problem by model transformation. A parallel dependent task is converted into sequential chains which are scheduled on uniprocessors using common scheduling algorithms. It is important to determine the execution order of subtasks within chains which enhances the schedulability of task set. In this subsection, we will describe the related researches found in the state-of-the-art.

The Canonical Scheduling Form

In 1991, Harbour et al. [68, 69] studied the problem of scheduling dependent periodic tasks on uniprocessor systems. Each task is composed of sequential subtasks which are characterized by specific WCETs and fixed priority. Thus, each parallel task may have different priorities during its execution based on the priority of its current running subtask. There is no intra-subtask parallelism in this model, i.e., subtasks are sequential threads and they are similar to the classical uniprocessor model with the only exception of the multiple priorities.

In this model, a scheduling difficulty arises due to the different priorities within the same parallel task, which complicates the calculations of the worst-case response time of the task. In the case of the classical real-time task model, the worst-case response time of a certain task occurs in the case of synchronous task sets, in which its job is activated at the same time with all higher priority jobs in the set. Hence, a schedulability test is required to verify the finish time of the task with the lowest priority in the system and check if it is less than its deadline. The authors showed an example to prove the inaccuracy of this scenario in the case of dependent tasks. They

concluded that a correct schedulability test has to check the deadlines of more than one job of a particular task in the busy period.

In order to simplify the scheduling analysis of dependent tasks and reduce the number of different priorities of subtasks, Harbour et al. [68] introduced a canonical form of dependent tasks which changes the priorities of its subtasks. It is defined as follows:

Definition 2.1 (Canonical task form)**.** A task is said to be *canonical* if it consists of consecutive subtasks that increase in priority.

The transformation into the canonical form is done by changing the priorities of subtasks and then by combining subtasks of the same priority into segments. The authors proved that this transformation does not affect the scheduling characteristics of the original dependent tasks. The finish time of a transformed task into its canonical form is the same as the finish time of the original task.

The schedulability analysis proposed in this paper is based on checking the final deadline of the task and identifying the blocking and preemption effects on the canonically-transformed tasks.

Procedure: Let τ_i' be the canonical form of a dependent task τ_i in the task set. τ_i consists of n_i subtasks, each is denoted by $\tau_{i,j}$ where $1 \leq j \leq n_i$.

The schedulability condition starts by calculating the response time of the first subtask $\tau_{i,1}$ in the canonical task τ_i', then iteratively determines the finish time of its successors as a function of its finish time until the final subtask τ_{i,n_i}. This procedure is performed on every job in τ_i''s busy period.

Schedulability Condition: All jobs of task τ_i will meet their deadlines if the following condition is satisfied:

$$max(k-1)T_i + D_i - E_{i.n_i}(k) \geq 0 \tag{2.1}$$

for $k \leq N_i$

- T_i is the period of task τ_i.
- D_i is the deadline of task τ_i.
- $E_{i.n_i}(k)$ is the finish time of the final subtask τ_{i,n_i} of the k^{th} job of task τ_i.
- L_i is the length of busy period of task τ_i of synchronous task set.

- $N_i = \lceil \frac{L_i}{T_i} \rceil$ is the number of jobs of task τ_i in its busy period.

The schedulability condition has been extended in the state-of-the-art to consider more general task models, including recurrent tasks with intra-task parallelism and resource sharing. For instance, Richard et al. [107] analyzed the scheduling problem of graph tasks on uniprocessor systems where each task consists of a set of dependent fixed-priority subtasks with precedence constraints. In order to remove the parallelism within subtasks, the authors provided a graph-to-chain transformation which is described as follows:

Graph-to-Chain Transformation: A graph task is transformed into a chain by placing the eligible subtasks of the graph with the highest priority subtasks at the tail of the chain. A subtask is called eligible if it has no predecessor subtasks or all its predecessors are already in the chain.

As in the transformation from [68], Richard et al. [107] proved that this transformation does not affect the schedulability of the original graph tasks, and the scheduling results of any algorithm is the same for both forms; the original graph task and its sequentially-transformed chain. It is also proved by examples that periodic dependent tasks suffer from scheduling anomalies regarding smaller execution time and removal of a precedence relation between subtasks. In other words, a feasible periodic graph task on uniprocessor may become infeasible if it executes for less than its WCET or if one or more of its intra-task precedence constraints are removed.

Additionally, the authors provided a schedulability condition for fixed-priority scheduling of arbitrary-deadline chains on uniprocessor systems, by following these given steps:

Schedulability Test:

- Transform each task chain into its canonical form (from [68]).
- Determine the blocking and preemption effects in the chain.
- Calculate the finish time (worst-case response time) of the task and the chain during the busy period.

For any subtask $\tau_{i,j}$ of transformed chain τ'_i with priority $P_{i,j}$, the worst-case interference on $\tau_{i,j}$ in a busy period starting at $t = t_0$ is identified by the following scenario:

- Subtask $\tau_{i,j}$ is activated at $t = t_0$.

- A job of each task that starts by a high-priority segment of multiple blocking effect arrives at t_0 (a segment is a sub-chain of τ_i).
- A job of the longest task that starts by a high-priority segment of single blocking effect is activated at t_0.

The schedulability condition is applied on all chains of the task set based on this scenario to calculate the worst-case response time.

In 2005, Zhao et al. [120] generalized the canonical transformation to schedule a set of fixed-priority sporadic graph tasks on uniprocessor systems. Similarly to [107], each graph consists of a set of subtasks which are identified by specific WCET and fixed priority. The authors proposed to transform graph tasks into canonical chains, and to use the chains to establish the worst-case response time of a graph task for a fixed-priority scheduling. The scheduling analysis is studied for different scheduling situations: the preemptive scheduling, non-preemptive scheduling and hybrid tasks in which a single graph have both preemptive and non-preemptive subtasks.

In this work, the authors transformed graphs into sequential canonical chains, then they calculated the finish time of chains by analyzing their blocking and the preemption effects. In [121], the work had been extended to consider Fixed Job Priority (FJP) scheduling of graph tasks when Earliest Deadline First (EDF) algorithm is used. The authors provided schedulability conditions for preemptive and non-preemptive graph scheduling.

Intermediate Timing Parameters

In 2008, Jayachandran and Abdelzaher [72] targeted uniprocessor scheduling in distributed systems where tasks are represented by Directed Acyclic Graphs (DAGs), by applying priority-based resource scheduling and resource partitioning. Their main motivation behind the transformation is to get advantage of the uniprocessor scheduling theories and algorithms which are studied thoroughly. The difference between this work and above-described researches is that the proposed transformation depends only on the load which the analyzed task encounters along its execution path (it is linear to the number of other scheduled tasks). Their main contribution is summarized as follows:

Given a distributed task $\mathcal{A}$ in a distributed task system of workload W_{dist}, it is possible to systematically construct a uniprocessor task $\mathcal{B}$ of a uniprocessor workload W_{uni}, such that if $\mathcal{B}$ is schedulable on uniprocessor system, then $\mathcal{A}$ is schedulable in the distributed system.

This is done by applying a *Delay Composition* theorem (from [71]) which is defined as the maximum time a task waits for its slot on the execution platform. The scheduling analysis provides a worst-case bound on the end-to-end delay of a job under both preemptive and non-preemptive scheduling in the distributed system. Then a transformation is proposed for distributed task to convert them into uniprocessor-compatible form so as to use common traditional scheduling theorems. This transformation is applied to various situations such as static-priority scheduling, dynamic priority scheduling, aperiodic task scheduling and partitioned-resource systems.

At first, Jayachandran and Abdelzaher [72] considered parallel tasks whose subtasks form a path in a directed graph. This work was then extended to tasks whose subtasks form a DAG with internal precedence constraints and each subtask executes on a different resource. In order to apply the Delay Composition theorem on DAGs, tasks have to be split into smaller tasks which form a path in the DAG with artificial deadlines after each merger of subtasks. Assigning these artificial deadlines adds pessimism to the schedulability analysis, but the Delay Composition theorem reduces the need to impose artificial deadlines to only certain stages in the execution interval where two or more subtasks merge. The Delay Composition theorem performs well and obtains better results when compared to other techniques.

A different approach to schedule dependent tasks with precedence constraints has been proposed by Chetto et al. [41]. In 1990, the authors studied the problem of scheduling sets of two kinds of tasks: independent periodic sequential tasks and sporadic groups of dependent graphs with precedence constraints. All task groups execute on uniprocessor systems. They also provided a schedulability test for task sets when preemptive EDF algorithm is used.

Based on their task model, real-time uniprocessor scheduling algorithms can be applied directly to schedule the first type of tasks which belongs to the classical model. Regarding task group, they considered the case of aperiodic dependent tasks that may arrive at any time and execute one time only. Each task in the graph is characterized by an execution time, a release time and a deadline.

For practical reasons, a transformation method was proposed to convert dependent tasks into independent tasks with modified timing parameters. As a result, the same scheduler can be used to schedule both periodic and aperiodic tasks in the set without distinction. The transformation algorithm starts by modifying the release and deadline of dependent tasks, then it constructs a priority list in a way that preserves both timing and precedence constraints.

For a given task in the graph, its deadline is modified based on the deadline of its successors, while its release time is modified based on the finish time of its predecessors. The second part of the algorithm constructs a priority list for tasks based on the modified parameters using EDF scheduling algorithm. The authors proved that the original task group is schedulable if and only if the modified group is schedulable, which means that such transformation is optimal since it respects all timing and precedence constraints of tasks whenever it is possible. The authors provided as well a polynomial online acceptance test to verify whether an occurring task group can be accepted or not.

Recently in 2011, Stigge et al. [113] considered a task model that uses directed graphs to represent the release scenario of jobs in terms of order and timing on uniprocessor systems. Namely, a task is represented by a DAG where each vertex represents one type of jobs that can be released by the task and the edges represent the order in which jobs are released. In this work, the main contribution is to show that the feasibility problem can be decided in pseudo-polynomial time. In [114], the authors studied the computational complexity of the extension to the model which allows to express global inter-release time constraints between non-adjacent job releases.

2.3.2 Parallel Scheduling on Multiprocessor Systems

On multiprocessor systems, preemptive scheduling of jobs with precedence constraints has been proved NP-Hard in the strong sense in 1975 by Ullman [116]. Moreover in 1989, Han and Lin [67] analyzed the effect of job parallelism on the complexity of multiprocessor scheduling of hard real-time systems. They considered a system which consists of a set of independent jobs where each jobs is allowed to execute on multiple processors simultaneously. They proved that parallel fixed priority scheduling on multiprocessor systems is NP-Hard and an exact schedulability analysis of job parallelism is intractable (hard to be controlled or dealt with).

Based on the characteristics of parallel tasks and their internal structure, parallel scheduling is divided into two approaches :

- **Gang Scheduling** [55–57, 60, 74]: It is defined as the scheduling in which all parallel threads are forced to execute simultaneously on multiple processors due to concurrent necessary communications between them. As shown in Figure 2.1, a parallel task is characterized by its WCET and the number of processors that it executes on. Gang

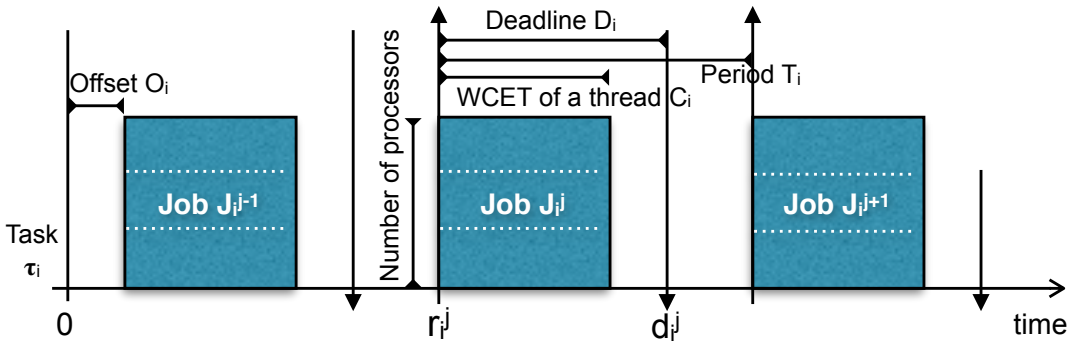


FIGURE 2.1: Periodic constrained-deadline parallel task τ_i of the Gang model.

scheduling is a prominent feature of the *Connection Machine CM-5* [87] which was a series of supercomputers that supported parallelism. The first result regarding real-time Gang scheduling was provided by Kato and Ishikawa [74] who presented an preemptive Gang EDF scheduling algorithm. Later on, Goossens and Berten [60] provided an exact schedulability condition for FTP Gang scheduling on identical processors.

Another parallel approach that is relatively similar to the Gang model is the Work-Limited Parallelism described in [45, 46]. On a system of m identical processors, each sporadic implicit-deadline task τ_i is characterized as (C_i, T_i, Γ_i) where $\Gamma_i = (\gamma_{i,1}, \gamma_{i,2}, ..., \gamma_{i,m})$. As in the classical sequential model, C_i denotes τ_i's WCET, T_i is its period and parameter Γ_i represents job parallelism of τ_i, i.e., its degree of parallelism. A job that executes for t time units on j processors completes $(\gamma_{i,j} * t)$ units of execution. Collette et al. [45] proved that time complexity of task set feasibility is linear and they provided a theoretically optimal scheduling algorithm for such systems.

- **Multi-Threaded Scheduling**: its parallel tasks consist of threads that execute on multiple processors based on the decisions of the real-time scheduler. Unlike the Gang model, parallelism is a choice given to the scheduler and it can execute tasks either sequentially or in parallel. The task model defines the number of threads of each parallel task (or segment) and the scheduler determines which threads execute in parallel and which sequentially. An example of this model is the recurrent tasks[1] which is the base of famous parallel programming libraries such as OpenMP and POSIX Thread (pthread). In the remainder of this section, we will focus on researches and results which concentrate on this scheduling approach and specially the parallel scheduling of recurrent task model such as the Fork-join model and the Multi-Threaded Segment model. Then we will explain in more details the related researches done on Directed Acyclic Graph model.

[1]For more details, please refer to Section 1.3.3 (General Parallel Real-time Task Models).

RT-OpenMP is a real-time concurrent platform presented in [2] based on OpenMP [1]. It includes real-time semantics and scheduling of parallel tasks and it provides a true parallel programming interface. In [2], the platform performance is evaluated by executing parallel task sets under various partitioned scheduling strategies and utilization.

There are two main scheduling approaches resulting from the researches found in the state-of-the-art regarding the problem of scheduling recurrent task model, (i) the Model Transformation approach which is used to simplify the scheduling process by converting the task model and (ii) the Direct Scheduling approach which adapts the scheduling process to take into consideration the special characteristics of parallel tasks and their dependencies.

2.3.2.1 Model Transformation Scheduling Approach

The Multi-Threaded model of parallel tasks (such as the Fork-Join and the Segment model) defines parallel tasks as a sequence of segments which consists of parallel threads. Each segment is activated when its predecessor segments terminate their execution. Hence, the task model does not require individual offsets or deadlines for parallel segments, which complicates the scheduling process. Many researches from the state-of-the-art propose to schedule parallel tasks by modifying them in a way that gets rid of the internal execution dependencies and converts it into a set of independent sequential tasks with individual offsets and deadlines. Then, scheduling decisions and analyses are performed using intermediate offsets and deadlines which are assigned by the transformation approach and used by real-time algorithms to be scheduled on multiprocessor systems.

The Model Transformation approach simplifies the problem of parallel scheduling at the expense of incurring some non-trivial transformation overheads. Also, to the best of our knowledge, there is no optimal method which assigns intermediate timing parameters to subtasks and guarantees the feasibility of the task set.

Stretching Algorithm of Fork-Join Model

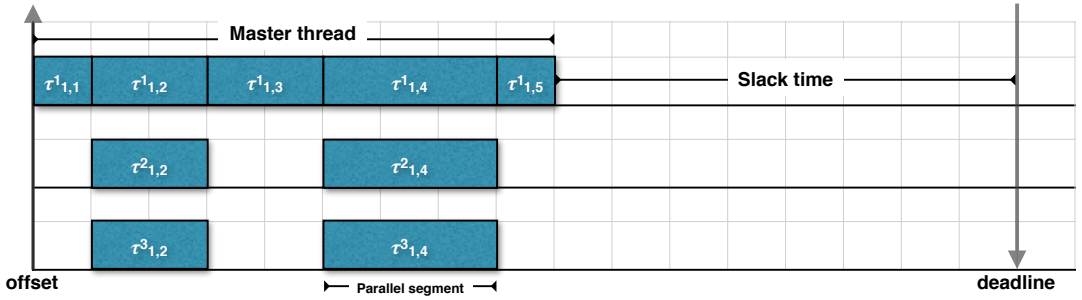
Lakshmanan et al. [77] introduced the Fork-Join (FJ) model of parallel tasks in real-time systems, in which they studied partitioned scheduling of periodic implicit-deadline tasks on homogeneous multiprocessor systems. Figure 2.2(a) shows an example of a FJ task τ_1 which consists of 5 segments and a master thread of length equal to 9. The master thread of any FJ task is

defined as the thread with the longest sequential execution time. The FJ task begins executing the master thread and when a fork event happens, the master thread splits into a number of identical threads that execute in parallel which forms a parallel segment. When all of the threads of a parallel segment terminate their execution, they join to resume the execution of the master thread. The slack time of a FJ task is defined as the timing difference between the deadline of the task and the WCET of its master thread.

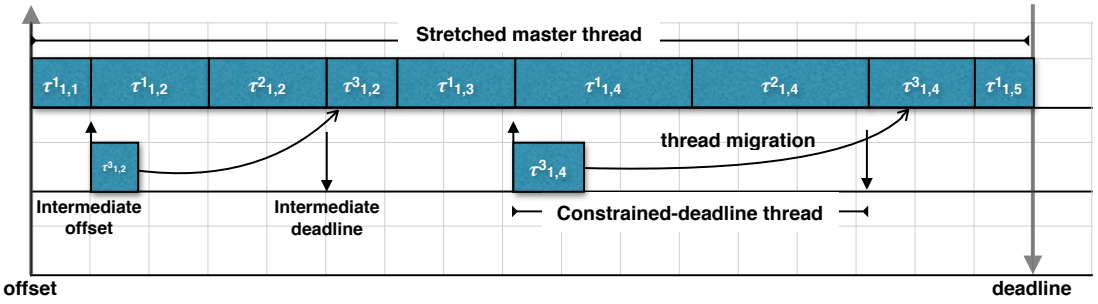
After identifying the best-case and the worst-case structure of FJ task sets from feasibility perspective, the authors showed that the parallel structure of FJ tasks is undesirable and should be avoided. As mentioned in their work, "FJ task sets on multiprocessor systems can have schedulable utilization bounds slightly greater than and arbitrarily close to uniprocessor schedulable utilization bounds. From the perspective of schedulability, it is therefore desirable to avoid such task structures as much as possible ...".

As a result, Lakshmanan et al. provided a stretching algorithm for the FJ tasks so as to execute them as sequential as possible. The algorithm is divided into two main cases:

- Low utilization FJ tasks whose utilization is not greater than 1 are forced to execute sequentially, since their total WCET is less than the deadline. The threads of each task are ordered to form a sequential chain which executes on a single processor.
- High utilization FJ tasks whose utilization is greater than 1 must execute in parallel and one processor is not enough for them to be feasible. Hence, the stretching algorithm is applied as follows:
  - The stretching algorithm aims to fairly fill the slack of the FJ task by fractions of threads from the parallel segments of the task, based on a distribution factor. The distribution factor is calculated based on the relation between the slack time of the task and the minimum sequential execution time of all of its parallel segments.
  - The master thread is stretched up to its deadline which extends the parallel segments. The remaining parallel threads are then forced to execute within fixed execution intervals.
  - The result of applying the stretching algorithm is a fully-stretched master thread (with utilization equal to 1) and a set of constrained-deadline tasks with fixed offsets and deadlines. The fully-stretched master threads are assigned their own processors by the real-time scheduler, while constrained-deadline threads are scheduled using classical multiprocessor algorithms.



(a) A Fork-Join task τ_1.



(b) The result of the stretching algorithm applied on τ_1.

FIGURE 2.2: An example of stretching algorithm of Fork-Join task model.

Figure 2.2(b) shows an example of the stretching algorithm applied on the FJ task from Figure 2.2(a).

Lakshmanan et al. proposed to use partitioned preemptive Deadline Monotonic algorithm to schedule the constrained-deadline parallel threads resulting from the stretching. They proved that stretched task sets have a resource augmentation bound equal to 3.42 when *FBB-FFD* with Deadline Monotonic is used. This implies that any FJ task set that is feasible on m unit-speed processors can be scheduled after stretching on m processors that are 3.42 times faster when scheduled using FBB-FFD[2] [59] with DM.

The main disadvantage of the FJ stretching algorithm lies in the system overheads caused by the stretching. The algorithm forces a thread from each segment of the stretched task to execute on two processors. We proposed in [54] a modified version of the stretching algorithm which is called the *Segment Stretching algorithm*. It aims at reducing thread migrations and preemptions forced by the stretching algorithm and we proved that it has the same resource augmentation bound of the original stretching algorithm.

The stretching algorithm is an interesting approach of scheduling FJ tasks. In Chapter 3, we extend this algorithm to consider the Directed Acyclic Graph model which is more general than

[2]FBB-FFD stands for **F**isher **B**aruah **B**aker-**F**irst **F**it **D**ecreasing algorithm.

the FJ. Also, we proposed a modified algorithm which reduces the number of thread migrations caused by the stretching algorithm.

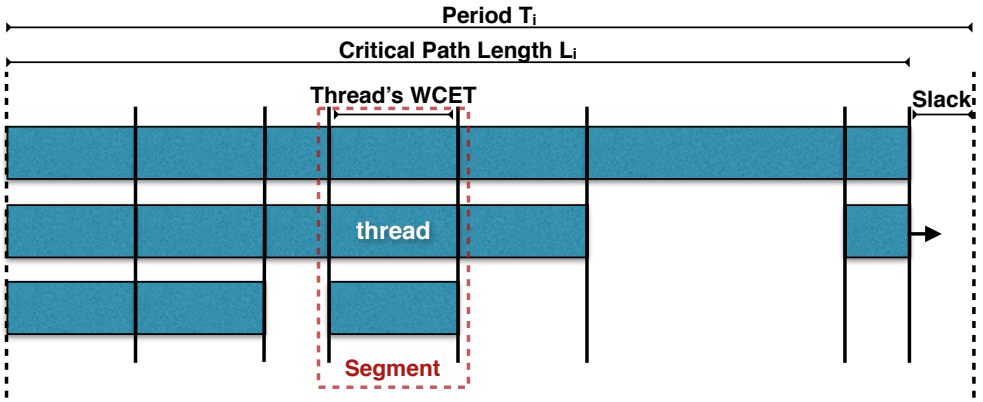
Decomposition Algorithm for Multi-Threaded Segment Model

Saifullah et al. [108] studied the scheduling of synchronous implicit-deadline parallel tasks of the Multi-Threaded Segment (MTS) model[3] (which was extended later to DAG tasks in [109]). They simplified the scheduling problem of parallel tasks by using a *Decomposition* algorithm which assigns intermediate offsets and deadlines for each segment. These intermediate parameters are used by real-time algorithms to schedule tasks after they are transformed into a set of independent sequential subtasks. The schedulability analysis of decomposed tasks is based on their densities, and segments are assigned intermediate deadlines which guarantee that the density of any segment in the decomposed task τ_i is not greater than $\frac{2C_i}{T_i}$. Briefly, the Decomposition algorithm is summarized as follows:

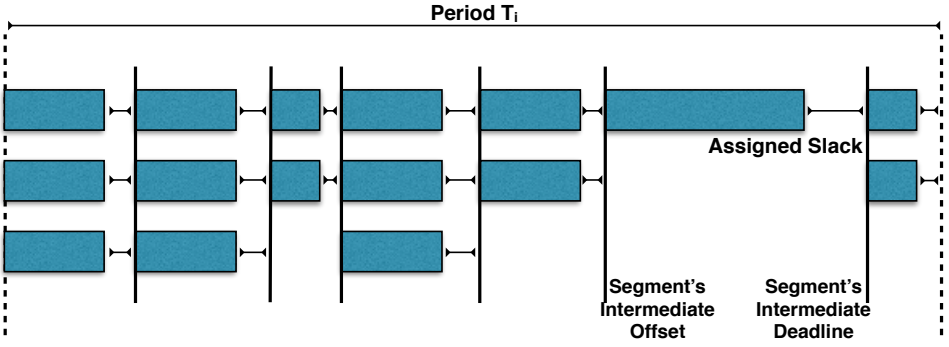
- The parallel segments of a MTS task τ_i are divided into groups based on the number of their threads w.r.t.a threshold $\theta_i = \frac{C_i}{2T_i - L_i}$: heavy segments and light segments.
- The task's deadline is split into intermediate deadlines for all segments based on their groups; whether they are all light segments, heavy segments or a mix of light and heavy segments. As a result, each thread of each segment has enough slack time to execute.
  - If all segments are light, then task deadline is split proportionally among all segments according to the WCET of each thread.
  - If all segments are heavy, then task deadline is split proportionally among all segments based on their workload (the total execution time of each segment).
  - If a task contains some heavy and light segments, then the period T_i of the task is divided into two parts. A total of $(T_i - L_i/2)$ is split proportionally among heavy segments according to the work of each segment, and the remaining $L_i/2$ time units are assigned to the light segments.

Figure 2.3 shows an example of the Decomposition algorithm applied to a MTS task which is shown in Figure 2.3(a).

[3] For more details about the model and notations, please refer to Section 1.3.4.



(a) A Multi-Threaded Segment task.



(b) The result of the Decomposition algorithm applied on the task.

FIGURE 2.3: An example of Decomposition algorithm of Multi-Threaded Segment task model (from [109]).

Finally, the authors proved that the scheduling of the decomposed constrained-deadline tasks has a resource augmentation bound equal to 4 in the case of preemptive GEDF scheduling algorithm, and 4 plus a constant non-preemptive overhead for non-preemptive GEDF scheduling.

Optimizing the Number of Processors for Multi-Threaded Segment Scheduling

Following the same reasoning of the Decomposition algorithm, Nelissen et al. [93] modified the parallel MTS model by proposing an online optimization algorithm that assigns intermediate offsets and deadlines for segments. Unlike the traditional MTS model, the authors considered a task model which consists of a sequence of parallel segments, and each segment consists of a number of different threads with specific WCETs. The optimization algorithm is proved to be optimal when the schedulability test is based on task densities. The intermediate deadlines are assigned to segments in a way that minimizes the number of processors needed for scheduling, by imposing the average density of the task to the maximum number of segments. As a result,

minimizing the density of segments leads to minimizing the number of processors needed to schedule the task.

The authors provided some generalizations to the task model by applying the optimization algorithm on the DAG model. However, their solution has an exponential complexity and requires to test all possible segments of a graph in order to compute the optimized segment densities. In [103], we proposed a Graph-to-Segment transformation which solves this problem and provides a compatible transformation of DAGs to MTS model which maintains the optimality of the optimization algorithm and reduces the complexity.

2.3.2.2 Direct Scheduling Approach

The used of indirect scheduling approaches through parallel model transformation requires assigning intermediate timing parameters to subtasks/threads of parallel tasks, which causes additional overhead and a substantial amount of pessimism. In this approach, reducing the execution interval of a parallel task by dividing it into smaller intervals of threads seems to severely limit the flexibility of the scheduling. A thread after transformation is forced to execute within an execution interval limited by intermediate offset and deadline, while in reality, the thread can execute outside this interval. Such problem affects the schedulability of parallel tasks.

A Direct scheduling approach [8, 119] is used in the state-of-the-art to schedule parallel tasks without transformation in order to adapt scheduling algorithms and analyses to take into consideration dependencies and precedence constraints of tasks. In the remainder of this subsection, we present the principle researches found in the state-of-the-art regarding the Direct scheduling approach and we highlight their main contributions.

Worst-Case Response Time of Fork-Join Model

Axer et al. [10] provided a worst-case response time of FJ tasks with arbitrary deadlines when partitioned preemptive fixed priority scheduling is used. The authors considered a task set that consists of both traditional independent tasks and parallel FJ tasks that execute on multiprocessor systems. The worst-case response time of both tasks (independent and FJ) is calculated based on the busy-period approach. The definition of busy-period of traditional independent tasks (from Definition 1.1 on page 4) is modified to the case of FJ tasks as follows:

Definition 2.2 (Busy period of a FJ task [10])**.** A busy period of a FJ task is a time interval in which all response times of the FJ task depend on the execution of at least one previous activation in the same busy period, except for the very first activation of the task.

Global EDF Scheduling Analysis for Multi-Threaded Segment Model

In 2012 Andersson and Niz [6] studied the global EDF scheduling of sporadic constrained-deadline tasks for the Multi-Threaded Segment model on multiprocessor systems. Global EDF assigns priorities to jobs based on their absolute deadlines and all segments (including their threads) inherit the same priority as their job. Hence, the scheduling does not depend on the common approach of assigning intermediate offsets and deadlines of segments. The schedulability analysis which led to EDF condition is based on system demand technique from [26, 34] (Demand Bound Function and its refinements: Maximum and Forced-Forward Demand Bound).

Interference-Based Analysis for Multi-Threaded Segment Model

The concept of interference and problem window of the classical task model was introduced in [11, 30] and interference bounding techniques in [17, 29]. They expressed the relation between a given task and the interference from higher priority jobs that may block its execution. Chwa et al. [44] extended the notion of interference to capture thread-level parallelism more accurately, by considering the problem of global preemptive scheduling of sporadic synchronous MTS tasks on a platform of m identical processors. As described earlier, the parallel model defines a task as a sequence of segments with one or more synchronous threads.

Chwa et al. provided interference definitions and the notion of parallelism-awareness to take into consideration the precedence constraints within segments of the tasks.

Definition 2.3 (Critical thread)**.** A thread is said to be critical if it finishes last among all the threads belonging to the same segment.

Based on this definition, the interference definition for a task can be modified to consider the worst-case scenario in which the parallel task interferes on the critical thread.

Definition 2.4 (Critical Interference)**.** Critical interference $\mathcal{I}_k(a, b)$ is the sum of all intervals in which a critical thread from task τ_k is ready for execution but cannot execute due to other higher-priority threads in the time interval $[a, b)$.

Due to the structure of parallel MTS tasks, the source of interference on a given thread is hard to be identified. Hence, Chwa et al. presented a *p-depth critical* interference which represents the behavior of parallel execution and allows to figure out exactly how many interfering threads are executing simultaneously when τ_i delays the execution of another task τ_k.

Definition 2.5 (p-depth critical interference)**.** p-depth critical interference $\mathcal{I}_{i,k}(p, a, b)$ of task τ_i on task τ_k during interval $[a, b)$ is the cumulative length of all intervals in which (i) a critical thread of τ_k is ready to execute but does not, and (ii) exactly p threads of τ_i are executing.

These interference definitions, which are more adapted to intra-task parallelism of the task model, are used to derive efficient global EDF schedulability conditions that are directly applicable to synchronous parallel task models on multiprocessor systems.

New Multi-Thread Task Model

In 2011, Lupu and Goossens [86] presented a new model of the recurrent task model in which a task consists of a sequence of independent threads. These threads share the same deadline and each thread is a sequential process that requires a single processor. In this work, the authors define two different classes of real-time schedulers for this model which are summarized as follows:

- Hierarchical schedulers manage tasks with a task-level rule and use a second rule to schedule threads within each task (thread-level rule).
- Global thread schedulers use a single scheduling rule to assign priorities to threads regardless of their tasks.

An exact schedulability condition is provided for each class of schedulers. The performance of these schedulers is also compared with Gang scheduling.

Directed Acyclic Graph Model

In this subsection, we concentrate on the general model of parallel tasks which is the Directed Acyclic Graphs (DAGs), and we present related work regarding multiprocessor scheduling problem. The DAG model is more general than the FJ and MTS tasks which are represented as a sequence of segments. The precedence constraints and dependencies between subtasks of DAG

task define the execution flow of the DAG and complicates the scheduling process and analysis of such tasks.

Baruah et al. [27] in 2012 introduced the problem of scheduling DAG tasks on multiprocessor systems. The considered model is the sporadic arbitrary-deadline DAG which consists of a set of dependent vertexes (subtasks or jobs as referred to in the original paper). All vertexes of a DAG are released simultaneously and have to execute within a specified relative deadline w.r.t. their release. The authors started by proving that the synchronous arrival sequence, in which successive jobs are released periodically, is not the worst-case behavior of the sporadic DAG task.

Since the scheduling problem of the recurrent task model is computationally intractable, efficient approximations can be provided as solutions. The authors analyzed EDF scheduling algorithm and found that it has a resource augmentation bound of at most 2. The authors provided sufficient polynomial and pseudo-polynomial EDF schedulability conditions which determine whether a given task set is schedulable by EDF on a platform of m identical processor. The polynomial EDF condition is proved to have a resource augmentation bound equal to $3 - \frac{1}{m}$. It is worth mentioning here that the schedulability analysis provided in this paper used two global timing characteristics of each DAG task which are its critical path length and its total WCET. The internal structure of DAGs and the exact execution flow of their subtasks are not considered in the analysis.

GEDF Multiprocessor Scheduling of DAG sets (from [81])

Subsequently, Li et al. [81] studied global preemptive scheduling of a task set of sporadic implicit-deadline DAGs on platform of m identical processors. This is an extension of the problem presented in [27], which considered the scheduling of a single arbitrary-deadline DAG task. The authors analyzed global EDF scheduling of such task sets and proved a resource augmentation bound equal to $2-(1/m)$ for arbitrary-deadline DAGs. Moreover, the authors provided a capacity augmentation bound for global EDF equal to $4-(2/m)$ which can serve as a schedulability condition. It states that if a DAG set has a total utilization at most $m/(4-(2/m))$ and the critical path length of each DAG is not greater than $1/(4-(2/m))$ its deadline, then it can be scheduled on a machine with m processors under GEDF.

A sufficient global EDF schedulability condition is based on the following lemma which was inspired from an observation given in [27]:

Lemma 2.6. *(from [81]) If the total workload A_k^a on the a^{th} job of DAG task τ_k is bounded by*

$$A_k^a \leq bmD_k - (m-1)D_k$$

then this job can meet its deadline on m identical processors with speed of b.

On a system of m processors of speed b, the proof of Lemma 2.6 is based on two straightforward observations regarding the schedulability of DAGs:

- At each incomplete sub-step[4] (at least one processor is idle), the remaining critical path length for each unfinished job is decreased by 1.
- The total work F^t done in an interval of length corresponding to t steps on m processors of speed b, in which there are t^* incomplete sub-steps is defined by the relation:

$$\begin{aligned} F^t &\geq m(bt - t^*) + t^* \\ &\geq bmt - (m-1)t^* \end{aligned}$$

Furthermore, the total workload A_k^a on the a^{th} job of DAG τ_k ($J^{k,a}$) is calculated as follows:

$$A_k^a = R^{k,a} + \sum_i u_i n_i^{k,a} D_i$$

where:

- $n_i^{k,a}$ is defined as the number of jobs of task τ_i which are released after the release of job $J^{k,a}$ but have deadlines no later than the deadline of $J^{k,a}$.
- $R^{k,a}$ is the total carry-in (jobs which are released before $J^{k,a}$ but have deadlines later than $J^{k,a}$'s deadline) work from every task τ_i.

$$R^{k,a} \leq \sum_i u_i \alpha_i^{k,a} + m.\max_i(\alpha_i^{k,a})$$

where $\alpha_i^{k,a}$ is the number of steps between the absolute release time of $J^{k,a}$ and the absolute deadline of the carry-in job of task τ_i.

[4] Each time unit (step) is divided into sub-steps based on the speed of processors.

GEDF Multiprocessor Scheduling of DAG sets (from [36])

Bonifaci et al. [36] studied the same scheduling problem as in [81]. In this work, the authors considered a sporadic DAG set on a platform of identical processors. Similarly, this work is an extension to the research done in [27] and it considers the case of scheduling multiple DAGs. The authors studied DAG scheduling in case of global preemptive EDF and DM scheduling algorithms. They proved that global EDF has a resource augmentation bound equal to $2-(1/m)$ and $3-(1/m)$ in the case of global DM. These bounds correspond to the ones obtained for the traditional model of sporadic independent sequential tasks which was proved in [26, 35, 95] for GEDF and [26] for GDM.

Moreover, The authors provided schedulability conditions with pseudo-polynomial time complexity accompanied with simple polynomial time sufficient conditions to test global EDF and DM schedulability.

Based on DAG schedulability analyses from [27, 36, 81], the internal structure of DAGs is not considered. These researches use the critical path length and the total WCET of DAGs in the analysis to represent a DAG task. However, we believe that DAG schedulability analysis can be improved and more adapted to DAGs if more timing parameters about subtasks and knowledge about their execution flow are included. In Chapter 4, we analyze this assumption and we provide Direct scheduling analyses for DAGs when the scheduling is aware of the internal structure of DAGs.

Chapter 3

Scheduling of Parallel Tasks using Model Transformation

In this chapter, we discuss the scheduling of parallel real-time tasks on homogeneous multiprocessor systems using the Model Transformation approach. Generally, the scheduling process of parallel tasks is not trivial and it is more complicated than multiprocessor scheduling of independent sequential tasks because of internal dependencies and precedence constraints of parallel tasks. The model transformation approach is a pre-step to the parallel scheduling process, and it consists of converting a parallel task into a collection of independent sequential threads. The objective of the Model Transformation is to facilitate the scheduling process of parallel tasks by using common sequential scheduling algorithm for multiprocessor systems and by applying their scheduling conditions and bounds directly to parallel tasks. Furthermore, the Model Transformation approach can be used for other purposes such as reducing overhead costs of the system or reducing the number of processors required for the scheduling process which leads to energy consumption reduction in the system.

This chapter is organized as follows. In Section 3.1, we provide a detailed introduction regarding the Model Transformation approach. In Section 3.2, we remind the reader of our task model which is the Directed Acyclic Graphs (DAGs). We provide in Section 3.3 a stretching algorithm for DAGs denoted by DAG-Str (DAG-Stretching) algorithm. We analyze the schedulability of the DAG-Str algorithm with global preemptive scheduling algorithms for periodic implicit-deadline DAG tasks on homogeneous multiprocessor systems. Furthermore, we provide a resource augmentation bound analysis for global Earliest Deadline First (GEDF) scheduling algorithm equal to $\frac{3+\sqrt{5}}{2}$ for any stretched task set if $n < \varphi.\overline{m}$, where φ is the golden ratio, n

is the number of DAGs in the set and $\overline{m}$ is the number of remaining processors in the system after applying the stretching algorithm.

In Section 3.4, we propose a modified version of the DAG-Str algorithm which is denoted by the Seg-Str (Segment Stretching) algorithm. This algorithm is based on the Model Transformation approach and it aims at reducing the number of job migrations and preemptions resulting from applying the DAG-Str algorithm. As in the case of the previous algorithm, we analyze the performance of Seg-Str algorithm by proving that it has the same resource augmentation bound as for the DAG-Str algorithm. Finally in Section 3.5, we support these performance analyses by providing experimental simulation results, in which we compare the schedulability of DAG-Str algorithm with another algorithm from the state-of-the-art which belongs to the Model Transformation approach.

3.1 Introduction and Motivation

Based on the basic definition of the real-time task model, an independent periodic constrained-deadline sequential task generates an infinite number of jobs. Each job is characterized by an absolute offset, a WCET, an absolute deadline and a period. By using these timing parameters and characteristics, a real-time scheduler can predict the demand of scheduled tasks and the execution flow of each of their jobs at all times, which enables it to take adapted scheduling decisions regarding the scheduled task set.

In the case of the parallel DAG model, a DAG task consists of a set of subtasks with precedence constraints which define the execution order of these subtasks. When a DAG job is activated, only its source subtasks are activated while the rest of its subtasks are in the ready state waiting to be activated. A ready subtask is activated when all of its predecessor subtasks complete their execution. Hence, the activation time of subtask jobs is dynamic based on the scheduling decisions of the scheduler regarding their predecessors. As a result, the scheduling of DAG tasks on multiprocessor systems is more difficult compared to the scheduling of independent sequential tasks. At any time instant in the execution interval of a DAG set, the scheduler does not have all necessary information regarding the scheduling process nor the execution flow of the subtasks, and it is only aware of the currently active subtasks of DAG tasks at a given time instant, and it has no information about how the successor subtasks will be executed.

Another reason causing the DAG scheduling difficulty is that subtasks are only characterized by their WCET and they inherit the remaining timing parameters from their DAG, such as offset, deadline and period. For example, a subtasks has no intermediate deadline. A DAG deadline miss occurs whenever a subtask misses the deadline of the DAG. However, the global deadline miss of the DAG may be avoided by the scheduler if the subtask has its own intermediate deadline which is earlier than the DAG's deadline.

In the remainder of this section, we propose a DAG stretching algorithm based on the model transformation approach which is mainly used to simplify the DAG scheduling problem. Instead of complicating the scheduling process with dependencies and structure-aware decisions, the parallel task model is converted into the common independent sequential real-time task model. According to this, parallel tasks are transformed into a collection of independent sequential threads. The independence between threads is ensured by assigning them intermediate offsets and deadlines on which the scheduling decisions are taken. As a result, the scheduling process is simplified and it is considered as the common scheduling problem of independent threads on multiprocessor systems which has been well studied. Common scheduling algorithms can be used and their performance analyses can be directly applied to transformed threads.

The idea behind the transformation technique is to define extra timing parameters for subtasks of parallel tasks to force them to execute independently. A parallel task is transformed into a sequence of parallel segments characterized by an intermediate offset and an intermediate deadline. Each segment contains a set of threads which are fragments of subtasks of the parallel task. Subtasks are activated based on their intermediate offset and not at the end of their predecessor subtasks. Threads of the transformed parallel task execute based on intermediate timing parameters defined by the transformation. It is worth mentioning here that the Model Transformation approach is a pre-step to the scheduling process which modifies the timing characteristics of the task set, and can be associated to any scheduling algorithm for multiprocessor systems.

The model transformation can be considered as a trivial approach. However, it suffers from a basic disadvantage which is the loss of model generality due to the modification of the parallel task model. As described above, the Model Transformation approach consists of adding more timing parameters to the subtasks of DAG tasks. As a result, the general parallel DAG task model is transformed into a less general sequential independent model, with extra restricted local timing parameters.

In this chapter, we start by proposing the DAG-Str algorithm which is based on the Model Transformation approach to schedule periodic implicit-deadline DAG tasks on multiprocessor systems. This transformation is an extension of the task stretching (Task-Str) algorithm proposed by Lakshmanan et al. [77] (from Subsection 2.3.2) for the Fork-Join parallel task model (from Subsection 1.3.4.1). The concept of DAG-Str algorithm is to execute parallel tasks as sequentially as possible so as to avoid parallel executions of tasks. Briefly, the stretched parallel task is converted into a fully-stretched master thread with utilization equal to 1 and a collection of independent threads with intermediate offsets and deadlines, which will be explained in more details in the remainder of this chapter.

Then, we analyze the performance of DAG-Str algorithm when global preemptive Earliest Deadline First (GEDF) scheduling algorithm is used. We provide a resource augmentation bound for this scheduling algorithm so as to evaluate the performance of the DAG-Str algorithm.

3.2 Task Model and Notation

In this chapter, we consider the Directed Acyclic Graph (DAG) model of parallel tasks described in Subsection 1.3.4. More precisely, we consider a task set τ of n periodic DAGs that run on a system of m identical unit-speed processors. The task set τ is represented by $\{\tau_1, ..., \tau_n\}$. Each DAG task τ_i, where $1 \leq i \leq n$, is a periodic implicit-deadline graph which consists of a set of subtasks under precedence constraints that determine their execution flow. A DAG task τ_i is characterized by $(n_i, \{1 \leq j \leq n_i | \tau_{i,j}\}, G_i, D_i)$, where n_i is the number of DAG's subtasks, the second parameter represents the set of subtasks of τ_i, G_i is the set of directed relations between these subtasks and D_i is the relative deadline of τ_i. Since each DAG task has an implicit deadline, its period T_i is the same as its deadline ($T_i = D_i$).

Let $\tau_{i,j}$ denote the j^{th} subtask of the set of subtasks forming the DAG task τ_i, where $1 \leq j \leq n_i$. Each subtask $\tau_{i,j}$ is a single-threaded sequential task which is characterized by a WCET $C_{i,j}$. All subtasks share the same absolute deadline and period of their respective DAG. The total WCET C_i of DAG τ_i is defined as the sum of WCETs of its subtasks, i.e., $C_i = \sum_{j=1}^{n_i} C_{i,j}$. Let U_i denote the utilization of τ_i where $U_i = \frac{C_i}{T_i}$, and $\delta_i$denotes its density with $\delta_i = \frac{C_i}{min(D_i, T_i)}$. For an implicit-deadline task, whose deadline is equal to its period, the density of the task is the same as its utilization, while the density of a constrained-deadline task is $\delta_i = \frac{C_i}{D_i}$.

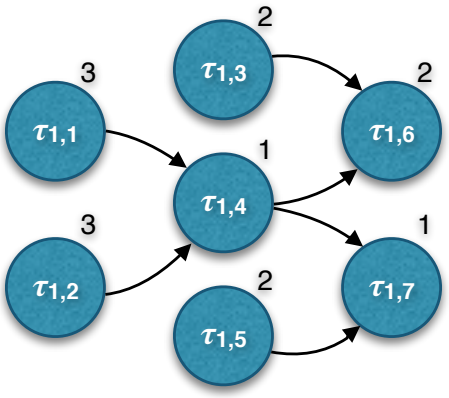


FIGURE 3.1: An example of a DAG task τ_1 which consists of 7 subtasks. The number on the upper right corner of each subtask represents its WCET and the arrows represent their precedence constraints.

Based on the structure of DAG tasks, let τ_i^{master} denote the critical path (or the master thread) of DAG τ_i which is defined as the longest sequential execution path in the DAG when it executes on a virtual platform composed of an infinite number of processors. Its length L_i is the minimum response time of the DAG, which means that it needs at least L_i time units to finish its execution in the best case. A subtask that is part of the critical path is referred to as a critical subtask, while non-critical subtasks are the ones executing in parallel with the critical ones.

Based on the execution flow of a DAG when it executes on infinite number of processors, it can be described as a structure of multiple execution paths. Each path defines the sequential execution flow of certain subtasks from a source to a sink. These paths intersect each others and the non-critical subtasks from each path execute in parallel with the master thread τ_i^{master}. For example, Figure 3.2 (on page 59) shows the execution diagram of DAG τ_1 (from Figure 3.1) when it executes on an infinite number of processors. In this example, τ_1 consists of 7 subtasks and 6 execution paths: $\{\{\tau_{1,1}, \tau_{1,4}, \tau_{1,6}\}\ \{\tau_{1,1}, \tau_{1,4}, \tau_{1,7}\}\ \{\tau_{1,2}, \tau_{1,4}, \tau_{1,6}\}\ \{\tau_{1,2}, \tau_{1,4}, \tau_{1,7}\}\ \{\tau_{1,3}, \tau_{1,6}\}\ \{\tau_{1,5}, \tau_{1,7}\}\}$. The master thread of τ_1 is either $\{\tau_{1,1}, \tau_{1,4}, \tau_{1,6}\}$ or $\{\tau_{1,2}, \tau_{1,4}, \tau_{1,6}\}$ with a length $L_1 = 6$. Since both paths are identical w.r.t. their length, we consider the former to be the master thread of the DAG arbitrarily. Thus, DAG task τ_1 needs at least 6 time units in order to execute on unit-speed processors. Let Sl_i denote the positive slack time available to DAG τ_i when it is scheduled exclusively on an infinite number of processors without interference from other tasks. Sl_i is given by:

$$Sl_i = D_i - L_i \tag{3.1}$$

The slack time of the master thread of a DAG is the same as the DAG's slack, since it is the DAG's longest path. Back to the previous example, if we assume that τ_1 has a deadline $D_1 = 10$,

then its slack time is equal to $Sl_1 = 4$.

A DAG task is said to be feasible if subtasks of all of its jobs respect its deadline D_i. For any given DAG task τ_i, there are two trivial necessary conditions regarding its scheduling. A task set τ is deemed unfeasible when scheduled using any scheduling algorithm on m unit-speed processors if, at least, one of the following conditions is false:

- $U(\tau) \leq m$
- $\forall \tau_i \in \tau, \quad L_i \leq D_i$

3.3 DAG Stretching (DAG-Str) Algorithm

Based on our DAG model, subtasks are not assigned their own timing parameters by the model and they inherit the global parameters of their respective DAGs. During the scheduling process, subtasks of a DAG task execute within the execution interval of the DAG (between the release time and the absolute deadline of each DAG job), while the exact activation time of each subtask is unknown prior to the scheduling process. However, the DAG scheduling can be simplified by avoiding the internal parallelism of DAG tasks and assigning their subtasks intermediate offsets and deadlines. Hereby, we propose a stretching algorithm for parallel DAG tasks based on the model transformation approach.

The DAG stretching (DAG-Str) algorithm aims at converting parallel tasks into sequential threads and getting rid of the parallel structure of the DAGs and the dependencies between their subtasks. The parallel tasks are forced to execute as sequentially as possible. As for the model transformation approach, the DAG-Str algorithm is a pre-step of the scheduling process which is used to transform each DAG into a set of independent constrained-deadline threads with at most one implicit-deadline fully-stretched master thread (its utilization is equal to 100%). Each independent thread is assigned intermediate offset and deadline to ensure its execution independence. After the stretching transformation, the scheduling of a DAG task is done based on the intermediate timing parameters of threads using any real-time scheduling algorithm, and a task is deemed feasible if all of its jobs respect their assigned intermediate deadline. Before explaining the concept of our algorithm, we start by analyzing the DAG model and by identifying its characteristics that led to the stretching algorithm. As stated earlier, the DAG-Str algorithm is an extended version of a similar stretching algorithm for a special-case

of parallel tasks for the Fork-Join model that we presented in detail in Subsection 2.3.2.1 on page 39.

DAG Stretching Algorithm (DAG-Str)

For a periodic implicit-deadline DAG task τ_i, the DAG-Str algorithm stretches the master thread τ_i^{master} of a DAG task to its deadline by filling the slack time of the DAG by fragments of non-critical subtasks. As a result, each remaining non-critical subtask (if any) is divided into one or more independent threads and they are forced to execute in parallel with the stretched master thread within a fixed activation interval (between their intermediate offset and their intermediate absolute deadline).

For each DAG task $\tau_i \in \tau$, if τ_i has a utilization less than or equal to 1 ($C_i \leq D_i$), then it fits completely on a single processor and the stretching algorithm transforms it into a single master thread which contains all of its subtasks and forces them to execute sequentially in an order that maintains their execution dependencies. Otherwise, if DAG τ_i has a utilization greater than 1 ($C_i > D_i$), then the algorithm fully stretches the master thread of τ_i up to its deadline D_i. Besides, the stretching algorithm generates a collection of independent constrained-deadline sequential threads with intermediate offsets and deadlines which execute in parallel with the fully-stretched master thread. The intermediate offsets and deadlines are important for the scheduling process and also to maintain the precedence constraints of DAG τ_i. Furthermore, the generated fully-stretched master threads have utilization equal to 1, so, it is only logical to assign them dedicated processors. The independent sequential threads are scheduled on the remaining processors of the system using any partitioned or global multiprocessor scheduling algorithm. In this work, we consider global preemptive EDF scheduling algorithm.

For clarity reasons, the reader is advised to refer to the example in Figures 3.2 and 3.3 in order to have a better understanding of our DAG stretching algorithm. More details about the example are provided at the end of this section.

3.3.1 The Multi-Threaded Segment (MTS) Representation

In order to perform our stretching algorithm and to apply it on DAG tasks, we propose to transform every DAG in the task set into a Multi-Threaded Segment (MTS) representation. This model transformation of DAGs considers the execution order of subtasks when they execute on unlimited number of processors according to their precedence constraints. As described

previously in Subsection 1.3.4.2 on page 22, a MTS task $\overline{\tau}_i$ consists of a sequence of parallel segments. Each segment consists of a number of parallel threads having the same WCET. Let S_i denote the set of parallel segments of $\overline{\tau}_i$, and s_i is its total number of segments. Each segment $S_{i,j}$, $1 \leq j \leq s_i$, consists of $m_{i,j}$ threads, and $\overline{\tau}^k_{i,j}$ denotes the k^{th} thread of segment $S_{i,j}$. Threads which belong to the same segment are characterized by the same WCET value denoted by $c_{i,j}$. The number of threads and their WCETs vary from one segment to another, and each segment contains at least one thread.

In this work, MTS task $\overline{\tau}_i$ is not a new task but a representation form of the existing DAG task $\tau_i \in \tau$. This is done by considering that all subtasks of τ_i are executed as soon as possible on a virtual platform of unlimited number of processors. As a result, each subtask executes under precedence constraints and does not suffer from any interference from other subtasks in the system. For a DAG task τ_i, its source subtasks are activated at the activation time of DAG τ_i. Then, each subtask is activated as soon as its predecessors have completed their own execution. A segment in the MTS task is defined whenever a subtask finishes its execution. Thus, threads from the same segment have the same WCET. It is worth noticing that a subtask may be divided into more than one thread which are executing in successive segments from its MTS representation $\overline{\tau}_i$. Moreover, since the critical path of DAG τ_i is the path with the longest sequential WCET, then the relation between the critical path length L_i and segments S_i of $\overline{\tau_i}$ is given by $\sum_{\forall S_{i,j} \in S_i} c_{i,j} = L_i$. The critical subtasks execute sequentially within the segments of the task.

Example 3.1.

Figure 3.2 shows the MTS representation $\overline{\tau}_1$ of DAG τ_1 from Figure 3.1. Let DAG τ_1 be released at time $t = 0$ and have a deadline $D_1 = 10$. Then, its source subtasks $\{\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,5}\}$ are activated at $t = 0$ as well. At $t = 2$, both subtasks $\tau_{1,3}$ and $\tau_{1,5}$ complete their execution and the first segment $S_{1,1}$ is defined. Their successors (subtasks $\tau_{1,6}$ and $\tau_{1,7}$) have to wait for their predecessor subtask $\tau_{1,4}$ to finish its execution before they can start their own. At time $t = 3$, subtasks $\tau_{1,1}$ and $\tau_{1,2}$ finish their execution and subtask $\tau_{1,4}$ starts its own. Segment $S_{1,2}$ is then defined. Finally, at time $t = 4$, subtasks $\tau_{1,6}$ and $\tau_{1,7}$ are activated, and segments $S_{1,3}$, $S_{1,4}$ and $S_{1,5}$ are defined at times 4, 5 and 6 respectively. The resulting MTS task $\overline{\tau}_1$ is represented as a sequence of 5 segments. The number and the WCET of threads of each segment are identified and shown in Figure 3.2. Subtask $\tau_{1,1}$ is spread on multiple segments. It is divided into two threads executing in segments $S_{1,1}$ and $S_{1,2}$.

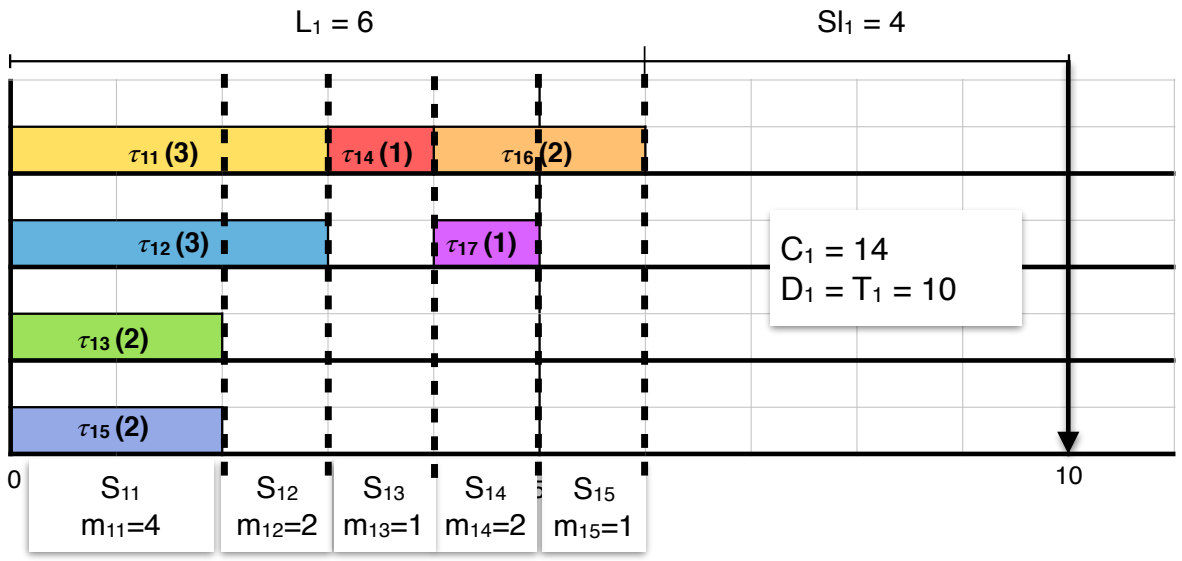


FIGURE 3.2: The Multi-Threaded Segment (MTS) representation $\overline{\tau_1}$ of DAG task τ_1 from Figure 3.1.

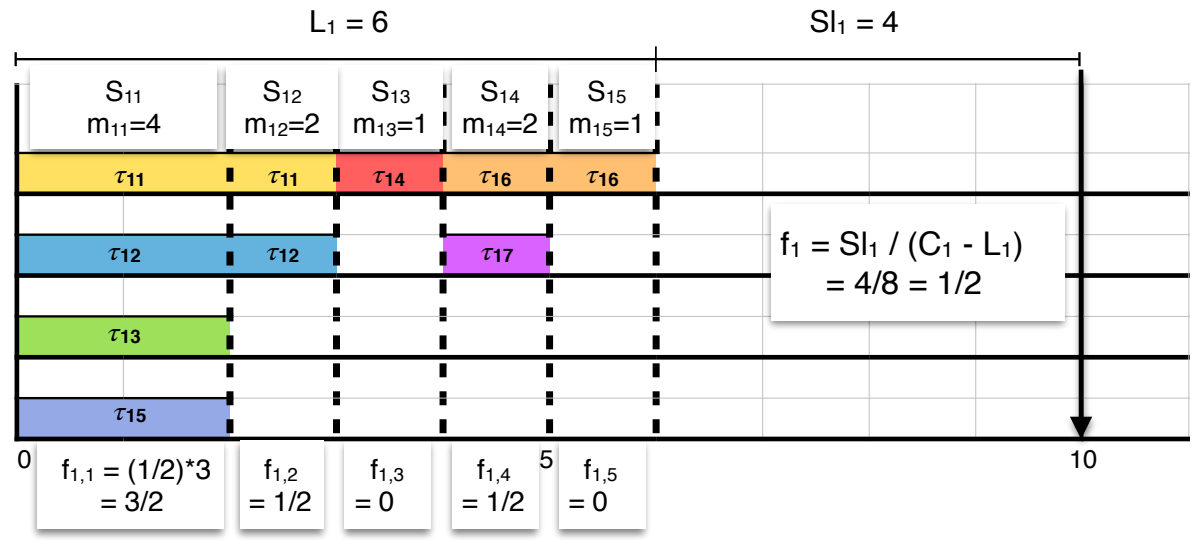
Moreover, the MTS task $\overline{\tau}_i$ shares the same deadline D_i and period T_i of DAG τ_i. Hence, Equation (3.1) regarding the slack Sl_i of τ_i remains correct for the MTS task $\overline{\tau}_i$. Also, the total WCET C_i of the original DAG task τ_i is the same for $\overline{\tau}_i$, but it can be represented differently based on the parameters of the MTS task $\overline{\tau}_i$ as follows:

$$C_i = \sum_{j=1}^{s_i} m_{i,j} \times c_{i,j}$$

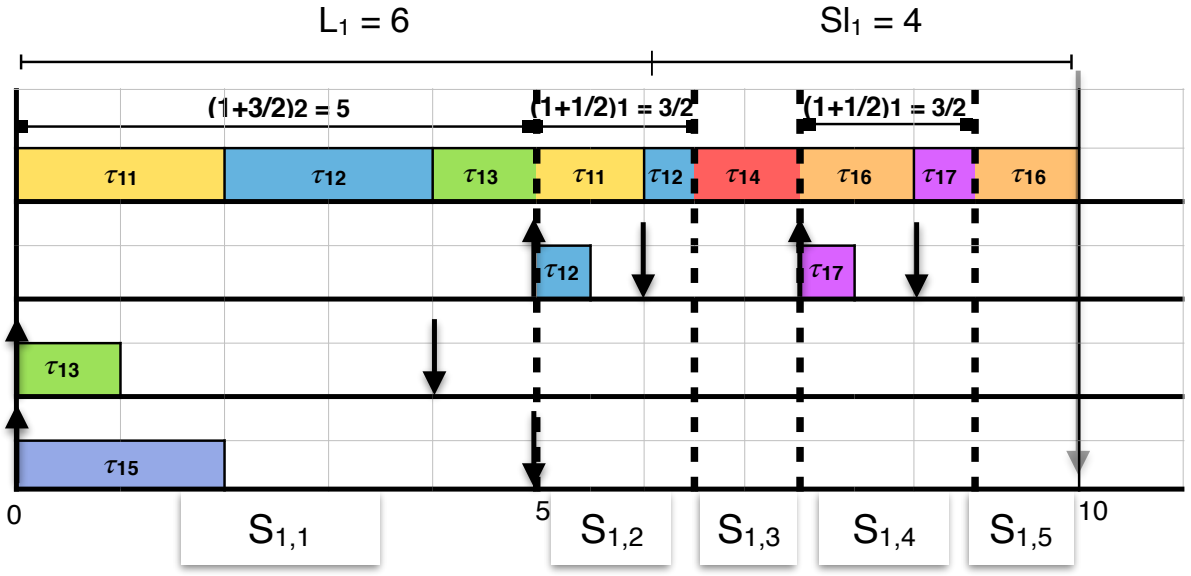
For a given DAG set τ, the DAG-Str algorithm is applied to the MTS form $\overline{\tau}_i$ of each DAG $\tau_i \in \tau$. The algorithm is explained in more details.

3.3.2 The DAG-Str Algorithm

As stated earlier, the DAG-Str algorithm executes DAGs as sequentially as possible. By doing so, the parallel structure of DAGs is removed and the dependencies between their subtasks are replaced by intermediate offsets and deadlines. The stretching algorithm is done based on the MTS representation $\overline{\tau}_i$ rather than on the DAG structure of τ_i. Hence, some threads of parallel segments of $\overline{\tau}_i$ are used to fill the slack Sl_i by adding them to the master thread of $\overline{\tau}_i$. The remaining threads are assigned intermediate offsets and deadlines based on the position of their segment w.r.t. the master thread. As a result, a subtask $\tau_{i,j}$ of DAG τ_i is said to be feasible if the threads from its corresponding $\overline{\tau}_i$ respect their intermediate deadlines.



(a) The MTS representation $\overline{\tau_i}$ of task τ_1 from Figure 3.1 showing the slack factors (f_i and $f_{i,j}$) derived using the stretching algorithm.



(b) An example of the DAG-Str algorithm when applied MTS task $\overline{\tau_1}$, showing the resulting the master thread and the constrained-deadline threads.

FIGURE 3.3: Example of the DAG stretching algorithm applied on DAG τ_1 from Figure 3.1.

The implicit-deadline DAG-Str algorithm is divided into two main cases, based on the timing parameters of the parallel DAG τ_i and its utilization U_i (the ratio between its total WCET C_i and relative deadline D_i):

- if $U_i \leq 1$, then DAG τ_i can be executed as a sequential task, because its total WCET C_i is less than its deadline D_i and it can be contained entirely within it. As a result, the DAG-Str algorithm transforms subtasks of τ_i into a single master thread τ_i^{master} in which all of the subtasks are forced to execute sequentially. In this case, the total WCET of the

stretched master thread C_i^{master} is equal to C_i and it has the same relative deadline and period of τ_i. If U_i is equal to 1, then the master thread is fully-stretched up to its deadline and the scheduler dedicates an entire processor for its execution. Otherwise, the master thread is an independent periodic implicit-deadline threads which can be scheduled with any multiprocessor scheduling algorithm.

It is worth mentioning that the stretching is done directly on the original DAG task τ_i and the use MTS representation $\overline{\tau}_i$ is not necessary, because the transformation of a DAG into a chain is trivial.

- if $U_i > 1$, then it is impossible for DAG τ_i to be completely transformed into a single master thread, since its total WCET C_i is larger than its deadline D_i. Thus, the stretching algorithm cannot avoid parallelism in this case. Applying the DAG-Str algorithm on $\overline{\tau}_i$ generates a fully-stretched master thread τ_i^{master} in addition to a set $\widehat{\tau}_i$ of constrained-deadline threads with intermediate offsets and deadlines. These extra timing parameters of each thread are important to keep the dependencies between the subtasks of the original task, and prevent parallel execution between threads of the same subtask. As in the case of the stretched master thread which has a utilization equal to 1, the scheduler assigns the generated fully-stretched master thread to a dedicated processor for its execution, while the constrained-deadline threads are scheduled independently on the available processors of the system.

As stated earlier, the DAG-Str algorithm aims at stretching the critical path of task $\overline{\tau}_i$ up to its deadline and create a fully-stretched master thread. In order to do so, certain non-critical threads from all segments of $\overline{\tau}_i$ are used to fill the slack Sl_i of the task uniformly. To achieve a uniform filling of the slack, fragments from these threads are added to the master thread. In the case of parallel tasks with utilization greater than 1, the total WCET C_i of $\overline{\tau}_i$ including its critical path length L_i definitely exceeds its deadline D_i. The remaining amount of execution time which can be used to fill the slack is equal to $(C_i - L_i)$. We define a unit factor f_i of each execution unit in $(C_i - L_i)$ that have to be added to the slack Sl_i as follows:

$$\begin{aligned} f_i &= \frac{Sl_i}{C_i - L_i} = \frac{D_i - L_i}{C_i - L_i} \\ &< \frac{D_i - L_i}{D_i - L_i} = 1 \qquad \longleftarrow \text{since } C_i > D_i \end{aligned} \tag{3.2}$$

In order to clarify the meaning of factor f_i, consider that each non-critical execution time unit in task $\overline{\tau}_i$ is divided into two parts. First part of length equal to f_i is added to the master thread τ_i^{master} and $(1 - f_i)$ executes in parallel with τ_i^{master}. The total execution time added from non-critical threads to the master thread is equal to Sl_i. However, forcing each execution unit in $(C_i - L_i)$ to be interrupted and migrated between processors is not practical, and might lead to huge migration. So, the filling of the slack is based on the execution requirement of each segment in the task rather than of the threads directly. We fill the slack with the maximum number of entire threads from each segment. Thus, at most one thread from each segment will be partially used to fill the slack. From the MTS representation $\overline{\tau}_i$, each segment $S_{i,j} \in S_i \in \overline{\tau}_i$ has $m_{i,j}$ threads in which one of them is a critical thread. Hence, the total WCET of non-critical threads in segment $S_{i,j}$ is equal to $(c_{i,j} \times (m_{i,j} - 1))$. Based on the definition of f_i (from Equation (3.2)), we conclude that a total of $(f_i \times c_{i,j}(m_{i,j} - 1))$ time units from each segment $S_{i,j} \in S_i$ will be added to the master thread.

Now, we want to identify how many threads of each segment $S_{i,j}$ are added to the master thread based on the total execution time of the segment. We identify how many entire threads we can add to the master thread since the threads of segment $S_{i,j}$ have equal WCETs. Let $f_{i,j}$ denote the segment factor of $S_{i,j}$ which determines the number of threads to be added to the master thread:

$$\begin{aligned} f_{i,j} &= \frac{f_i \times c_{i,j}(m_{i,j} - 1)}{c_{i,j}} \\ &= f_i \times (m_{i,j} - 1) \end{aligned} \tag{3.3}$$

According to Equation (3.3), each segment $S_{i,j}$ adds $\lfloor f_{i,j} \rfloor$ entire threads and a part of a thread of length $(f_{i,j} - \lfloor f_{i,j} \rfloor)c_{i,j}$ to the master thread. As a result, the slack Sl_i of task $\overline{\tau}_i$ is completely filled and a fully-stretch master thread τ_i^{master} with utilization equal to 1 is generated. We conclude that each segment $S_{i,j}$ adds in total $(1 + f_{i,j})$ time units from its threads to the master thread (including the critical thread), while the remaining threads of the segment execute in parallel with the master thread. In order to maintain dependencies and precedence constraints between subtasks of original DAG τ_i, segments should execute sequentially and the threads of a segment should finish their execution before the successor segment is activated. Since the master thread is fully stretched, then it has to no slack and it cannot be delayed. The parallel threads of a segment have execution window equal to the length of their threads in the master thread.

Hence, each segment $S_{i,j}$ has an intermediate relative deadline $D_{i,j}$ calculated as follows:

$$D_{i,j} = (1 + f_{i,j}) \times c_{i,j} \tag{3.4}$$

From the definition of the MTS model and since segments of a task $\overline{\tau}_i$ execute sequentially, the activation time of a segment is equal to the finish time of its predecessor segment. At any time $t \geq 0$, there is only one active segment from active job of each task $\overline{\tau}_i \in \tau$. According to this, we can define an intermediate offset $O_{i,j}$ for each segment $S_{i,j} \in \overline{\tau}_i$ based on the intermediate relative deadlines of the segments, where:

$$\forall S_{i,j} : j > 1 \quad \rightarrow \quad O_{i,j} = \sum_{k=1}^{j-1} D_{i,k}$$

and $O_{i,1} = 0$ (Relative offset w.r.t. the offset of DAG τ_i).

After applying the DAG-Str algorithm, a segment $S_{i,j}$ of $\overline{\tau}_i$ is defined by:

- a constrained-deadline thread $\tau_{i,j}^{master}$ which is part of the master thread τ_i^{master} of $\overline{\tau}_i$, and has a WCET equal to its relative deadline where $C_{i,j}^{master} = D_{i,j}$.
- $(m_{i,j} - \lfloor f_{i,j} \rfloor - 2)$ independent remaining constrained-deadline threads with a WCET equal to $c_{i,j}$ and a deadline $D_{i,j}$.
- one remaining thread with WCET equal to $(1 + \lfloor f_{i,j} \rfloor - f_{i,j})c_{i,j}$ and a constrained-deadline $(1 + \lfloor f_{i,j} \rfloor)c_{i,j}$. The remaining WCET is added to the end of the master thread $\tau_{i,j}^{master}$ (just before the deadline of its segment) so as to fill its slack. We can notice that the remaining thread has a shorter deadline than the other entire remaining threads from the same segment. This is done so as to force the first part of the thread (which is a constrained-deadline thread) to finish its execution before the start of the second fragment within the master thread. As a result, both fragments of the divided thread are guaranteed to execute sequentially.

For each segment $S_{i,j}$, the total number of remaining threads in the segment (including the partial thread), which execute in parallel with the master thread, is given by:

$$q_{i,j} = m_{i,j} - \lfloor f_{i,j} \rfloor - 1 \tag{3.5}$$

Accordingly, $q_{i,j}$ independent threads from segment $S_{i,j}$ of task $\overline{\tau}_i \in \tau$ are scheduled using any regular multiprocessor scheduling algorithm. While the fully-stretched master threads are assigned to dedicated processors. After applying the DAG-Str algorithm on each task $\overline{\tau}_i \in \tau$, each task generates at most one fully-stretched master thread. Hence, the total number of fully-stretched master threads, and the number of their dedicated processors, cannot exceed n, which is the total number of tasks in the original DAG set τ. As a result, the constrained-deadline tasks are scheduled on the $\overline{m}$ remaining processors of the system. The relation between the remaining processors $\overline{m}$ and the total number of processors in the system m is given as follows:

$$\overline{m} \geq m - n \tag{3.6}$$

Algorithm 3.1 shows the DAG-Str algorithm with its steps and the generated threads after stretching. Its input is the DAG task τ_i to be stretched and it returns the generated master thread τ_i^{master} (fully-stretched or not) and the set of constrained-deadline threads $\widehat{\tau}_i$. Now, we present an example of the stretching algorithm applied to DAG task τ_1.

Example 3.2 (DAG Stretching Algorithm)**.**
We consider the DAG τ_1 from Figure 3.1. The DAG-Str algorithm is summarized in three main steps which are shown in Figures 3.2 and 3.3. We assume that task τ_1 is a periodic implicit-deadline DAG with $C_1 = 14$ and a deadline $D_1 = 10$. Its utilization U_1 is equal to 1.4 which is greater than 1, hence it cannot be completely transformed into a sequential thread and cannot be executed on a single processor. As a result, it has to be represented by its MTS form $\overline{\tau}_1$ as shown in Figure 3.2. The subtasks of τ_1 execute as soon as possible when we consider an execution platform of infinite number of processors, and the only blocking effect on a subtask is due to its predecessors.

The MTS task $\overline{\tau}_1$ consists of 5 segments. Segment $S_{1,1}$ has 4 threads with $c_{1,1} = 2$, while the threads of other segments have WCET equal to 1. Segments $S_{1,2}$ and $S_{1,4}$ have two threads and segments $S_{1,3}$ and $S_{1,5}$ have a single thread. The critical path of τ_1 is considered the master thread τ_1^{master} of $\overline{\tau_1}$, and its length L_1, which is equal to 6, represents the sequential execution time of segments in $\overline{\tau_1}$. Based on Equation (3.1), the slack Sl_1 is equal to 4. The total WCET of τ_1 is equal to 14 and the non-critical execution time (critical subtasks are excluded) is equal to 8. According to Equation (3.2), the unit factor of $\overline{\tau}_1$ is equal to $f_1 = \frac{4}{8} = \frac{1}{2}$. From Equation (3.3), factor $f_{1,j}$ of segment $S_{1,j}$ depends on the number of its non-critical threads, where $f_{1,1} = \frac{1}{2} \times (4-1) = \frac{3}{2}$, $f_{1,2} = f_{1,4} = \frac{1}{2}$ and $f_{1,3} = f_{1,5} = 0$, as shown in Figure 3.3(a).

Algorithm 3.1 DAG Stretching (DAG-Str) Algorithm

Input: $\tau_i = (n_i, \{1 \leq j \leq n_i | \tau_{i,j}\}, G_i, D_i)$
Output: $\tau_i^{master}, \widehat{\tau}_i$

if $C_i == T_i$ **then** ▷ Convert DAG τ_i into a fully-stretched sequential thread
  for $\forall\ \tau_{i,j} \in \tau_i$ **do**
    $\tau_i^{master} \leftarrow \tau_i^{master} \cup \{\tau_{i,j} : C_{i,j}\}$
  end for
else
  if $C_i < T_i$ **then** ▷ Convert DAG τ_i into a single sequential thread
    for $\forall\ \tau_{i,j} \in \tau_i$ **do**
      $\widehat{\tau}_i \leftarrow \widehat{\tau}_i \cup \{\tau_{i,j} : C_{i,j}\}$
    end for
  else
    $\overline{\tau}_i \leftarrow DagToMTS(\tau_i)$
      ▷ Represent DAG task τ_i as a multi-threaded segment task $\overline{\tau}_i$.
    $f_i \leftarrow \frac{Sl_i}{C_i - L_i}$
    for $\forall\ S_{i,j} \in S_i$ **do**
      $f_{i,j} \leftarrow f_i \times (m_{i,j} - 1)$
      $q_{i,j} \leftarrow \lfloor f_{i,j} \rfloor + 1$
      for $k = 1$ to $q_{i,j}$ **do** ▷ $q_{i,j}$ threads of segment $S_{i,j}$ are added to the master thread
        $\tau_i^{master} \leftarrow \tau_i^{master} \cup \{\overline{\tau}_{i,j}^{k} : c_{i,j}\}$
      end for
      $\tau_i^{master} \leftarrow \tau_i^{master} \cup \{\overline{\tau}_{i,j}^{(q_{i,j}+1)} : (f_{i,j} - \lfloor f_{i,j} \rfloor) \times c_{i,j}\}$
      $\tau_{i,j}^{tmp} \leftarrow \{\overline{\tau}_{i,j}^{(q_{i,j}+1)} : (1 + \lfloor f_{i,j} \rfloor - f_{i,j}) \times c_{i,j}\}$
      $D_{i,j}^{tmp} \leftarrow (1 + \lfloor f_{i,j} \rfloor) \times c_{i,j}$
      if $j == 1$ **then**
        $O_{i,j}^{tmp} \leftarrow O_i$
      else
        $O_{i,j}^{tmp} \leftarrow O_{i,(j-1)} + D_{i,(j-1)}$
      end if
      $\widehat{\tau}_i \leftarrow \widehat{\tau}_i \cup \tau_{i,j}^{tmp}$
      for $k = (q_{i,j} + 2)$ to $m_{i,j}$ **do**
        $\tau_{i,j}^{tmp} \leftarrow \{\overline{\tau}_{i,j}^{k} : c_{i,j}\}$
        $D_{i,j}^{tmp} \leftarrow (1 + f_{i,j}) \times c_{i,j}$
        if $j == 1$ **then**
          $O_{i,j}^{tmp} \leftarrow O_i$
        else
          $O_{i,j}^{tmp} \leftarrow O_{i,(j-1)} + D_{i,(j-1)}$
        end if
        $\widehat{\tau}_i \leftarrow \widehat{\tau}_i \cup \tau_{i,j}^{tmp}$
      end for
    end for
  end if
end if

$(O_i^{master} \leftarrow O_i$
$(D_i^{master} \leftarrow D_i$
return $(\tau_i^{master}, \widehat{\tau}_i)$

It is worth noticing that segments with a single thread have factors equal to zero, because they contain a single critical thread that is already included in the master thread.

Based on the segment factor $f_{i,j}$, we can identify how many threads to be added to the master thread from each segment. As described earlier, $(1 + f_{i,j})$ threads from segment $S_{i,j}$ are used to fill the master thread. Figure 3.3(b) shows the stretching of task τ_1 and its final result. Two entire threads and a half from segment $S_{1,1}$ are added to the master thread where each thread has a WCET equal to 2. If we consider that the original DAG task τ_1 has no offset, then the first segment $S_{1,1}$ has no offset and its sequential execution time after stretching defines its intermediate deadline and it is calculated as $D_{1,1} = \frac{5}{2} \times 2 = 5$. The third thread of segment $S_{1,1}$ (denoted by $\tau_{1,3}$ in Figure 3.3(b)) is divided into two parts of the same length, one is added to the master thread τ_1^{master} and the other part is an independent constrained-deadline thread with an offset $O_{1,3} = 0$ and a relative deadline $D_{1,3} = 4$. The fourth remaining thread of the segment has a deadline equal to the deadline of the segment which is 5. The difference between both threads is that thread $\tau_{1,3}$ has to finish earlier than the deadline of the segment so as to give its other fragment (the one added to the master thread) enough time to execute. Fragments from the same thread have to execute sequentially and never in parallel, which is guaranteed by the assignment of an earlier relative deadline. By applying the same calculations for all of the segments, we obtain the result in Figure 3.3(b). Segments $S_{1,2}$ and $S_{1,4}$ add $\frac{1}{2}$ thread to the master thread. This means that there is only one partial thread left to execute in parallel with the master thread from each segment.

As shown in Figure 3.3(b), the DAG-Str algorithm generates a master thread τ_1^{master} with WCET and deadline equal to 10, and 4 constrained-deadline threads grouped in set $\widehat{\tau}_1$ as follows: $\widehat{\tau}_1$ = $\{\{\tau_{1,3} : (0, 1, 4, 10)\}, \{\tau_{1,5} : (0, 2, 5, 10)\}, \{\tau_{1,2} : (5, 0.5, 1, 10)\}, \{\tau_{1,7} : (7.5, 0.5, 1, 10)\}\}$, where each thread is identified by its intermediate offset, WCET, intermediate deadline and period. These threads are scheduled as independent constrained-deadline threads on $\overline{m}$ processors. In this example, we consider that task set τ contains a single DAG τ_1 which executes on a system of 2 identical processors. Then the master thread occupies one processor for itself, and the other is used for the scheduling of the parallel threads, then $\overline{m} = 1 \geq (2-1)$ (based on Equation (3.6)).

Lemma 3.1. *If a stretched task set is schedulable using any scheduling algorithm on m processors, then the original DAG task set τ (before stretching) is also schedulable.*

Proof. As described above, the DAG-Str algorithm assigns intermediate offsets and deadlines for threads of each DAG in τ without changing their original offsets and deadlines. Intermediate

offsets and deadlines maintain the precedence constraints of the original DAG, which means that if the stretched task set is schedulable while respecting the intermediate deadlines of the threads, then the original DAG task set τ must be schedulable as well while respecting the DAGs' deadlines. □

3.3.3 Resource Augmentation Bound Analysis

In this section, we provide a performance analysis to show the effectiveness of the DAG-Str algorithm by calculating its resource augmentation bound[1] (speedup factor) when GEDF scheduling algorithm is used. The performance of a scheduling algorithm is evaluated based on the calculation of the required increase of processor speed in order to guarantee the schedulability of feasible task sets. This performance metric is useful for comparing the performance of a scheduling algorithm with other DAG scheduling approaches and with algorithms found in the state-of-the-art.

Global Earliest Deadline First (GEDF) Scheduling Algorithm

We analyze the performance of DAG-Str algorithm of parallel tasks by calculating its resource augmentation bound when preemptive GEDF is used to schedule the constrained-deadline threads $\widehat{\tau}_i$ generated from each stretched DAG $\tau_i \in \tau$ on $\overline{m}$ processors of the system, while the fully-stretched master threads are assigned their own processors. GEDF is a common scheduling algorithm in real-time systems, despite of its optimality loss when it is used on multiprocessor systems.

Let $\widehat{\tau}$ be the set of all constrained-deadline threads generated after applying the DAG-Str algorithm on every task in $\tau_i \in \tau$ (fully-stretched master threads are excluded, and implicit-deadline stretched master threads from tasks with utilization less than 1 are included). We prove that GEDF scheduling of $\widehat{\tau}$ when executing on $\overline{m}$ processors, has a resource augmentation bound equal to $\frac{3+\sqrt{5}}{2}$ for all tasks with $n < \varphi \times \overline{m}$, where φ is the golden ratio whose value is equal to $\frac{1+\sqrt{5}}{2}$, the remaining number of processors is denoted by $\overline{m} \geq m - n$, where m is the original number of processors in the system and n is the number of tasks in the set τ. This implies that if a task set $\widehat{\tau}$ is feasible on $\overline{m}$ unit-speed processors, then it is schedulable using GEDF on $\overline{m}$ processors of speed $\frac{3+\sqrt{5}}{2}$. For task sets that do not satisfy the condition $n \geq \varphi \times \overline{m}$,

[1] Check Resource Augmentation Bound Definition 1.3 on page 9

a lower bound of speedup factor equal to 4 is available by using the decomposition algorithm from [108, 109].

Our resource augmentation bound analysis is based on the following sufficient schedulability condition of GEDF algorithm.

Theorem 3.2 (From [17]). *Any sporadic constrained-deadline sequential task set τ, with a total density $\delta^{sum}(\tau)$ and a maximum density $\delta^{max}(\tau)$, is schedulable using preemptive GEDF policy on m unit-speed processors if*

$$\delta^{sum}(\tau) \leq m - (m-1)\delta^{max}(\tau)$$

Theorem 3.3. *Let $\widehat{\tau}$ be a constrained-deadline task set of n parallel threads with $n < \varphi.\overline{m}$ which is feasible on $\overline{m}$ unit-speed processors. Then, $\widehat{\tau}$ is schedulable using GEDF on $\overline{m}$ processors of speed greater than or equal to $\frac{3+\sqrt{5}}{2}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ (the golden ratio).*

Proof. Task set $\widehat{\tau}$ is a set of independent constrained-deadline threads that is generated after applying the DAG-Str algorithm. Hence, we can apply the scheduling condition from Theorem 3.2 and it is schedulable on the remaining processors $\overline{m}$ of the system if:

$$\delta^{sum}(\widehat{\tau}) \leq \overline{m} - (\overline{m}-1)\delta^{max}(\widehat{\tau})$$

We start by calculating the maximum thread density $\delta^{max}(\widehat{\tau})$ of all the threads in $\widehat{\tau}$. As stated earlier, $\widehat{\tau}$ consists of two types of threads: implicit-deadline stretched master threads from DAG tasks with utilization less than 1 and a collection of constrained-deadline threads from each DAG task with a utilization higher than 1. We consider both cases for calculating the maximum thread density $\delta^{max}(\widehat{\tau})$:

- **Case 1**: an implicit-deadline DAG task τ_i with $U_i < 1$, is completely transformed into a single master thread τ_i^{master} with the same timing parameters as the original DAG τ_i. This means that τ_i^{master} has a WCET equal to C_i and a deadline and period equal to D_i. Its density $\delta_i^{master} = U_i$, which is definitely less than 1 from our initial assumption. Thus, $\delta_i^{master} < 1$.

- **Case 2**: an implicit-deadline DAG task τ_i with $U_i > 1$ is transformed using the DAG-Str algorithm into a fully-stretched master thread and a set of constrained-deadline threads with intermediate offsets and deadlines. Based on the structure of multi-threaded segment task $\overline{\tau}_i$, only one segment is active at every time instant t. Each segment $S_{i,j} \in S_i \in \overline{\tau}_i$, with $m_{i,j} > 1$, has at most two types of parallel threads, entire and partial threads. The threads of segment $S_{i,j}$ (other than the one in the master thread) have a maximum density $\delta_{i,j}^{max}$ of:

$$\begin{aligned}
\delta_{i,j}^{max} &= \max(\delta_{i,j}^{entire}, \delta_{i,j}^{partial}) \\
&= \max\{\frac{c_{i,j}}{(1+f_{i,j})c_{i,j}}, \frac{(1+\lfloor f_{i,j} \rfloor - f_{i,j})c_{i,j}}{(1+\lfloor f_{i,j} \rfloor)c_{i,j}}\} \\
&= \max\{\frac{1}{1+f_{i,j}}, \frac{1-(f_{i,j}-\lfloor f_{i,j} \rfloor)}{(1+f_{i,j})-(f_{i,j}-\lfloor f_{i,j} \rfloor)}\} \\
&= \frac{1}{1+f_{i,j}} \qquad \leftarrow \forall \ m_{i,j} > 1
\end{aligned} \tag{3.7}$$

From Equation (3.7), the maximum density of threads of segment $S_{i,j}$ in $\overline{\tau}_i$ depends on the segment factor $f_{i,j}$. From Equation (3.3), $f_{i,j}$ depends on the number of threads in this segment. Hence, the segment with the smallest number of threads has the highest density. We can conclude that the highest density of a thread in task $\overline{\tau}_i$ belongs to the segment with the smallest possible number of threads, where $m_{i,j} = 2$.

$$\begin{aligned}
\delta_i^{max} &= \frac{1}{1+(f_i \times (2-1))} \qquad \rightarrow \quad \text{from Eq. (3.3)} \\
&= \frac{1}{1+f_i}
\end{aligned} \tag{3.8}$$

From Equation (3.2), we can derive the following relation:

$$\begin{aligned}
\forall f_i \geq 0 \quad \rightarrow \quad \delta_i^{max} = \frac{1}{1+f_i} \quad \text{and} \quad f_i \geq \frac{D_i - L_i}{C_i} \\
\frac{1}{1+f_i} \leq \frac{1}{1+\frac{D_i-L_i}{C_i}} = \frac{C_i}{C_i+D_i-L_i} \leq 1 \\
(\text{Since } L_i \leq D_i).
\end{aligned}$$

When we consider both cases, the maximum density $\delta^{max}(\widehat{\tau})$ of all threads in $\widehat{\tau}$ is:

$$\begin{aligned}
\delta^{max}(\widehat{\tau}) &= \max_{\widehat{\tau}_i \in \widehat{\tau}} \{\delta_i^{master}, \delta_i^{max}\} \\
&\leq 1
\end{aligned} \tag{3.9}$$

It is worth noticing here that DAG tasks with utilization equal to 1 are not considered in the previous cases, because the DAG-Str algorithm transforms them into fully-stretched master threads that are assigned to specific processors, and they are not scheduled using GEDF algorithm. Hence, they are not included in the resource augmentation bound of GEDF scheduling.

The total density $\delta^{sum}(\widehat{\tau})$ of thread set $\widehat{\tau}$ is the sum of densities of all threads in $\widehat{\tau}$. At any time instant t, there is a single active segment from each stretched DAG task whose utilization is greater than 1, in addition to the stretched master thread from DAG tasks with utilization less than 1. We calculate the density in the first case by considering the threads of the segment with the highest density. Let δ_i^{sum} be the sum of densities of such threads in a certain segment $S_{i,j}$ in a task $\overline{\tau}_i$:

$$\begin{aligned}
\delta_i^{sum} &\leq \frac{1}{1+f_{i,j}} \times (m_{i,j} - 1 - \lfloor f_{i,j} \rfloor) && \longrightarrow \quad \text{from Eq. (3.5)} \\
&\leq \frac{m_{i,j}-1}{f_i \times (m_{i,j}-1)} \\
&\leq \frac{1}{f_i} \\
&\leq \frac{C_i - L_i}{D_i - L_i} \leq \frac{C_i}{D_i - L_i} && \longrightarrow \quad \text{from Eq. (3.2)}
\end{aligned} \tag{3.10}$$

Now, we consider the second case where DAG task τ_i has a utilization less than 1 and is transformed into a periodic implicit-deadline stretched master thread. The density of the task is denoted by $\delta_{master} = \dfrac{C_i}{D_i} \leq \delta_i^{sum}$.

For thread set $\widehat{\tau}$, let $\delta^{sum}(\widehat{\tau})$ be the sum of densities of every DAG task in the original task set τ:

$$\begin{aligned}
\delta^{sum}(\widehat{\tau}) &\leq \sum_{\forall U_i > 1} \frac{C_i}{D_i - L_i} + \sum_{\forall U_i < 1} \frac{C_i}{D_i} \\
&\leq \sum_{\tau_i \in \tau} \frac{C_i}{D_i - L_i}
\end{aligned} \tag{3.11}$$

In order to calculate the resource augmentation bound, we consider that thread set $\widehat{\tau}$ executes on $\overline{m}$ processors with a minimum speed ν, where $\nu > 1$. Increasing the speed of processors affects the execution parameters of task τ_i such as C_i and L_i, while the relative deadline D_i and period T_i stay the same.

The critical path length L_i of any task τ_i has a necessary feasibility condition:

$$\forall \tau_i \ : \ 1 \leq i \leq n \quad \longrightarrow \quad L_i \leq D_i \tag{3.12}$$

Otherwise, task τ_i is not feasible on unit-speed processors.

On a processor that is ν times faster, the critical path length L_i^ν is given by:

$$\forall \tau_i \ : \ 1 \leq i \leq n \quad \longrightarrow \quad L_i^\nu = \frac{L_i}{\nu} \leq \frac{D_i}{\nu} \tag{3.13}$$

The same notation is applied to the WCET C_i^ν of task τ_i when it runs on processors ν times faster:

$$\forall \tau_i \ : \ 1 \leq i \leq n \quad \longrightarrow \quad C_i^\nu = \frac{C_i}{\nu} \tag{3.14}$$

The total density of task set $\delta^{sum,\nu}$ on speed-ν processors is:

$$\delta^{sum,\nu}(\widehat{\tau}) \leq \sum_{\tau_i \in \tau} \frac{C_i^\nu}{D_i - L_i^\nu} \tag{3.15}$$

From Equation (3.13),

$$\forall 1 \leq i \leq n \quad \longrightarrow \quad L_i^\nu \leq \frac{D_i}{\nu} \Rightarrow (D_i - L_i^\nu) \geq D_i(1 - \frac{1}{\nu})$$

Using this result,

$$\begin{aligned} \delta^{sum,\nu}(\widehat{\tau}) &\leq \sum_{\tau_i \in \tau} \frac{C_i/\nu}{D_i(1 - \frac{1}{\nu})} \\ &\leq \frac{1}{\nu - 1} \sum_{\tau_i \in \tau} \frac{C_i}{D_i} \\ &\leq \frac{m}{\nu - 1} \end{aligned}$$

From Equation (3.9), the maximum density $\delta^{max,\nu}(\widehat{\tau})$ of task set $\widehat{\tau}$ on speed-ν processors is:

$$\delta^{max,\nu}(\widehat{\tau}) = \frac{\delta^{max}}{\nu} \leq \frac{1}{\nu} \tag{3.16}$$

Therefore, task set $\widehat{\tau}$ is schedulable under global preemptive EDF on $\overline{m}$ speed-ν processors if:

$$\frac{m}{\nu-1} \leq \frac{\overline{m}+n}{\nu-1} \leq \overline{m} - (\overline{m}-1)\frac{1}{\nu}$$

$$\begin{aligned} \frac{\overline{m}}{\nu-1} + \frac{n}{\nu-1} &\leq \overline{m} - \frac{\overline{m}}{\nu} + \frac{1}{\nu} \\ \frac{1}{\nu-1} + \frac{1}{\nu} - \frac{1}{\overline{m}\nu} \leq \frac{1}{\nu-1} + \frac{1}{\nu} &\leq 1 - \frac{n}{(\nu-1)\overline{m}} \\ \frac{2\nu-1}{\nu^2-\nu} &\leq 1 - \frac{n}{\overline{m}(\nu-1)} \end{aligned}$$

In order to have a positive value for the right hand side of the inequality, we consider the following:

$$0 < \frac{n}{\overline{m}(\nu-1)} < 1 \Rightarrow \overline{m} > \frac{n}{\nu-1}$$

$$\begin{aligned} \frac{2\nu-1}{\nu^2-\nu} &< 1 \\ 0 &< \nu^2 - 3\nu + 1 \\ \nu \geq \frac{3+\sqrt{5}}{2} \quad \text{for } n < \varphi.\overline{m} \text{ , where } \varphi &= \frac{1+\sqrt{5}}{2} \end{aligned}$$

where φ is the golden ratio.

This concludes the proof of the resource augmentation bound. □

3.4 Segment Stretching (Seg-Str) Algorithm

Based on the DAG-Str algorithm, the stretching of the master thread of a DAG task is done by filling its slack time with non-critical threads of the DAG. As a result, at most one thread from each segment in the DAG is forced to be preempted and executes as independent thread. It then continues its execution within the master thread. This technique ensures that threads of all segments are used to fill the slack time of the DAG and independent constrained-deadline threads are assigned a slack time. Until now, we considered that the cost of job preemption and

migration between processors is negligible, and it does not add considerable overheads to the system execution. Hence, we evaluate the schedulability performance of the DAG-Str algorithm without considering the costs of migration and preemption.

In this section, we present the Seg-Str (Segment Stretching) algorithm which is a modified version of the DAG-Str algorithm that reduces job migrations and preemptions caused by the stretching algorithm. Briefly, the Seg-Str algorithm forces a single thread from each stretched job to migrate between processors, which is less than the number of migrations forced by the DAG-Str algorithm. First, we start by presenting the Seg-Str algorithm and by explaining it in details using an example. Then, we compare the performance of both stretching algorithms w.r.t. schedulability and number of migrations. Finally, we prove that the Seg-Str algorithm has the same resource augmentation bound as the DAG-Str algorithm when global EDF is used.

3.4.1 Concept and Algorithm

In the case of DAG-Str algorithm, each segment $S_{i,j}$ from MTS task $\overline{\tau}_i \in \tau$ participates in the master thread τ_i^{master} with a sum of WCET equal to $(1 + f_{i,j})c_{i,j}$. If segment factor $f_{i,j}$ is not integer (i.e., $f_{i,j} > \lfloor f_{i,j} \rfloor$), then a thread from segment $S_{i,j}$ is divided into two fragments, the first one executes sequentially within the master thread of the task, while the second fragment executes independently as a constrained-deadline sequential thread.

The Seg-Str algorithm has the same objective of the DAG-Str algorithm. It aims at executing parallel tasks as sequentially as possible by filling the slack of the DAG with threads from its segments of its MTS form. It uses the unit and segment factors calculated earlier for DAG $\overline{\tau}_i$ (f_i from Equation (3.2) and $f_{i,j}$ from Equation (3.3) respectively). However, the difference between both stretching algorithms lies in the slack filling step. In Seg-Str algorithm, only entire threads are allowed to fill the slack except for a single thread which is divided into two threads to allow a full stretching of the master thread.

Each segment $S_{i,j} \in S_i \in \overline{\tau}_i$ participates in the master thread with $(1 + \lfloor f_{i,j} \rfloor)$ entire threads. Let C_i^{rem} denote the remaining non-critical WCET after the entire threads are used to fill the slack time Sl_i, where:

$$C_i^{rem} = \sum_{\forall S_{i,j} \in S_i} (f_{i,j} - \lfloor f_{i,j} \rfloor) c_{i,j} \tag{3.17}$$

From this formula, we conclude that C_i^{rem} is less than the sum of sequential WCET of each segment with $m_{i,j}$ greater than $(1 + f_{i,j})$. Otherwise, there will be no remaining slack time

needed to be filled. So, instead of distributing the C_i^{rem} value on all of these segments fairly, as in the case of the DAG-Str algorithm, the Seg-Str algorithm calculates the maximum number of entire segments that can be added to the master thread and it identifies the segment from which a thread will be divided into two fragments (if necessary). Let $S_{i,x}$ be the segment which contains the divided thread that is used to fully stretch the master thread to its deadline. Let x_i be the thread factor where $(x_i \times c_{i,x})$ time units are added to the master thread. These 2 values can be calculated using an iterative algorithm shown in Algorithm 3.2.

From each segment $S_{i,j}$ which is a predecessor of segment $S_{i,x}$ (i.e., $j < x$), an entire thread is added to the master thread. Hence, the number of entire threads that execute within the master thread is equal to $(1+\lceil f_{i,j} \rceil)$. From Segment $S_{i,x}$, the thread, whose index is $(1+\lceil f_{i,j} \rceil)$, is divided into two fragments, one executes independently and the other within the master thread. Finally, successor segments of $S_{i,x}$ (whose indexes are lower than x) remain unchanged and $(1 + \lfloor f_{i,j} \rfloor)$ entire threads are added to the master thread. In conclusion, one thread at most from all segments of $\overline{\tau}_i$ is forced to migrate between processors due to the unfair filling of the slack time from segments.

Algorithm 3.2 Procedure to calculate x_i and $S_{i,x}$ for the Segment Stretching (Seg-Str) Algorithm

Input: $\overline{\tau}_i$
Output: $x_i, S_{i,x}$

$C_i^{rem} \leftarrow \sum_{\forall S_{i,j} \in S_i} (f_{i,j} - \lfloor f_{i,j} \rfloor) c_{i,j}$
for $\forall\, S_{i,j} \in S_i$ **do**
  if $m_{i,j} > (1 + f_{i,j})$ **then**
    if $C_i^{rem} < c_{i,j}$ **then**
      $S_{i,x} \leftarrow S_{i,j}$
      $x_i \leftarrow \frac{C_i^{rem}}{c_{i,j}}$
      break
    end if
  end if
  $C_i^{rem} \leftarrow C_i^{rem} - c_{i,j}$
end for
return $(x_i, S_{i,x})$

For any segment $S_{i,j} \in S_i$, its local intermediate deadline $D_{i,j}$ depends on its execution position w.r.t. segment $S_{i,x}$. It is defined as follows:

$$D_{i,j} = \begin{cases} (\lceil f_{i,j} \rceil + 1)c_{i,j}, & j < x \\ (\lceil f_{i,j} \rceil + x_i)c_{i,j}, & j = x \\ (\lceil f_{i,j} \rceil)c_{i,j}, & j > x \end{cases} \tag{3.18}$$

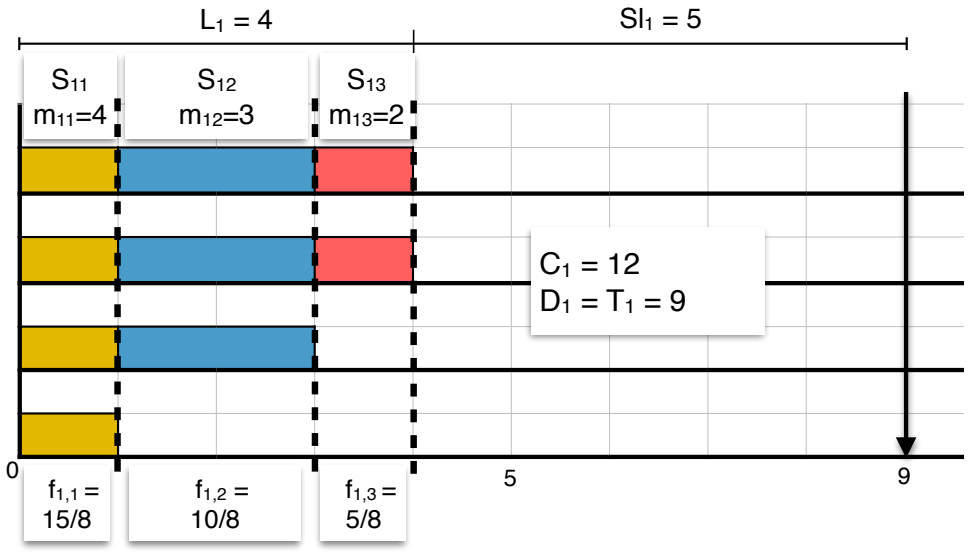
After applying the Seg-Str algorithm, a segment $S_{i,j}$ of $\overline{\tau}_i$ consists of the following threads:

- a constrained-deadline thread $\tau_{i,j}^{master}$ which is part of the master thread τ_i^{master} with a WCET $c_{i,j}$ equal to its relative deadline $D_{i,j}$.
- $n_{i,j}$ independent constrained-deadline threads with a WCET $c_{i,j}$ and a relative deadline $D_{i,j}$, where $n_{i,j}$ is defined as follows:

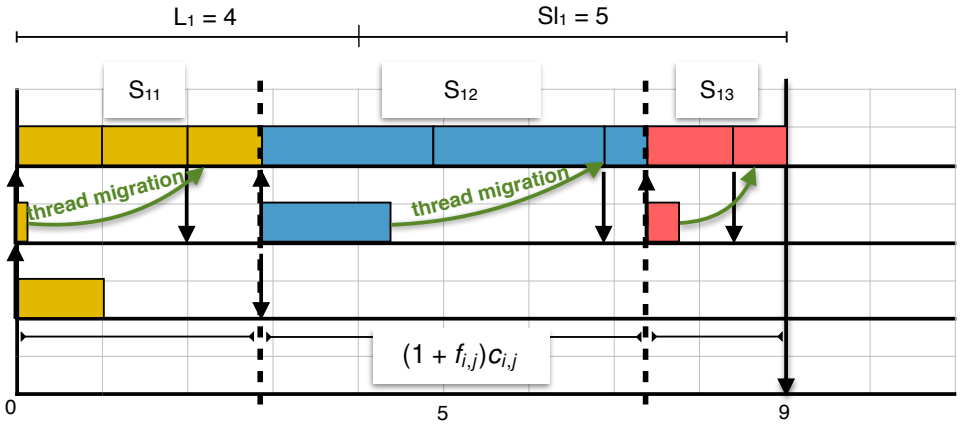
$$n_{i,j} = \begin{cases} (m_{i,j} - \lceil f_{i,j} \rceil - 1), & j < x \\ (m_{i,j} - \lceil f_{i,j} \rceil), & j \geq x \end{cases} \tag{3.19}$$

- if segment $S_{i,j} = S_{i,x}$, then it has an extra independent constrained-deadline thread whose WCET is equal to $(1 - x_i)c_{i,j}$ and its relative deadline is equal to $\lceil f_{i,j} \rceil c_{i,j}$. As in the case of DAG-Str algorithm, the deadline of the partial thread is shorter than the deadline of its segment, so as to prevent the two fragments of the same thread from executing in parallel, and force them to execute sequentially.

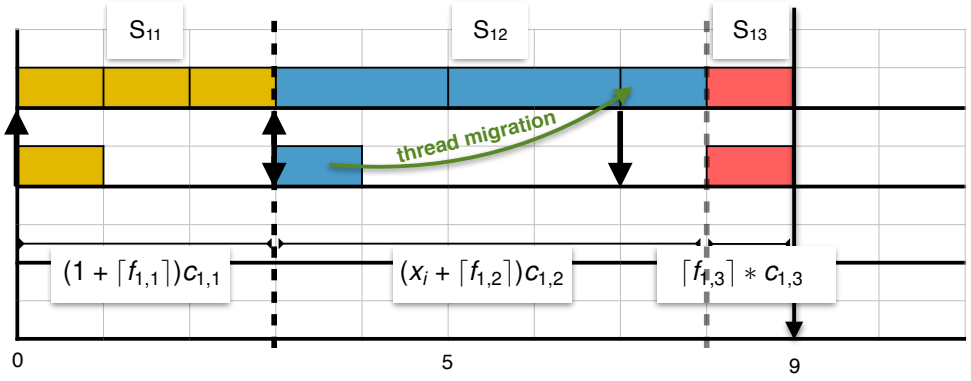
Example 3.3 (Segment Stretching Algorithm)**.**
In the example of Figure 3.4, we compare the Seg-Str algorithm with the DAG-Str algorithm. We consider a MTS task $\overline{\tau}_1$ *which consists of* 3 *segments as shown in Figure 3.4(a). Segment* $S_{1,1}$ *consists of* 4 *threads with* $c_{1,1} = 1$*. The second segment* $S_{1,2}$ *has* 3 *threads of* $c_{1,2} = 2$ *and finally segment* $S_{1,3}$ *consists of* 2 *threads with* $c_{1,3} = 1$*. Task* $\overline{\tau}_1$ *has an implicit-deadline equal to* 9*, a critical path length* L_1 *equal to* 4 *and a slack time* Sl_1 *equal to* 5*. We should mention here that we considered a MTS task directly and not the DAG task. We can imagine that* $\overline{\tau}_1$ *is the MTS representation form of a DAG task* τ_1 *with a total WCET* C_1 *equal to* 12 *and a deadline equal to* 9*, and precedence constraints between subtasks translated into the MTS form in Figure 3.4(a). We ignored the exact structure of the DAG task since it does not affect the stretching algorithms.*



(a) Multi-Threaded Segment (MTS) task $\overline{\tau}_1$.



(b) DAG-Str algorithm.



(c) Seg-Str algorithm.

FIGURE 3.4: Example of Seg-Str algorithm.

Based on the timing parameters of task $\overline{\tau}_1$*, its non-critical WCET is equal to* 8 *and its unit factor* f_1 *is equal to* $\frac{5}{8}$*. From Equation* (3.3)*, the factor of each segment is calculated as:* $f_{1,1} = \frac{15}{8}$*,* $f_{1,2} = \frac{10}{8}$ *and* $f_{1,3} = \frac{5}{8}$*. Starting by the DAG-Str algorithm, the result of our stretching algorithm is shown in Figure 3.4(b). Based on the distribution factors of segments, segment* $S_{1,1}$ *has* 2 *entire threads in the master thread in addition to a partial thread of WCET equal to* $\frac{7}{8}$*. The remaining independent threads of this segment are two threads, the first has a WCET equal to* $\frac{1}{8}$ *and a deadline equal to* 2*, while the other has a WCET equal to* 1 *and a deadline equal to* $\frac{23}{8}$*. As shown in the figure, a fragment of the first thread is forced to migrate to execute within the master thread, this thread migration is shown by a directed arrow in the figure. Similarly, segment* $S_{1,2}$ *adds two entire threads into the master thread in addition to* $\frac{1}{2}$ *execution time unit which is part of a single remaining thread. This thread has a WCET equal to* $\frac{3}{2}$ *time units and a deadline equal to* 4*. Finally, segment* $S_{1,3}$ *generates a single independent remaining thread with a WCET equal to* $\frac{5}{8}$ *time units and a deadline equal to* 1*. From this example, we can notice that each segment of task* $\overline{\tau}_1$ *has a partial remaining thread, and there are* 3 *forced migrations of threads so as to fully stretch the master thread.*

Figure 3.4(c) shows the result of applying the Seg-Str algorithm on the same task $\overline{\tau}_1$*. Based on the distribution factor of each segment, the remaining non-critical WCET* C_i^{rem} *is equal to* 2 *time units, and it is calculated as* $C_i^{rem} = (\frac{7}{8} + 2 \times \frac{2}{8} + \frac{5}{8}) = 2$ *(refer to Equation* (3.17)*). It is clear that this value is strictly less than the sequential execution time of segments which is equal to* $(1 + 2 + 1) = 4$*. Instead of dividing the* C_i^{rem} *on the three segments of* $\overline{\tau}_1$*, we calculate the values* $S_{i,x}$ *and* x_i *that determine which segment contains a partial thread (if any). Starting from the first segment* $S_{1,1}$*, it has* $c_{1,1} = 1$ *which can be used entirely to fill the master thread. Consequently, the remaining WCET* C_i^{rem} *is updated to be equal to* $(2 - 1) = 1$*. The number of remaining threads* $n_{1,1}$ *is calculated from Equation* (3.19)*, it is equal to* $n_{1,1} = (4 - \lceil \frac{15}{8} \rceil - 1 = 1)$*.*

Then we consider the second segment $S_{1,2}$ *whose WCET* $c_{1,2}$ *is equal to* 2*. Segment* $S_{1,2}$ *cannot be added entirely to the master thread because its sequential WCET is greater than the remaining WCET* C_i^{rem}*. Based on Algorithm 3.2, segment* $S_{1,2}$ *is considered as* $S_{1,x}$ *and* x_1 *is calculated as* $\frac{C_1^{rem}}{c_{1,2}} = \frac{1}{2}$*. According to this, a thread from this segment is divided into two parts of the same length, the first one is added to the end of the segment's master thread* $\tau_{1,2}^{master}$ *with WCET equal to* $\frac{1}{2} \times 2 = 1$*. The second half is an independent constrained-deadline thread with WCET equal to* $(1 - \frac{1}{2}) \times 2 = 1$ *and a deadline equal to* $(\lceil \frac{10}{8} \rceil \times 2) = 4$*. The number of remaining threads* $n_{1,2}$ *from this segment is equal to* $3 - \lceil \frac{10}{8} \rceil = 1$*.*

Regarding the last segment, $S_{1,3}$ is a successor of segment $S_{1,x}$, hence there is no remaining WCET ($C_1^{rem} = 0$). Only $\lceil \frac{5}{8} \rceil$ entire threads from this segment are added to its master thread $\tau_{1,3}^{master}$. As shown in Figure 3.4(c), there is a single remaining independent thread with WCET equal to $c_{1,3} = 1$ and its deadline is equal to 1 as well.

From this example, we can notice how the Seg-Str algorithm succeeds in reducing the number of forced migrations compared to the DAG-Str algorithm. As shown in Figure 3.4(b), there are 3 thread migrations due to the DAG-Str algorithm, which is proportional to the number of segments s_1 in the MTS task $\overline{\tau}_1$. However, it is guaranteed that a single migration is forced by applying the Seg-Str algorithm, and this is independent from the number of segments in the task, as shown in Figure 3.4(c).

3.4.2 Resource Augmentation Bound Analysis

In this section, we analyze the performance of the Seg-Str algorithm by calculating its resource augmentation bound (speedup factor). We prove that the modifications we have done to the DAG-Str algorithm that led to the Seg-Str algorithm do not affect its resource augmentation bound analysis. In the case of global preemptive EDF (GEDF) scheduling algorithm on multiprocessor systems, the Seg-Str algorithm has a resource augmentation bound equal to $\frac{3+\sqrt{5}}{2}$ under certain conditions as stated in Theorem 3.3 on page 68.

For a given DAG set and based on the concept of the Seg-Str algorithm, we can notice that the generated fully-stretched master threads are identical to the ones generated by the DAG-Str algorithm. Hence, the number of processors dedicated to these master threads remain the same, and the number of available processors for the set $\widehat{\tau}$ of independent periodic constrained-deadline threads is equal to $\overline{m}$ (where $\overline{m}$ is defined in Equation (3.6)). The DAG tasks whose utilization is less than 1 are transformed into a single sequential master thread as in the case of DAG-Str algorithm.

The only difference lies in the constrained-deadline threads that belong to $\widehat{\tau}$. The timing characteristics of these threads change when they cease to migrate between processors. In the remainder of this section, we prove that the timing changes do not affect the upper bounds of thread densities, in particular the maximum density of the threads $\delta^{max}(\widehat{\tau})$ and the total density $\delta^{sum}(\widehat{\tau})$ of task set $\widehat{\tau}$, which lead to the same resource augmentation bound based on GEDF schedulability condition from Theorem 3.2 on page 68.

Lemma 3.4. *For a given set $\widehat{\tau}$ of independent constrained-deadline threads generated by Seg-Str algorithm, the maximum thread density $\delta^{max}(\widehat{\tau})$ has the following upper bound:*

$$\delta^{max}(\widehat{\tau}) \leq 1$$

Proof. After applying the Seg-Str algorithm, the threads of set $\widehat{\tau}$ are divided into two two categories based on their corresponding DAG tasks:

- **Case 1:** if DAG task τ_i has a utilization U_i less than 1, then it is completely stretched to a single implicit-deadline master thread $\widehat{\tau}_i^{master}$. The density of this thread is $\delta_i^{master} = U_i < 1$.
- **Case 2:** if DAG task τ_i has a utilization U_i greater than 1, then the density of its threads depends on their corresponding segment and its relation with segment $S_{i,x}$. Threads of any segment $S_{i,j}$, which is a predecessor of $S_{i,x}$, have a density equal to $\frac{c_{i,j}}{(1+\lceil f_{i,j} \rceil)c_{i,j}}$. If segment $S_{i,j}$ is a successor of $S_{i,x}$, then the density of its threads is equal to $\frac{c_{i,j}}{\lceil f_{i,j} \rceil c_{i,j}}$. Finally, segment $S_{i,x}$ has two types of threads, entire threads with density equal to $\frac{c_{i,j}}{(x_i+\lceil f_{i,j} \rceil)c_{i,j}}$, and a partial thread with density equal to $\frac{(1-x_i)c_{i,j}}{\lceil f_{i,j} \rceil c_{i,j}}$.

Based on these densities and knowing that $(0 \leq x_i < 1)$, the maximum thread density $\delta^{max}(\widehat{\tau})$ is:

$$\begin{aligned}\delta^{max}(\widehat{\tau}) &= \max\left(\frac{1}{(1+\lceil f_{i,j} \rceil)}, \frac{1}{\lceil f_{i,j} \rceil}, \frac{1}{(x_i+\lceil f_{i,j} \rceil)}, \frac{(1-x_i)}{\lceil f_{i,j} \rceil}\right) \\ &\leq \frac{1}{\lceil f_{i,j} \rceil} \leq 1\end{aligned}$$

which includes the density bound calculated in Case 1, and the lemma is proved. □

Lemma 3.5. *For a given set $\widehat{\tau}$ of independent constrained-deadline threads generated by the Seg-Str algorithm, the total density $\delta^{sum}(\widehat{\tau})$ of thread set is:*

$$\delta^{sum}(\widehat{\tau}) \leq \sum_{\tau_i \in \tau} \frac{C_i}{D_i - L_i}$$

Proof. Based on the structure of the implicit-deadline Multi-Threaded Segment task, only one of its segments is active at any time instant t, and the threads of a segment cannot start their execution before the completion of their predecessor segments. Hence, the total density (δ_i^{sum}) of a MTS task $\overline{\tau}_i$ cannot exceed the maximum density of its segments S_i. From Lemma 3.4, the maximum density of any thread in $\widehat{\tau}_i$ cannot exceed $\frac{1}{\lceil f_{i,j} \rceil}$. From Equation (3.19), the maximum number of threads in any segment $S_{i,j} \in \overline{\tau}_i$ cannot exceed $(m_{i,j} - \lceil f_{i,j} \rceil)$, which is a successor segment of $S_{i,x}$. According to this, the total density of $\overline{\tau}_i$ is:

$$\delta_i^{sum} \leq \frac{m_{i,j} - \lceil f_{i,j} \rceil}{\lceil f_{i,j} \rceil}$$

Since, $\lceil f_{i,j} \rceil \geq 1$ and $\lceil f_{i,j} \rceil \geq f_{i,j}$, then,

$$\begin{aligned} \delta_i^{sum} &\leq \frac{m_{i,j} - 1}{f_{i,j}} \\ &\leq \frac{1}{f_i} && \leftarrow \quad \text{from Eq. (3.3)} \\ &\leq \frac{C_i}{D_i - L_i} \end{aligned}$$

The total density of the set $\widehat{\tau}$ is calculated as:

$$\delta^{sum}(\widehat{\tau}) \leq \sum_{\forall \widehat{\tau}_i \in \widehat{\tau}} \frac{C_i}{D_i - L_i}$$

which concludes the proof. □

Based on these two lemmas, we prove that the Seg-Str algorithm has the same resource augmentation bound as the DAG-Str algorithm when GEDF scheduling algorithm is used.

Theorem 3.6. *When the Seg-Str algorithm is applied, if a constrained-deadline task set $\widehat{\tau}$ of parallel threads is feasible on $\overline{m}$ unit-speed processors, with $n < \varphi \times \overline{m}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ (the golden ratio) and n is the number of DAGs in task set τ, then $\widehat{\tau}$ is schedulable using GEDF on $\overline{m}$ processors of speed at least $\frac{3+\sqrt{5}}{2}$.*

Proof. The resource augmentation bound of global EDF scheduling algorithm is based on its sufficient schedulability condition from Theorem 3.2. This condition depends on the number of processors of the system, on the maximum thread density $\delta^{max}(\widehat{\tau})$ and on the total density $\delta^{sum}(\widehat{\tau})$ of thread set $\widehat{\tau}$. Based on the Seg-Str algorithm, the number of processors $\overline{m}$ available for $\widehat{\tau}$ is the same as for the DAG-Str algorithm and it is shown in Equation (3.6).

Regarding the maximum thread density and the total density of the set, these values have the same upper bounds as the ones calculated in the case of the DAG-Str algorithm. By comparing the result of Lemma 3.4 with the DAG-Str upper bound from Equation (3.9), we conclude that both stretching algorithms have the same upper bound, since the maximum density of any thread in $\widehat{\tau}$ cannot exceed 1.

Similarly, both algorithms have an upper bound of the total density of the set which is equal to $\sum_{\forall \widehat{\tau}_i \in \widehat{\tau}} \frac{C_i}{D_i - L_i}$, as shown in Lemma 3.5 and Equation (3.11).

By using these upper bounds and by following the same proof of Theorem 3.3, the Seg-Str and the DAG-Str algorithms have the same resource augmentation bound, which concludes the proof. □

3.5 Simulation-Based Evaluation

In this section, we provide simulation-based evaluations for the Stretching algorithms used for DAG scheduling. We support our theoretical analysis of resource augmentation bounds of these algorithms by performing extensive simulations. We start by the DAG-Str algorithm and we compare its performance with the Decomposition algorithm[2] (DCMP) which belongs to the Model Transformation approach of DAG scheduling. The simulation results proved that the DAG-Str algorithm has better schedulability than the DCMP algorithm w.r.t. system utilization and to the number of processors. Then, we evaluate the schedulability performance of the Seg-Str algorithm w.r.t. the DAG-Str algorithm. The simulation results confirm our theoretical analysis, and both stretching algorithms have relatively similar schedulability success rates.

In order to perform the simulation experiments, we use *YARTISS*, our real-time simulation tool. The generation of DAG tasks is done randomly so as to guarantee the reliability of the experimental results. In Chapter 5, we explain in detail *YARTISS* simulator and its task generator classes. We use UUniFast-Discard algorithm [33] which fairly distributes the system utilization on its tasks, to be used afterward to compute their timing parameters (WCET, deadline and period). We also use the Hyper-period Limitation Technique [62] in the assignment of task periods so as to limit the length of their hyper period. In these simulations, we considered an execution interval of length equal to twice the length of the hyper period.

[2]For more details, please refer to Subsection 2.3.2.2 on page 46.

In this section, we considered global preemptive EDF scheduling algorithm which is used on execution platforms of multiple identical processors. In Chapter 5, we perform more experiments to evaluate the schedulability performance of our stretching algorithms compared to scheduling algorithms using the Direct Scheduling approach[3]. Furthermore, we show simulation-based evaluations of these algorithms when global Deadline Monotonic scheduling algorithm is used. This scheduling algorithm belongs to the Fixed Task Priority family while EDF belongs to the Fixed Job priority family.

3.5.1 DAG-Str Algorithm

In order to analyze the schedulability performance of the DAG-Str algorithm, we chose to compare it through simulation with the Decomposition (DCMP) algorithm. Both algorithms belong to the Model Transformation approach of DAG scheduling and they aim at avoiding the inter-subtask dependencies within DAG tasks by converting the model into independent sequential threads with intermediate timing parameters. As described earlier in Subsection 2.3.2.2, the DCMP algorithm distributes the slack time of a MTS task τ_i on its segments in a way that guarantees an upper bound of segment density equal to $\frac{2C_i}{D_i}$. Based on the slack time, the DCMP algorithm assigns intermediate offsets and deadlines to threads of each segment in the task.

In these simulation experiments, we consider execution platforms of m identical processors, where $m = \{2, 4, 8, 16\}$. We vary system utilization $U(\tau)$ as a percentage of number of processors m, where $U(\tau) = \{20\%, 40\%, ..., 100\%\}$ of m. For each system utilization, we generate 100,000 random DAG sets using *YARTISS* which are scheduled using GEDF algorithm.

DAG-Str Algorithm vs. DCMP Algorithm

The first simulation results are provided in Figure 3.5. They are obtained by varying two scheduling parameters; the number of unit-speed processors m on which DAG sets are executing, and the system utilization where the maximum utilization of a task set is equal to m. Figure 3.5 shows the results between the DAG-Str algorithm and the DCMP algorithm. For a given DAG set, both algorithms are applied first, then the scheduling simulation is performed on the generated threads, based on their intermediate timing parameters. It is worth noticing that the DCMP algorithm does not generate fully-stretched threads with utilization equal to 1, hence its

[3]The Direct Scheduling approach of DAG scheduling is explained in Chapter 4.

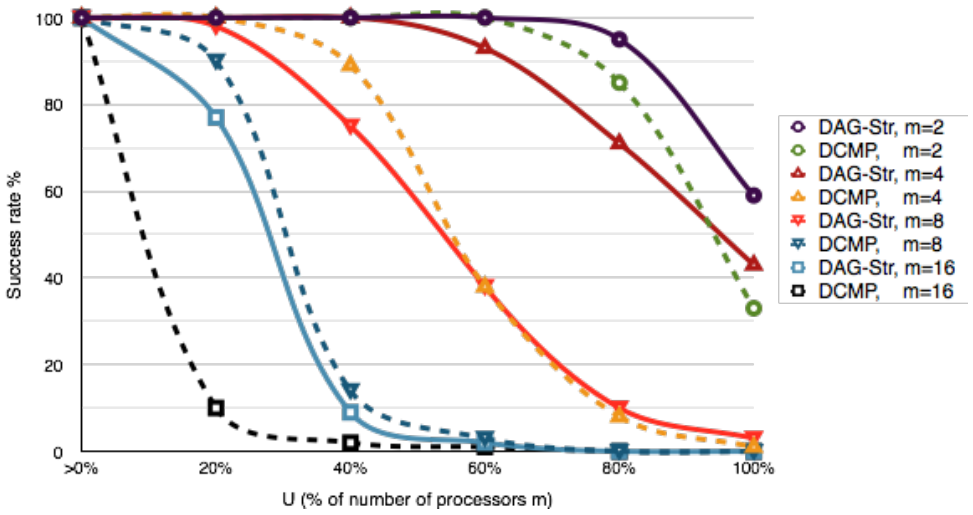


FIGURE 3.5: Comparison results of GEDF scheduling simulation between the DAG-Str algorithm and the DCMP algorithm.

thread set executes on all m processors of the system. While the DAG-Str algorithm executes on $\overline{m}$ processors (where $\overline{m}$ is less than or equal to m from Equation (3.6)). Figure 3.5 shows the success rate of schedulable DAG sets with GEDF on m processors (from 2 to 16). The x-axis of the figure represents the system utilization (its percentage w.r.t. the maximum utilization which is equal to m). In order to simplify the reading of results, the schedulability of DAG-Str algorithm is represented by solid curves in the figure and the schedulability of DCMP algorithm by dashed curves. Furthermore, every number of processors m is represented by a symbol for both scheduling algorithms. We chose a round symbol for $m = 2$, a triangle symbol for $m = 4$, an inverted triangle symbol for $m = 8$ and a square symbol for $m = 16$.

Based on the results of our simulations, we notice that the performance of DAG-Str algorithm is better than the DCMP algorithm for the scheduling of the DAG tasks, and the percentage of schedulable stretched task sets are higher than the decomposed ones. In general, the percentage of schedulable task sets decreases when the number of processors increases for both algorithms. For example, when $m = 2$, more than 90% of stretched task sets are schedulable when system utilization is equal to 80% and around 85% of decomposed task sets are schedulable for the same utilization. However, when $m = 16$, task sets with utilization higher than or equal to 40%, the schedulability percentage plummets. We can notice that the DAG-Str algorithm is less affected compared to the DCMP algorithm when the number of processors increases. The schedulability gap between both algorithms increases with the number of processors in the system.

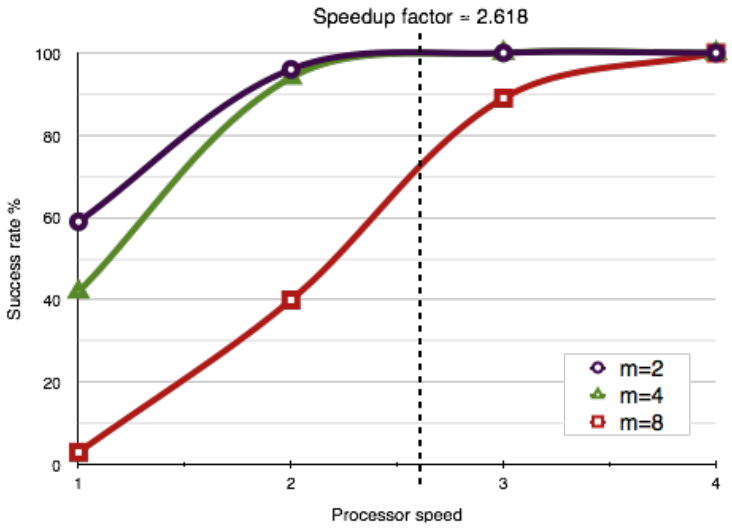


FIGURE 3.6: Simulation results show the effect of processor speed on the schedulability of the DAG-Str algorithm.

The effect of varying the speed of processors on DAG-Str schedulability

Here, we analyze the effect of increasing the speed of processors on the schedulability of stretched DAG tasks when DAG-Str algorithm is used. Based on the resource augmentation bound of DAG-Str, the schedulability of stretched DAG tasks increases as a function of the speed of processors, and feasible task sets become schedulable using GEDF when processor speed $\nu > \frac{3+\sqrt{5}}{2}$. In these experiments, we increased the speed of processors from $\nu = 1$ up to 4 by steps of 1, while varying the number of processors and system utilization as in the above simulations.

The simulation results are shown in Figure 3.6. For each m from 2 to 8, we simulate GEDF scheduling for task sets of maximum utilization equal to m. Based on simulation results from Figure 3.5, task set of such utilization have the least success schedulability rates, and this utilization value is considered as the worst case when processors are loaded. As shown in Figure 3.6, schedulability of stretched task sets increases when the speed of processors increases. In the case of $m = 2$ and $m = 4$, a success rate of almost 100% is achieved when the speed of processors $\nu \geq 2$, while we need a processor speed equal to 4 for $m = 8$. When $\nu \geq 4$, all task sets become schedulable when DAG-Str algorithm is used. The speedup factor of DAG-Str algorithm with GEDF is represented by a vertical line at $t = 2.618$ in the figure. After this time instant, the schedulability rate of $m \leq 4$ is equal to 100%. In the case of $m = 8$, the schedulability is more than 70%, which can be explained by the necessary utilization condition

($U(\tau) \leq m$) of DAG tasks. As to be shown later in Example 4.5 (on page 113), there exist DAG sets, whose utilization is equal to $U(\tau) = m$, that are not feasible on m unit-speed processors.

3.5.2 Seg-Str Algorithm vs. DAG-Str Algorithm

In this section, we compare the Seg-Str algorithm with the DAG-Str algorithm. In these simulations, we aim at applying our Seg-Str algorithm on DAG tasks and compare their schedulability with the DAG-Str algorithm while varying system utilization during simulation. We proved earlier in this chapter than both stretching algorithms have the same resource augmentation bound. Hereby, the simulation experiments can be an additional indication regarding the schedulability performance of these algorithms.

We considered 4 identical processors with task set utilization varies from 0.025 to 0.975 times the number of processors in steps of 0.025. For each utilization value we run 10,000 task sets each of 16 parallel tasks with implicit deadline which leads to 4,000,000 task sets in total.

Then, we performed scheduling simulations for GEDF algorithm. The scheduling results are shown in Table 3.1. We can notice, that both stretching algorithms are not comparable. In other words, Seg-Str may successfully schedule task sets that are not schedulable by DAG-Str algorithm, and vice versa. Also, the difference in the number of scheduled task sets between both algorithms is minor compared to the total number of generated task sets. For example, the Seg-Str schedules more task sets than the DAG-Str for all system utilization $U_i \leq 0.350$. Otherwise, the DAG-Str algorithm performs better. In general, the schedulability of DAG-Str algorithm is slightly better than the Seg-Str algorithm for more system utilization. These results show that both algorithms have the same schedulability rate and performance, this complies with the concept of the Seg-Str algorithm which is designed in the aim of reducing the scheduling overheads due to migrations and preemptions of jobs without affecting the schedulability success rate of the algorithm.

Regarding execution overheads due to migrations and preemptions, we did not perform experimental simulations in order to prove the out-performance of the Seg-Str algorithm over the DAG-Str algorithm. Since, the design concept of the Seg-Str algorithm aims at reducing the number of migrations. At most, one thread preemption was forced in the Seg-Str algorithm from each stretched DAG. While a thread from each segment of stretched DAGs are forced to preempt and migrate to master thread when the DAG-Str algorithm is applied.

U_i	DAG-Str	Seg-Str	U_i	DAG-Str	Seg-Str
0.050	10,000	10,000	0.550	9756	9741
0.100	10,000	10,000	0.600	9741	9724
0.150	9989	9997	0.650	9102	9071
0.200	9982	9990	0.700	8730	8716
0.250	9980	9988	0.750	8182	8160
0.300	9932	9949	0.800	7690	7615
0.350	9908	9913	0.850	6753	6719
0.400	9882	9880	0.900	5980	5883
0.450	9846	9832	0.950	5183	5126
0.500	9801	9794	1.000	4390	4389

TABLE 3.1: Scheduling comparison between DAG-Str algorithm and the Seg-Str algorithm.

3.6 Summary

In this chapter, we discussed the scheduling of parallel DAG tasks on multiprocessor systems using the Model Transformation approach. We started by proposing the DAG Stretching (DAG-Str) algorithm which is based on the Task Stretching (Task-Str) algorithm proposed by Lakshmanan et al. [77]. The DAG-Str is more general than the Task-Str since it considers DAG tasks instead of the parallel FJ tasks. We proposed to use the DAG-Str algorithm with the GEDF scheduling algorithm and we provided a resource augmentation bound as a performance metric.

Then, we proposed the Segment Stretching (Seg-Str) algorithm which is a modified version of the DAG-Str algorithm. The objective of this modification was to reduce the number of thread migrations and preemptions, when compared to the DAG-Str algorithm. Finally, the performance of the stretching algorithms had been evaluated using extensive simulations.

Chapter 4

Direct Scheduling Approach of Parallel DAG Tasks

In this chapter we discuss another scheduling approach for parallel DAG tasks which is called the Direct Scheduling. Unlike the Model Transformation approach[1], Direct Scheduling approach aims at scheduling parallel DAG tasks on multiprocessor systems using real-time scheduling algorithm directly on DAGs without modifying their model or their timing characteristics. The Model Transformation approach converts the parallel DAG model into a simplified independent sequential model to avoid the internal dependencies within the parallel tasks. This transformation facilitates the scheduling process of parallel tasks at the expense of generality loss of the DAG model.

In Direct Scheduling approach, the scheduling of parallel DAGs is done without any modification of the model, and the scheduler can be aware of the intra-task parallelism of such tasks and the precedence constraints that determine the execution order of their subtasks. Despite that regular multiprocessor scheduling algorithms such as EDF and FP are originally designed for independent sequential tasks and are not adapted for parallel tasks, they are still widely considered in many parallel scheduling researches such as [27, 36, 44, 81, 82][2]. However, if knowledge regarding internal execution structure of parallel tasks is not included in the scheduling process and analyses, the proposed schedulability conditions and performance bounds can be deemed pessimistic. To the best of our knowledge, recent researches on parallel tasks in general and on

[1] For more details, refer to Chapter 3 "Scheduling of Parallel Tasks using Model Transformation".
[2] For more details regarding the about related works, please refer to Section 2.3 on page 31.

DAG tasks in particular, concentrate mainly on the external structure of tasks rather than on their internal structure.

In this chapter, we are interested in analyzing the importance of the internal structure of parallel tasks when common scheduling algorithms are used to schedule them on multiprocessor systems. Moreover, we focus on global preemptive scheduling of sporadic constrained-deadline DAG tasks using EDF scheduling algorithm mainly (or any work conserving algorithm). We consider a DAG model of parallel tasks which assigns global timing parameters to DAGs and not to their subtasks. For example, each DAG task is assigned a relative deadline which is also used by its subtasks. One of our contributions is to provide DAG scheduling analyses which considers the execution order of subtasks due to the dependencies between them. In order to achieve this, we add extra local timing parameters to subtasks which are derived from the global parameters of their corresponding DAGs, so as to identify their execution order. In Section 4.1, we define the following local parameters for each subtask in the system: the earliest activation time (local offset), the latest possible finish time (local deadline) and release jitter. These local timing parameters are different from the intermediate parameters assigned by the stretching algorithms from the Model Transformation approach because local parameters tend to define the maximum execution interval of each subtask based on its precedence constraints and without imposing any external intermediate timing constraints.

Then in Section 4.2, we study the Direct Scheduling at DAG-Level, in which real-time algorithms take scheduling decisions based on global timing parameters of DAGs. Also in this section, we analyze GEDF schedulability condition from [81] of DAG tasks on multiprocessor systems, which was provided originally by considering the external structure of DAG tasks. In this section, we revise the schedulability analyses while taking into consideration the internal structure of DAGs and the precedence constraints between their subtasks. As a result, we provide a more adapted and tighter schedulability condition for DAG scheduling for any work conserving algorithm and GEDF. The scheduling analyses are based on identifying upper bounds on interference and workload of executing tasks.

In Section 4.3, we propose a Subtask-Level scheduling of DAGs in which real-time algorithms take scheduling decisions based on the assigned local timing parameters of subtasks rather than the global parameters of DAGs. To the best of our knowledge, Direct Scheduling at Subtask-Level is a new approach that has not been used before in the scheduling of DAG tasks on multiprocessor systems. As in the previous section, we provide interference and workload analyses for this scheduling. Then, we provide schedulability conditions for any work conserving

algorithm and GEDF. Furthermore, we argue the advantage of Subtask-Level scheduling on the feasibility analysis of DAG tasks by adapting the necessary feasibility condition of processor load which is more accurate for Subtask-Level scheduling rather than at DAG-Level.

Finally in Section 4.3.5, we provide simulation-based evaluations for the Direct Scheduling approach to compare it with other DAG scheduling algorithms from the state-of-the-art. Later in Chapter 5, we prove the incomparability of Direct scheduling approach at DAG-Level and at Subtask-Level, and we perform more experiments to analyze their performance. Section 4.4 concludes this chapter and summarizes our contributions.

4.1 Defining Extra Timing Parameters of DAG Tasks

Parallel real-time tasks of DAG model have particular characteristics due to precedence constraints between their subtasks and their execution dependencies. In addition to choosing which DAG job to execute on which processor at what time, the real-time scheduler has to choose the execution order of subtasks based on their precedence constraints. This is the reason why DAG scheduling on multiprocessor systems is much harder than the scheduling of independent sequential tasks. As described in Subsection 1.3.4, a DAG task[3] has inter-task parallelism and consists of a set of subtasks with precedence constraints to identify their execution flow. It is clear that the scheduling of DAG tasks is affected by dependencies between their subtasks and their internal structure.

We suppose that each DAG task τ_i is a sporadic constrained-deadline task which generates an unlimited number of jobs with minimum inter-arrival time T_i between successive jobs. Let J_i^k (respectively J_i) denote the k^{th} job of DAG τ_i (respectively any job of τ_i when its index is irrelevant). DAG job J_i^k is characterized by an absolute release time r_i^k (respectively r_i) and an absolute deadline d_i^k (respectively d_i). It is said to be **ready** at time t if $r_i \leq t$, while a subtask job $J_{i,j}$ is said to be **ready** at time t if its DAG job J_i is ready and all its predecessor subtasks have completed their execution at or before t. As explained earlier in Section 3.2, the default DAG model defines a single timing parameter for each subtask $\tau_{i,j} \in \tau_i$ which is its WCET $C_{i,j}$. The other timing parameters of a common real-time task model such as a relative deadline, a period and an offset, are inherited from its DAG τ_i and they are shared with all of its subtasks. In other words, subtasks of the same DAG have to respect the global deadline D_i and their job

[3]In this chapter, we use the same notation described in Section 3.2 from Chapter 3 "Scheduling of Parallel Tasks using Model Transformation".

invocations are released at least T_i time units apart. Hence, the scheduling of DAG tasks is more challenging than the scheduling of independent sequential tasks.

In recent researches [27, 81], a DAG task τ_i has been considered as a single entity characterized by its total WCET C_i (which is the sum of WCET of all of its subtasks), its deadline D_i, its period T_i and its critical path length L_i. This assumption avoids considering the internal structure and the dependencies between subtasks in order to simplify the scheduling process of DAGs. Although these external timing parameters are enough to provide scheduling analyses for DAG tasks, we show that ignoring internal structure and inter-subtask parallelism leads to pessimistic analytical results.

Example 4.1 (Importance of internal structure in DAGs schedulability)**.** *In Figure 4.1, we provide an example to show the importance of internal structure for the scheduling process and analysis of DAG tasks. In this example, we present two DAG tasks τ_1 and τ_2 which have identical external structure. Both DAG tasks have the same period and deadline ($T_1 = D_1 = T_2 = D_2 = 4$) and the same total WCET $C_1 = C_2 = 6$. Furthermore, both DAG tasks have the same critical path length ($L_1 = L_2 = 4$) which means that they have the same slack time ($Sl_1 = Sl_2 = 0$) as well. Hence, a real-time scheduler or scheduling analysis based on these global timing parameters will consider both DAGs as identical. This leads to a more pessimistic scheduling decisions and analysis.*

We show the internal structure of both DAGs in Figure 4.1. We consider that DAG τ_1 consists of 3 subtasks each of WCET equal to 2 time units. The source subtask is $\tau_{1,1}$ and its successors are subtasks $\tau_{1,2}$ and $\tau_{1,3}$ which execute in parallel. While DAG τ_2 consists of the same number and WCET of subtasks as in DAG τ_1, but it has two source subtasks $\tau_{2,1}$ and $\tau_{2,2}$ and their successor is subtask $\tau_{2,3}$.

In this example, we compare quantity of work required by subtasks within a given time interval between the external graph parameters to the internal ones. First, if we consider that jobs from each DAG are activated at time t, and a study interval is defined as $[t, t+4)$, then both DAGs perform 6 time units each, whether we consider external or internal structure of DAGs, since the length of the interval is equal to their relative deadline. However, if we reduce the study interval to $[t+2, t+4)$, then it is impossible to identify the exact amount of executed work in the case of timing parameters based on external structure of DAGs. However, if we consider the internal structure of DAGs, we can identify the performed work in the reduced interval more accurately. In the case of DAG τ_1, it is executed for 4 time units in interval $[t+2, t+4)$, since subtasks

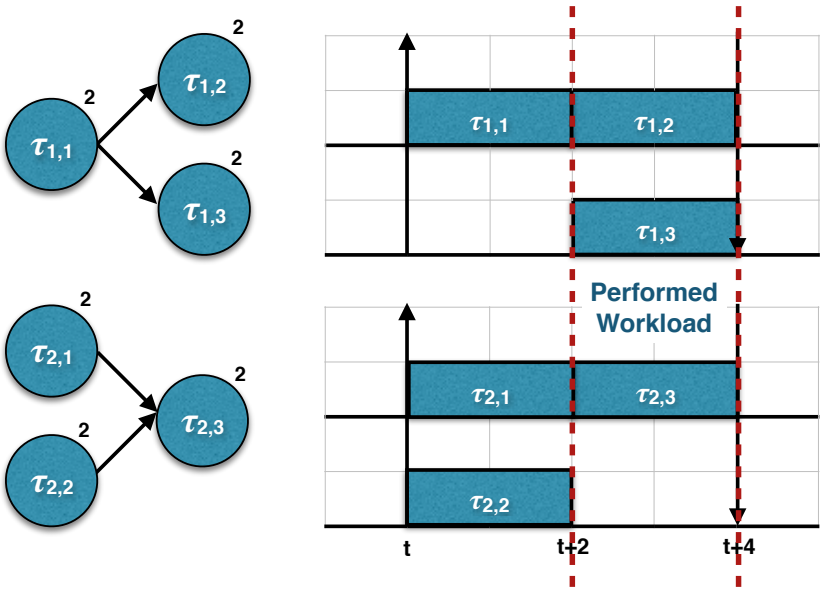


FIGURE 4.1: An example showing the importance of internal structure for scheduling DAGs with identical external structure

$\tau_{1,2}$ and $\tau_{1,3}$ are active in this interval. While DAG τ_2 performs only 2 time units within the same study interval, because subtask $\tau_{2,3}$ is the only active subtask in this time interval and it is impossible for its predecessors to execute in this interval without causing a deadline miss.

From this example, we conclude that the use of internal structure of DAGs leads to more accurate scheduling decisions and analysis, due to the added knowledge about the execution flow of subtasks within DAG tasks and their precedence constraints. However, to use such knowledge in real-time scheduling, it is necessary to define extra local timing parameters for subtasks to express their precedence constraints. There is a difference between these local parameters and the intermediate parameters assigned by the Model Transformation algorithms such as the DAG-Str and Seg-Str algorithm (from Chapter 3). In the latter case, the intermediate parameters are imposed on subtasks and they are used to modify the task model to get rid of its internal dependencies and insure independent execution. However, the local parameters from the Direct Scheduling approach are used to represent the maximum execution interval of each subtask based on its precedence constraints. Oppositely to the intermediate timing parameters assigned to subtasks by the Model Transformation approach.

In the remainder of this section, we analyze the structure of DAG tasks and their subtasks. We start by defining two main timing parameters for subtasks which are a local offset and a relative deadline. These two parameters define the maximum execution interval of each subtask w.r.t. the global parameters of its DAG. Then, we define a third timing parameter for each

subtask which is called the maximum release jitter to determine the activation interval within the execution interval of the subtask, i.e., the maximum delay of subtask activation based on the finish time of its predecessor subtasks.

4.1.1 Local Offset and Deadline for Subtasks

According to the DAG model, subtasks are characterized by their WCET and their execution depends on the precedence constraints of their DAGs. A subtask is allowed to execute when all of its predecessor subtasks have completed their own execution. Based on these characteristics, we provide the following definitions:

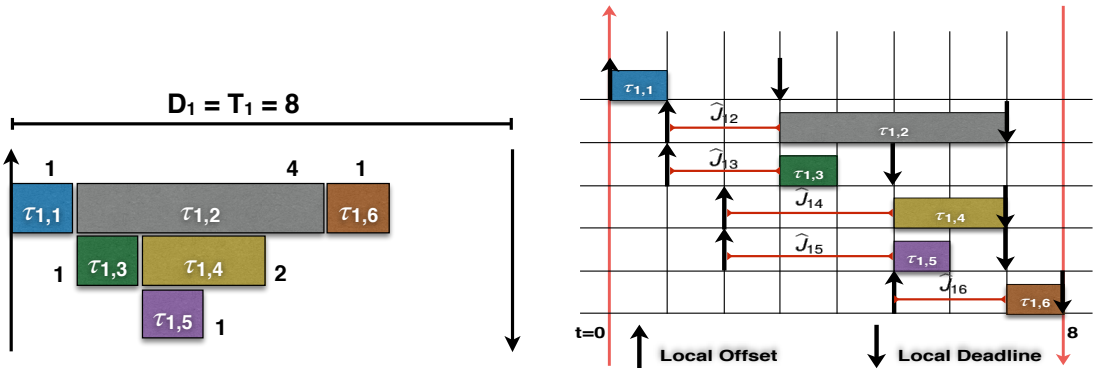
Definition 4.1. A **local offset** $O_{i,j}$ of subtask $\tau_{i,j} \in \tau_i$ is defined as the earliest possible release time of the subtask at which it becomes ready relative to the activation time of its DAG τ_i and based on the WCET of its predecessor subtasks.

Additionally, the local offset $O_{i,j}$ represents the length of the longest execution path from a source subtask in DAG τ_i to subtask $\tau_{i,j}$, when all of the subtasks in this path are activated as soon as possible and execute up to their WCET. Based on this definition, subtask $\tau_{i,j}$ cannot be ready at any time t, where $O_i \leq t < O_{i,j}$, since its predecessor subtasks have not finished their execution in this time interval when we consider their WCETs. Source subtasks of a DAG task have no local offsets since they have no predecessor subtasks and they are activated directly after the activation of their DAG task.

In order to calculate the local offset of a subtask, we consider that its DAG does not suffer from any external or internal blocking events which may delay the execution of its subtasks. The only accepted source of blocking in this calculation is the delay due to the execution of predecessor subtasks. Hence, we consider for the analysis the case where the DAG task executes on a system of unlimited number of processors, where all subtasks are activated as soon as possible. In that case, the response time of the DAG task is equal to its critical path length. For DAG task τ_i, the local offset $O_{i,j}$ of subtask $\tau_{i,j} \in \tau_i$ is represented by the following equation:

$$O_{i,j} = \max_{\forall \tau_{i,k} \in Parents(\tau_{i,j})} (O_{i,k} + C_{i,k}) \tag{4.1}$$

where all source subtasks of DAG τ_i have no local offsets (equal to 0).



(a) A DAG task τ_1 consists of 6 subtasks and their precedence constraints.

(b) The local timing parameters of each subtask in τ_1.

FIGURE 4.2: An example showing the local timing parameters of subtasks in a DAG task.

We apply a straightforward depth-first search algorithm (Algorithm 4.1) to calculate the local offset of each subtask of all DAGs in the set. The algorithm executes in linear time.

Algorithm 4.1 Local offset algorithm

Input: $\tau_i = (n_i, \{1 \leq j \leq n_i | \tau_{i,j}\}, G_i, D_i)$
Output: $O_{i,j} : \forall \tau_{i,j} \in \tau_i$ ▷ $O_{i,j}$ is the local offset of subtask $\tau_{i,j}$

```
procedure LOCAL_OFFSET(τ_{i,j})
    if τ_{i,j} == isSourceSubtask(τ_i) then
        return O_i
    else
        return max_{τ_{i,k} ∈ Parents(τ_{i,j})} (LOCAL_OFFSET(τ_{i,k}) + C_{i,k})
    end if
end procedure
```

Example 4.2 (Local offset assignment of subtasks)**.** *In Figure 4.2, we represent an implicit-deadline DAG task τ_1 with $D_1 = T_1 = 8$. It consists of 6 subtasks with a total WCET $C_1 = 10$. In Figure 4.2(a) we show the internal structure of the DAG task where each subtask is displayed as a square associated with a number which denotes its WCET. We consider that DAG τ_1 has a single source subtask $\tau_{1,1}$ which is activated at the same activation time of τ_1. In this example, we consider a time reference $t = 0$, hence, the local offset of the source subtask is $O_{1,1} = 0$.*

Then, we calculate the local offset of each subtask $\tau_{1,j}$, where $1 < j \leq 6$. Starting by the source subtask $\tau_{1,1}$, its children subtasks $\tau_{1,2}$ and $\tau_{1,3}$ are activated when $\tau_{1,1}$ finishes its execution. Their earliest possible activation time is equal to the WCET of $\tau_{1,1}$ which is $O_{1,2} = O_{1,3} = C_{1,1} = 1$. If we suppose that the DAG is released at time t, then it is impossible for these subtasks to be released earlier than $(t + 1)$[4]. Similarly, we calculate the local offset of subtasks $\tau_{1,4}$ and $\tau_{1,5}$ which are the children of subtask $\tau_{1,3}$. They have to wait for the completion of all predecessor

[4]In this thesis, we consider that jobs execute up to their WCET at all times.

subtasks ($\tau_{1,1}$ *and* $\tau_{1,3}$) *before they are released. Hence,* $O_{1,4} = O_{1,5} = O_{1,3} + C_{1,3} = 2$ (*from Equation* (4.1)).

Finally, subtask $\tau_{1,6}$ *has* 3 *parent subtasks* $\{\tau_{1,2}, \tau_{1,4}, \tau_{1,5}\}$*, and it has to wait for their completion before it can start its own. Based on Equation* (4.1)*, the local offset of subtask* $\tau_{1,6}$ *is calculated as:* $O_{1,6} = \max((1+4), (2+2), (2+1)) = 5$*, which represents the longest execution path from* $\tau_{1,1}$ *(included) to* $\tau_{1,6}$ *(excluded).*

We now define another important timing parameter for subtasks which is their local deadline.

Definition 4.2. A **local deadline** $D_{i,j}$ of subtask $\tau_{i,j}$ is defined as the latest possible finish time of $\tau_{i,j}$ after its release, which takes into consideration the amount of work to be executed by its successor subtasks when their WCETs are considered.

For each subtask $\tau_{i,j}$ of DAG τ_i, we calculate a local deadline $D_{i,j}$ by considering the longest sequential execution path starting from its successors to any sink subtask in the DAG, where each subtask in the path executes up to its WCET. As in the case of local offset, the local deadline is calculated while considering the best execution scenario of subtasks in which we assume that the system has an unlimited number of processors. In order to respect the global deadline of DAG τ_i, each subtask $\tau_{i,j}$ should leave enough time for its successors. The local deadline $D_{i,j}$ of subtask $\tau_{i,j}$ is calculated as follows:

$$D_{i,j} = \min_{\forall \tau_{i,k} \in Children(\tau_{i,j})} ((D_{i,k} + O_{i,k}) - C_{i,k} - O_{i,j}) \tag{4.2}$$

where any sink subtask $\tau_{i,j}$ of DAG τ_i has a local deadline equal to $(D_i - O_{i,j})$.

It is worth mentioning that the subtask's local deadline is a relative value and the maximum execution interval of subtask $\tau_{i,j}$ is defined as $[O_{i,j}, O_{i,j} + D_{i,j})$.

Observation 4.1. If subtask $\tau_{i,j}$ misses its deadline $r_{i,j} + D_{i,j}$ at time t (where $t < r_i + D_i$), then DAG τ_i will definitely miss its deadline as well, when subtasks execute up to their WCETs.

Proof. In order to calculate the local deadline of subtasks, we considered the scenario where the system has unlimited number of processors which means that subtasks do not suffer from any external blocking events. The only source of blocking is due to execution dependencies between subtasks of the same DAG. The local deadline of a subtask is defined as its latest possible finish time, which leaves enough time for the execution of its successors. So, missing this deadline

at some time $t < r_i + D_i$, means that the length of time interval $[t, r_i + D_i)$ is not enough for successor subtasks to execute even if the system has an unlimited number of processors, which leads to a deadline miss of the DAG. Based on this observation, an early deadline miss can be declared based on local deadlines of subtasks rather than waiting for the deadline miss of the DAG. □

In Algorithm 4.2, we show a straightforward recursive method based on the depth-first search algorithm, that calculates the local deadline of each subtask in the DAG. This algorithm executes in linear time.

Example 4.3 (Local deadline assignment of subtasks)**.** *Back to the example in Figure 4.2, we calculate the local deadline $D_{1,j}$ for each subtask $\tau_{1,j}$ in DAG τ_1. We start by the sink subtask $\tau_{1,6}$ which has no successor subtask, so it can finish its execution at DAG deadline, and its local relative deadline is equal to $D_{1,6} = (D_1 - O_{1,6}) = 3$. Its predecessor subtasks are $\{\tau_{1,2}, \tau_{1,4}, \tau_{1,5}\}$. These subtasks must have local relative deadlines that guarantee the schedulability of $\tau_{1,6}$ in the best case, and they have to finish their execution at time instant no later than $((D_{1,6} + O_{1,6}) - C_{1,6} - O_{1,j})$, where $j \in \{2, 4, 5\}$. Hence, their local deadlines are equal to $D_{1,2} = 6$, $D_{1,4} = D_{1,5} = 5$.*

Regarding subtask $\tau_{1,3}$, it has two children subtasks ($\tau_{1,4}$ and $\tau_{1,5}$). Similarly, local deadline $D_{1,3}$ is calculated as in Equation (4.2) where $D_{1,3} = min((D_{1,4} + O_{1,4} - C_{1,4} - O_{1,3}), (D_{1,5} + O_{1,5} - C_{1,5} - O_{1,3})) = min(4, 5) = 4$. Same calculations are applied to the remaining subtask $\tau_{1,1}$ which has two children subtasks $\tau_{1,2}$ and $\tau_{1,3}$, and its local deadline $D_{1,1}$ is equal to 3.

The timing characteristics of a given DAG task are enriched by including local offsets and deadlines to its subtasks. A constrained-deadline subtask $\tau_{i,j} \in \tau_i$ is characterized by $\{O_{i,j}, C_{i,j}, D_{i,j}, T_{i,j}\}$. Where $T_{i,j}$ is the period of the subtask which is equal to the period T_i of its DAG task τ_i.

Algorithm 4.2 Local deadline algorithm

Input: $\tau_i == (n_i, \{1 \leq j \leq n_i | \tau_{i,j}\}, G_i, D_i)$ ▷ Inputs: τ_i is a graph task, $\tau_{i,j}$ is a subtask in τ_i
Output: $D_{i,j} : \forall \tau_{i,j} \in \tau_i$

```
procedure LOCAL_DEADLINE(τ_{i,j})
    if τ_{i,j} = isSinkSubtask(τ_i) then
        return D_i − O_{i,j}
    else
        return min_{τ_{i,k} ∈ Children(τ_{i,j})} ( LOCAL_DEADLINE(τ_{i,k}) + O_{i,k} - C_{i,k}- O_{i,j})
    end if
end procedure
```

As stated earlier and from the definition of subtask parameters, the local offsets and deadlines assigned to subtasks are different from the intermediate offsets and deadlines assigned to subtasks using the algorithms from the Model Transformation approach. The latter parameters are imposed so as to alter the task model and get rid of dependencies. While the local parameters are used to identify the maximum execution interval of each subtask and to include knowledge about the internal structure of DAGs for the scheduling process.

4.1.2 Local Release Jitter of Subtasks

Since we considered that local offsets and deadlines of subtasks are calculated based on the best case execution scenario, in which all subtasks execute as soon as they are released with no interference or delays, the maximum execution interval of each subtask $\tau_{i,j} \in \tau_i$ is defined as $[O_{i,j}, O_{i,j} + D_{i,j})$. However, the actual activation time of subtasks still depends on the finish time of their predecessors, and subtasks have to respect the precedence constraints defined by their DAG. For a given subtask $\tau_{i,j}$ in DAG task τ_i, it is released after $O_{i,j}$ time units from the release time of τ_i, but if the execution of at least one predecessor subtask is delayed due to an interference from higher priority tasks in the set, then the activation of subtask $\tau_{i,j}$ will be delayed until all predecessors finish their execution.

As a result, the actual activation time of a subtask $\tau_{i,j}$ is dynamic and it reduces the length of its maximum execution interval. The length of the activation interval is denoted as the maximum release jitter $\widehat{j}_{i,j}$ and it is defined as follows:

Definition 4.3. A maximum release jitter $\widehat{j}_{i,j}$ of subtask $\tau_{i,j}$ represents the length of the maximum activation interval of $\tau_{i,j}$ in which its job can be activated at any time instant.

The maximum release jitter of a subtask is characterized as the maximum difference between the earliest activation time (local offset) of a subtask and the latest finish time (local deadline) of each of its predecessors.

$$\widehat{j}_{i,j} = \max_{\forall \tau_{i,k} \in Parents(\tau_{i,j})} (D_{i,k} - (O_{i,j} - O_{i,k})) \tag{4.3}$$

Based on Equation (4.3), we can identify a maximum activation interval for each subtask in the DAG. Hence, the interval of release time $r_{i,j}$ of job $J_{i,j}$ of subtask $\tau_{i,j}$ is defined as follows:

$$\forall J_{i,j} \ , \ \tau_{i,j} \in \tau_i \ : \ r_{i,j} \in [O_{i,j}, O_{i,j} + \widehat{j}_{i,j}]$$

It is worth noticing that the source subtasks of a DAG task does not have maximum release jitters since they are activated by the activation of their DAG task and they have no predecessors.

Example 4.4 (Maximum release jitter of subtasks)**.** *Based on the example of Figure 4.2, Figure 4.2(b) shows the maximum release jitter* $\widehat{j}_{1,j}$ *of each subtask* $\tau_{1,j}$ *in DAG task* τ_1*. Starting by the source subtask* $\tau_{1,1}$*, it is clear that it has no release jitter and* $\widehat{j}_{1,1} = 0$*. Regarding its children subtasks* ($\tau_{1,2}$ *and* $\tau_{1,3}$)*, their maximum release jitter is calculated as* $\widehat{j}_{1,2} = (D_{1,1} - (O_{1,2} - O_{1,1})) = 2$*, for subtask* $\tau_{1,2}$*, and* $\widehat{j}_{1,3} = \widehat{j}_{1,2}$ *for* $\tau_{1,3}$*.*

The same calculations are applied to the other subtasks in the DAG. The release jitter of subtasks $\tau_{1,4}$ *and* $\tau_{1,5}$ *is equal to* $\widehat{j}_{1,4} = \widehat{j}_{1,5} = 3$*, and* $\widehat{j}_{1,6} = 2$ *for subtask* $\tau_{1,6}$*.*

From Equation (4.3) and the example in Figure 4.2(b), we notice that the calculations of the maximum release jitter of subtasks are pessimistic, since we consider that predecessor subtasks execute as late as possible. The worst case execution scenario happens if we consider that critical subtasks of any DAG (subtasks forming its critical path) are activated at the end of their activation interval, hence, they will have no slack time. In the following sections, we provide an optimization of the release jitter of subtasks based on the interference concept accounted for in the schedulability analysis.

After defining the maximum release jitter, subtasks of a given DAG task are characterized by a local offset, a WCET, a local deadline, a period and a maximum release jitter. This is an enhancement to the original DAG task model which characterizes subtasks by their WCET only. In the following sections, we use these local timing parameters so as to define and analyze the Direct Scheduling approach at DAG-Level and at Subtask-Level.

4.2 Scheduling DAGs using Global Parameters (DAG-Level Scheduling)

Scheduling DAG tasks based on their global timing parameters (DAG-Level Scheduling) is categorized as a Direct Scheduling approach of DAGs. According to DAG-Level scheduling, DAGs are scheduled using regular real-time multiprocessor algorithms, while scheduling decisions are

taken based on the global timing parameters of the DAGs, such as their period, deadline or slack time. Accordingly, the internal structure of DAGs is not included in the decision-making process, and we assume that subtasks inherit the priority of their DAG.

For example, in the case of DM algorithm from the FTP priority-assignment scheme, the priority of DAG tasks is assigned based on their global relative deadline according to the task model, where the DAG with the smallest relative deadline has the highest priority. Similarly in the case of EDF algorithm, the DAG job, whose absolute deadline is the earliest, has the highest priority.

Recently, the DAG-Level scheduling approach has been used in many research concerning DAG scheduling (see Chapter (2) "Related Work"). However, we noticed that the schedulability analyses are done at DAG level, based on the global parameters of DAGs. In this section, we show the importance of internal structure of DAGs on the schedulability analysis of DAG-Level approach. This is achieved by analyzing a DAG-Level schedulability condition of GEDF scheduling algorithm from [81]. Then, we revisit the analysis by including extra information regarding the internal structure of DAGs leading to an optimized schedulability condition. In Section 4.3.5, we compare both conditions using simulation and we show how our condition outperforms the one from [81].

As we can notice from the schedulability bounds and tests from [81], no knowledge of internal structure of DAGs is required, the analysis depends on the global parameters of DAGs such as its utilization and global deadline. Lemma 2.6 (on page 48) provides an upper bound on the workload which is performed by a particular job of a DAG task in an interval equal to its deadline, on a system of m processors of speed b. We show that the schedulability condition in this lemma can be optimized if the knowledge regarding the internal structure of DAG is added.

In this section, we consider global EDF scheduling of sporadic DAGs on multiprocessor systems at DAG-Level. We are interested in the schedulability analysis of these systems when internal structure of DAGs is considered. Moreover, we provide interference and workload analyses of DAG task scheduling. For a given DAG job, we aim at quantifying the total workload done during an execution interval of length equal to its deadline. Usually, the performed workload consists of the execution time of this job and the performed work from interfering jobs of higher priorities in the set. In order to find an upper bound of workload, we identify the scenario which generates the highest interference on a particular job, then we analyze the possible interference on DAG tasks w.r.t. this scenario.

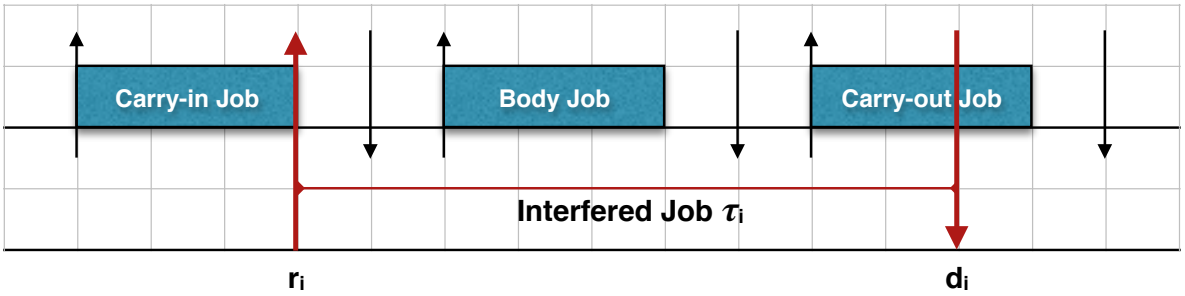


FIGURE 4.3: The different types of interfering jobs (carry-in, body and carry-out jobs).

4.2.1 Interference Analysis on DAGs

Based on the DAG model, each task τ_i generates an unlimited number of jobs and each job is denoted by J_i. Since we consider constrained-deadline DAGs in which the global deadline of a DAG task cannot exceed its period, there is at most one active job from each DAG at any time instant t. As a result, when a scheduling algorithm is used at DAG-Level, then interference on job J_i is caused by higher priority jobs from other DAGs, and it is impossible for a precedent job of DAG τ_i to participate in the interference.

For a given job J_i of DAG τ_i which is active in a fixed interval $[r_i, d_i)$, its execution can be interrupted by three types of interfering jobs generated by the higher priority DAGs (as shown in Figure 4.3):

- A body job is defined as any job that is released after the release time r_i of J_i and has an absolute deadline no later than the absolute deadline d_i of J_i. The interfering body job executes completely within the interval $[r_i, d_i)$.
- A carry-in job is a job that is released before r_i while its absolute deadline is in the interval $(r_i, d_i]$.
- A carry-out interfering job is defined as the job whose release time is in the interval $[r_i, d_i)$ and its absolute deadline is later than d_i.

It is worth noticing that there is at most one carry-in and carry-out jobs from each interfering constrained-deadline DAG task, while there is no limitation on the number of body jobs (it might be zero or more jobs within the interference interval based on its length w.r.t. the activation interval of the interfering job).

In the case of EDF scheduling algorithm, in which priorities are assigned to jobs based on their absolute deadlines, carry-out jobs have later absolute deadlines than interfering job J_i (from

definition) and they are assigned lower priority by the scheduling algorithm. Hence, they cannot interrupt the execution of J_i. Only carry-in and body jobs are considered in the analysis of this scheduling algorithm.

4.2.1.1 The Worst Case Interference Scenario for DAG tasks

In Lemma 2.6 (on page 48), an upper bound on the total workload A_k^a performed on job J_k^a of DAG τ_k is calculated. Then it is used to identify the minimum processor speed b that guarantees the GEDF schedulability of the task set. In this section, we optimize the computation of the workload while considering the internal structure of the interfering jobs. In order to identify an upper bound on workload, Bertogna et al. [32] characterized the worst-case job activation scenario that generates the maximum interference. The **maximum interference** on a DAG task τ_k is defined as the longest cumulative time intervals in D_k in which any subtask of τ_k is ready to execute but blocked by higher priority DAG tasks in the system.

Observation 4.2. For any sporadic DAG task, the maximum total interference on a particular job of DAG τ_k in a time interval of length D_k occurs when the jobs of other interfering DAGs are activated periodically.

Proof. The proof of this observation is straightforward. The interference of a DAG task τ_i on a particular job of another DAG τ_k depends on the number of jobs that are activated within the interference interval. Sporadic activation of real-time task τ_i means that successive jobs are separated by time intervals of length greater than or equal to T_i. In order to maximize the number of jobs within a fixed time interval, we consider the minimum separation time of successive jobs, which is after T_i time units and this is the definition of periodic real-time tasks. □

Therefore, to define the worst-case interference scenario on a particular task job from a sporadic task set, we consider periodic activation of tasks in the set during scheduling analysis.

Lemma 4.4 (Extended from [32])**.** *When GEDF scheduling algorithm successfully schedules a DAG set, the interference of a DAG task τ_i on a particular job of τ_k in a time interval of length D_k is maximized when the deadline of the last body job of τ_i is the same as the deadline of τ_k, and the carry-in job of τ_i executes just before its deadline.*

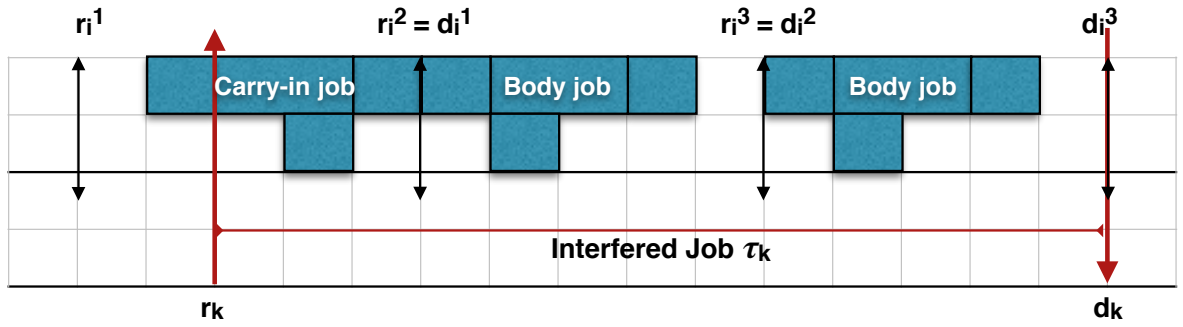


FIGURE 4.4: The worst-case interference scenario of an interfering DAG task τ_i on a job of DAG τ_k when GEDF scheduling algorithm is used.

Proof. The worst-case interference scenario for sequential tasks is described in [32]. It has been proved that this activation scenario of jobs generates the maximum workload in the time interval, as shown in Figure 4.4. By moving the interval backward or forward, the performed workload can only decrease. In this lemma, the worst-case interference scenario is extended to the parallel DAG model. Since we consider a DAG-Level scheduling in which scheduling decisions are taken based on the global parameters of the DAGs. Hence, the scenario described in [32] applies on DAG tasks.

Figure 4.4 shows the worst-case interference scenario generated by the jobs of interfering DAG τ_i on a particular job of DAG τ_k. The length of the interference interval $[r_k, d_k)$ is equal to DAG's deadline D_k. Hence, the interference bound can be analyzed while considering this activation scenario. As explained in [32], all interfering jobs in the interval have higher priorities than DAG τ_k when GEDF scheduling algorithm is used. All body jobs execute completely within the interference interval, while the carry-in job is forced to execute just before its deadline so as to include the maximum amount of its execution time in the interference. Shifting the interval backward or forward leads to a decrease of job interference. □

Let J_k^* denote the job of DAG τ_k which is defined in the worst-case interference scenario from Lemma 4.4 (and Figure 4.4). Job $J_{k,j}^*$ is the j^{th} subtask job from J_k^*. Regarding the carry-in job from the scenario described in Lemma 4.4. If a job of DAG τ_i executes just before its absolute deadline d_i (as in the case of carry-in jobs), then each subtask $\tau_{i,j} \in \tau_i$ will execute just before its calculated local absolute deadline $d_{i,j}$. In the remainder of this section, we explain the different types of interfering jobs in more detail and their performed workload. Then, we derive a GEDF schedulability condition for DAG tasks which is aware of their internal structure.

Body Job Interference

Based on the definition of an interfering body job, a DAG job J_i executes completely within the interference interval and it contributes with its total execution time C_i to the interference on J_k^*. All subtasks of DAG τ_i execute within the interval, and the internal structure of the DAG does not affect the total performed interference. Hence, the total interference of body DAG jobs can be calculated using the Demand Bound Function (DBF) (from Definition 1.2 on page 5).

Lemma 4.5. *The total body work on J_k^* of DAG task τ_k is the sum of the demand bound function of all the DAGs in task set in a time interval of length D_k.*

Proof. All subtasks of a body job execute completely in the interference interval and they contribute in the body workload. Their precedence constraints and execution order have no effect on the amount of the total workload. Also, the DAG's body jobs are defined as the jobs whose arrival time and deadline are within the interference interval. As a result, we can use the DBF to calculate the maximum workload done in this interval.

Let DBF_k^i denote the DBF of a DAG task τ_i on an interval equal to the relative deadline D_k of DAG task τ_k, which is calculated as follows:

$$DBF_k^i = \left(\left\lfloor \frac{D_k - D_i}{T_i} \right\rfloor + 1 \right)^+ \times \sum_{j=1}^{n_i} C_{i,j}$$

where $(x)^+ = \max(0, x)$.

The total workload of body jobs from all n interfering DAGs in τ, on a job of DAG τ_k is calculated as follows:

$$DBF_k = \sum_{k=1}^{n} DBF_k^i$$

□

Carry-in Job Interference

As stated earlier, the carry-in DAG job J_i on a job J_k from another DAG τ_k is defined as the job whose release time is earlier than the release of J_k and has an absolute deadline d_i in the interval $[r_k, d_k)$. For constrained-deadline tasks, there is at most one carry-in job from each interfering DAG in the task set.

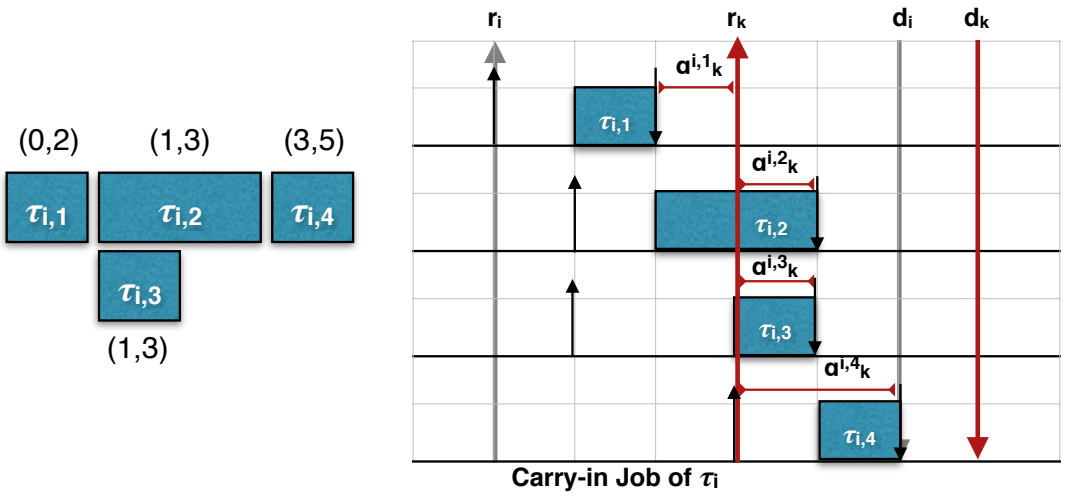


FIGURE 4.5: Carry-in interference of subtasks of DAG τ_i on DAG τ_k when GEDF is used. The local parameters of each subtask $\tau_{i,j}$ are represented as $(O_{i,j}, D_{i,j})$.

Let α_k^i denote the length of the interference interval of carry-in DAG job J_i on DAG job J_k, i.e., the length of the overlapping interval of parallel execution of both jobs. It is the interval between the release time of job J_k and the absolute deadline of the interfering job J_i, where $\alpha_k^i = d_i - r_k$. For independent sequential tasks, the carry-in interference can be easily identified, but it is more complicated in the case of parallel DAG tasks. As shown in the example from Figure 4.1 (on page 91), which explains the importance of the internal structure of DAG tasks in the schedulability analysis, the execution order of subtasks and their precedence constraints can be used to determine which subtasks to include in the carry-in interference. We can use the assigned local timing parameters of subtasks (offsets and deadlines) to determine which of them is active within the carry-in interference interval.

Let $\alpha_k^{i,j}$ denote the carry-in interference interval of subtask $\tau_{i,j} \in \tau_i$ on job J_k. It is defined as: $\alpha_k^{i,j} = d_{i,j} - r_k$, where $d_{i,j}$ is the absolute local deadline of subtask $\tau_{i,j}$. Figure 4.5 shows an example of a DAG task τ_i which has a deadline and period equal to 5 and consists of 4 subtasks. The figure shows the carry-in interference interval of each subtask in the job J_i when it interferes with another DAG job J_k (its release time r_k and absolute deadline d_k are shown in the figure). According to the example, subtask jobs of the carry-in job J_i are divided into the following categories based on their interference:

- If subtask $\tau_{i,j}$ has an absolute local deadline $d_{i,j}$ earlier than the release time of J_k, then it causes no carry-in interference, and $\alpha_k^{i,j}$ has a negative value.
- If subtask $\tau_{i,j}$ has a local offset which is later than the release time of job J_k, then it contributes with its total execution time in the carry-in interference.

- If subtask $\tau_{i,j}$ is released earlier than the release time of J_k and it has an absolute deadline in the interval of $[r_i, r_i + \alpha_k^i)$, then it has a partial carry-in interference on job J_k which is equal to $\alpha_k^{i,j}$.

Let ζ_k^i denote the maximum carry-in interference of a job from DAG τ_i on another job from DAG τ_k. We have:

$$\zeta_k^i = \sum_{\forall \tau_{i,j} \in \tau_i} \min\left(C_{i,j}, \max\left(0, \alpha_k^{i,j}\right)\right) \tag{4.4}$$

Lemma 4.6. *When GEDF algorithm successfully schedules a set τ of n DAG tasks, the maximum carry-in interference ζ_k on job J_k^* from other DAGs in an interval of length equal to D_k is at most:*

$$\zeta_k = \sum_{i=1}^{n} \sum_{j=1}^{n_i} \min\left(C_{i,j}, \max\left(0, \alpha_k^{i,j}\right)\right) \tag{4.5}$$

Proof. The proof is based on the worst-case interference scenario from Lemma 4.4, in which the worst carry-in interference of a DAG job on another happens when the carry-in interfering job executes just before its absolute deadline. In this situation, each subtask job $J_{i,j}$ of the carry-in job J_i cannot perform more than its WCET that executes within the interference interval. Hence, it cannot interfere with job J_k^* within a time interval of length more than $\alpha_k^{i,j}$ time units. Since each subtask $\tau_{i,j} \in \tau_i \in \tau$ is a sequential thread, its maximum workload is upper bounded by its WCET $C_{i,j}$. □

Schedulability Condition of GEDF based on DAG's Internal Structure

Based on the interference analysis of carry-in and body jobs, we derive the following theorem:

Theorem 4.7. *A set τ of n DAG tasks is GEDF schedulable on m processors of speed b if:*

$$\begin{aligned} &\forall \tau_k \in \tau, \\ &\sum_{i=1}^{n} DBF_k^i + \sum_{i=1, i \neq k}^{n} \zeta_k^i \leq bmD_k - (m-1)D_k \end{aligned} \tag{4.6}$$

where ζ_k^i is defined in Equation (4.4).

Proof. Lemma 2.6[81] (on page 48) provides an upper bound on the total workload A_k^a of interfering DAGs on job J_k^a in a time interval of length equal to D_k. In our interference analysis, which is based on the internal structure of DAG tasks, we consider the total workload of interfering jobs as the sum of body workload from Lemma 4.5 and carry-in workload from Lemma 4.6. □

Based on this theorem and for each DAG task in the set, we find the minimum speed b of processors that satisfies Equation (4.6). Then for all DAGs of the set, we consider the maximum value of b that guarantees the schedulability of the task set for GEDF scheduling algorithm.

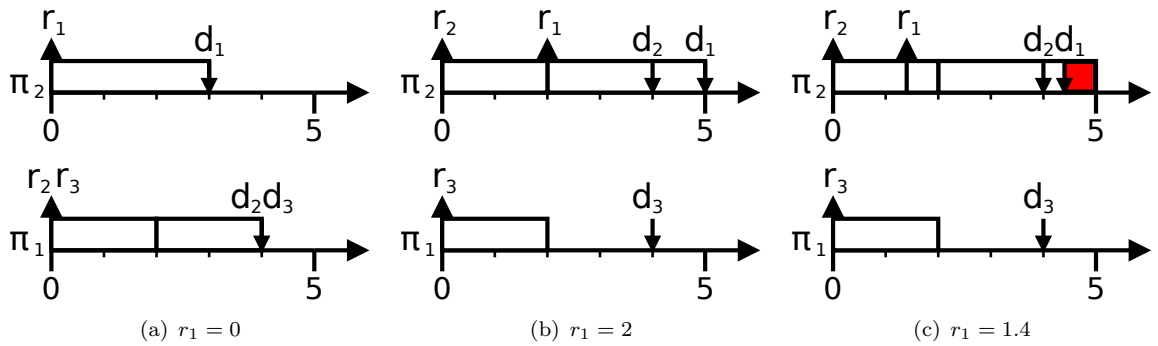
In order to avoid scheduling anomalies w.r.t. reduced execution time of jobs, we assume that subtasks execute up to their WCET at all time. Otherwise, if a subtask executes for less than its WCET, we consider that its processor is left idle in the interval between its finish time and its WCET.

Later in this chapter we provide in Section 4.3.5 simulation-based evaluations to compare our schedulability condition from Theorem 4.7 with the condition from [81] which is shown in Lemma 2.6.

4.2.2 Sustainability Analysis

In schedulability analysis, an important property is that the scheduling policy be *stable* to "positive" changes of the task timing parameters. For example, if a task set with processor utilization is schedulable according to a given scheduling policy, then it must be schedulable with a smaller utilization on the same execution platform. Otherwise, we can state that this policy is subject to scheduling anomalies. The sustainability w.r.t. positive variation in parameters has been studied in the case of EDF scheduling on uniform multiprocessors in [20].

The notion of sustainability of scheduling policy can be also applied to schedulability test. The common FP and EDF tests for uniprocessors have been examined in [18, 38]. Concerning the multiprocessor case, both scheduling policies and the schedulability tests have been discussed in [13] from the sustainability point of view. In particular, the GEDF scheduling policy was shown to be sustainable w.r.t. smaller execution requirements and later arrival times (sporadic case). However, sustainability of GEDF w.r.t. larger relative deadlines is not so straightforward. It depends on the implementation of the GEDF scheduling policy. If the priorities of jobs are computed using the priorities of tasks that generate these jobs (the *specified* priority), then



(a) $r_1 = 0$ (b) $r_1 = 2$ (c) $r_1 = 1.4$

FIGURE 4.6: An example of job collection generated by the sporadic $\{\tau_1(3,3), \tau_2(2,4), \tau_3(2,4)\}$ implicit-deadline task set where τ_i is characterized by (C_i, T_i).

GEDF is trivially sustainable w.r.t. larger relative deadlines since a smaller *actual* deadline of a job does not affect the scheduling decisions. Otherwise, it is not obvious that this property is guaranteed and it is safer to design the GEDF scheduler to compute job priorities according to the specified priorities.

Sustainability of scheduling policy

In this subsection, we review the property of sustainability of GEDF scheduling policy in the case of task sets composed of DAGs. Firstly, we give the definition of sustainability according to a scheduling policy. Secondly, we discuss three observations related to the three types of timing parameter relaxations.

Definition 4.8. (from [13]) Let $\mathcal{A}$ denote a scheduling policy. Let τ denote any sporadic task system that is $\mathcal{A}$-schedulable. Let J denote a collection of jobs generated by τ. Scheduling policy $\mathcal{A}$ is said to be sustainable if and only if $\mathcal{A}$ meets all deadlines when scheduling any collection of jobs obtained from J by changing the parameters of one or more individual jobs in any, some, or all of the following ways: (i) decreased execution requirements; (ii) larger relative deadlines; and (iii) later arrival times with the restriction that successive jobs of any task $\tau_i \in \tau$ arrive at least T_i time units apart.

According to this definition, we notice that GEDF is sustainable w.r.t. the three possible relaxations of DAG job parameters.

In order to apply the observations from [13], we used the example given in Figure 4.6. This example represents three different activation schemes that are generated by the same task set. For the sake of simplicity, we considered a task model of simple sequential jobs instead of DAG jobs.

	Decreased execution time	Larger relative deadline	Later arrival time
GEDF scheduling policy	sustainable	sustainable	sustainable, with specific deadlines
GEDF schedulability test (Theorem 4.7)	not self-sustainable	self-sustainable	self-sustainable

TABLE 4.1: Sustainability of GEDF scheduling policy and schedulability test from Theorem 4.7.

Observation 4.3. The GEDF scheduling policy is sustainable w.r.t. decreased execution requirements for any set of jobs that may be generated by a set of sporadic DAG.

Proof. As for Observation 1 in [13], this proof is based on the proof of predictability in [65, 66], and it can be applied easily on a collection of DAG jobs by considering each subtask job as an independent one. □

We observe the sustainability of GEDF w.r.t. decreased execution requirement by means of Definition 4.8. We consider a collection of jobs which have been generated by GEDF and with known release times. If this collection can be accurately scheduled w.r.t. these release times, no deadline is missed by decreasing execution requirements. In a more general case, online GEDF can generate several collections of jobs from the same sporadic task set. The same job can have different release times in several collections. We should consider jobs with release jitters if we want to observe the sustainability at task set level. Unfortunately, it has been proved in [66] that a preemptable, migratable and jittered release timed job scheduling is not predictable.

Observation 4.4. The GEDF scheduling policy is sustainable w.r.t. later arrival times for any set of jobs that may be generated by a set of sporadic DAGs.

Proof. Let τ denote a sporadic set of DAG tasks that is GEDF-schedulable. Let J' denote any collection of jobs obtained from J by increasing the arrival times of one or more individual jobs with the restriction that successive jobs of any task $\tau_i \in \tau$ arrive at least T_i time units apart. The collection J' of jobs could also have been generated by τ since it is GEDF-schedulable. Hence, the observation follows. □

According to Figure 4.6, the consequence could not be straightforward. In this example, we represent three jobs generated by a sporadic task set. In Figure 4.6(a) (respectively Figure 4.6(b)),

job J_1 is released at time $t = 0$ (respectively at time $t = 2$) and all jobs meet their deadline. However, in Figure 4.6(c), J_1 misses its deadline at time $t = 4$ since it is released at time $t = 1.4$ but no processor is available before time $t = 2$. In order to make it clear with the observation, we recall in the proof that the sporadic task set has to be schedulable according to GEDF, thus all possible release scenarios must be tested if schedulable.

Observation 4.5. The GEDF scheduling policy is sustainable w.r.t. larger relative deadlines for any set of jobs that may be generated by a set of sporadic DAGs if the scheduling algorithm is implemented by using the specified deadlines.

Proof. As explained in the section above, this observation is based on the implementation of GEDF. If the scheduling algorithm uses a larger relative deadline to compute job priorities, it is not clear that GEDF is sustainable w.r.t. deadline relaxations. But if the algorithm computes the priorities by considering the relative deadline of tasks, no change in the scheduling behavior will occur. □

The sustainability of GEDF, w.r.t timing parameters, shown in Observation 4.3, 4.4 and 4.5, is a good result. These observations are based on the fact that we considered a sporadic DAG set (or task set in a more general way) as GEDF-schedulable. We now propose a sufficient feasibility condition.

4.2.3 Schedulablity test

We have shown that GEDF is a sustainable scheduling policy for a sporadic DAG set. We now consider the sustainability of the schedulability test proposed in Theorem 4.7 according to the following definition.

Definition 4.9. (from [13]) Let $\mathcal{A}$ denote a scheduling policy, and F an $\mathcal{A}$-schedulability test for sporadic task systems. Let τ denote any sporadic task system deemed to be $\mathcal{A}$-schedulable by $\mathcal{F}$. Let J denote a collection of jobs generated by τ. $\mathcal{F}$ is said to be a sustainable schedulability test if and only if scheduling policy $\mathcal{A}$ meets all deadlines when scheduling any collection of jobs obtained from J by changing the parameters of one or more individual jobs in any, some, or all of the following ways: (i) decreased execution requirements; (ii) larger relative deadlines; and (iii) later arrival times with the restriction that successive jobs of any task $\tau_i \in \tau$ arrive at least T_i time units apart.

Observation 4.6. The test proposed in Theorem 4.7 is sustainable for sporadic DAG w.r.t. decreased execution requirements, larger relative deadlines and later arrival times.

Proof. A DAG set deemed schedulable by the test proposed in Theorem 4.7 is GEDF schedulable. GEDF is a sustainable scheduling policy w.r.t. decreased execution requirements, larger relative deadlines and later arrival times. The observation follows. □

In addition to the sustainability of our GEDF DAG schedulability test, we studied the *self-sustainability* of our test.

Definition 4.10. (from [13]) A schedulability test is self-sust-ainable if all task systems with "better" (less constraining) parameters than a task system deemed to be schedulable by the test are also deemed schedulable by the test.

In this case, the task set is not only required to remain schedulable under various parameter relaxations, but it is required to be verifiably schedulable by the same test.

Observation 4.7. GEDF DAG schedulability test from Theorem 4.7 is self-sustainable w.r.t. decreased execution requirements.

Proof. In Equation (4.6), decreasing execution requirements can only decrease the left side of the inequality since only DBF and ζ_i^k values depend on the WCET values. The inequality remains valid and the observation follows. □

Observation 4.8. GEDF DAG schedulability test from Theorem 4.7 is self-sustainable w.r.t. later arrival times.

Proof. In Equation (4.6), later arrival times can only decrease the left side of the inequality since only DBF values depend on the period values. The inequality remains valid and the observation follows. □

Observation 4.9. GEDF DAG schedulability test from Theorem 4.7 is not self-sustainable w.r.t. larger relative deadlines.

Proof. We assume a system with only one unit-speed processor. From Equation (4.6), we obtain:

$$\sum_{i=1}^{n} DBF_k^i + \sum_{i=1, i\neq k}^{n} \zeta_k^i \leq D_k$$

We recall that ζ_k^i is defined by the following expression:

$$\sum_{\tau_{i,j} \in \tau_i} \min\left(C_{i,j}, \max\left(0, d_{i,j} - r_k\right)\right) \tag{4.7}$$

Let us assume that in Equation (4.7), the "min" value is given by $\max(0, d_{i,j} - r_k)$. Then we can assume that there is $d'_{i,j} > d_{i,j}$ such that $\max(0, d'_{i,j} - r_k) > \max(0, d_{i,j} - r_k)$ but $\max(0, d'_{i,j} - r_k) \leq C_{i,j}$. The deadline $d'_{i,j} > d_{i,j}$ implies a larger carry-in interference of a job of τ_i on a job of τ_k which can miss its deadline. □

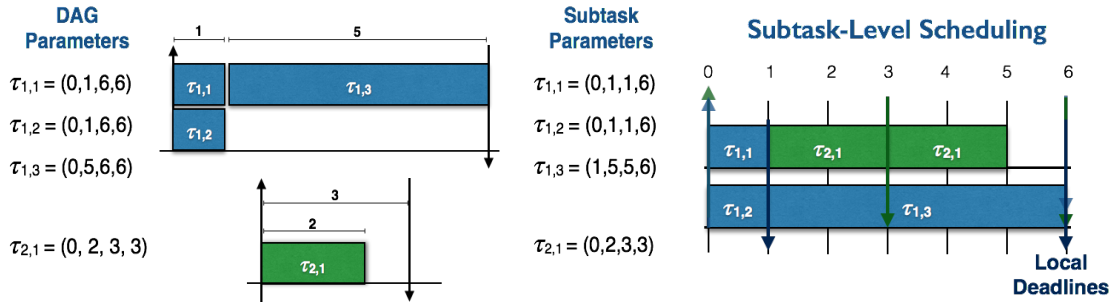
Unfortunately, our proposed schedulability test for GEDF is not self-sustainable w.r.t. larger relative deadlines. These results can be explained because the test is based on the analysis of the schedule on a study window of size corresponding to the DAG's deadline.

4.3 Scheduling DAGs using Local Parameters (Subtask-Level Scheduling)

In this section, we present a new scheduling approach for DAG tasks on multiprocessor systems which is the Subtask-Level Scheduling. As in any Direct Scheduling approach, DAG tasks are scheduled using common real-time multiprocessor algorithms, however, the scheduling decisions are taken based on the timing parameters of subtasks rather than global parameters of DAGs. Since subtasks in our DAG model are defined by WCET and precedence constraints, the Subtask-Level Scheduling uses the assigned local timing parameters (from Section 4.1) which are local offsets, local relative deadlines and release jitters.

This section is divided as follows: Subsection 4.3.2 provides an interference analysis of any multiprocessor work-conserving scheduling algorithm at a subtask level. Then we extend it to a workload analysis which serves as an upper bound on interference in Subsection 4.3.3. We provide a workload schedulability condition for GEDF scheduling algorithm in Subsection 4.3.4.

Finally in Subsection 4.3.5, we compare the different DAG scheduling approaches by using extensive simulation. We compare DAG-Level scheduling with Subtask-Level and we also compare them with other conditions based on Direct Scheduling of DAGs which are found in the state of the art.



(a) DAG set consists of 2 implicit-deadline DAGs.

(b) Sub-Level scheduling when GEDF algorithm is used.

FIGURE 4.7: Example of Subtask-Level Scheduling of DAG tasks.

An example of Subtask-Level Scheduling

The concept of Subtask-Level scheduling is straightforward. After assigning local timing parameters to subtasks, a scheduling algorithm is used to schedule them based on these parameters. Hence, subtasks of the same DAG task can be assigned specific priorities according to their local parameters, unlike DAG-Level scheduling which assigns the same priority to all subtasks of the same DAG. For example, in the case of GEDF scheduling algorithm, a subtask is activated whenever its predecessors terminate their execution, and the subtask job with the earliest local absolute deadline is assigned the highest priority.

An example of Subtask-Level approach with GEDF scheduling algorithm is shown in Figure 4.7. The DAG set is shown in Figure 4.7(a). It consists of 2 periodic implicit-deadline DAG tasks. The timing parameters of each subtask are shown as a quadruple which consists of an offset, a WCET, a relative deadline and a period. The DAG-Level parameters of subtasks are shown on the left side of the figure, while the assigned local parameters of subtasks are shown on the right hand side. Based on Subtask-Level scheduling, subtask jobs are assigned priorities based on their absolute deadlines as shown in Figure 4.7(b). At time $t = 0$, both DAG tasks are activated and jobs of subtasks $\tau_{1,1}$ and $\tau_{1,2}$ as assigned higher priority than the job of subtask $\tau_{2,1}$, because their absolute local deadline at $t = 1$ is earlier than the absolute deadline of $\tau_{2,1}$ at time $t = 3$. However, if DAG-Level scheduling is applied, then all subtask jobs of DAG τ_1 will be assigned a priority lower than DAG job of τ_2 at time $t = 0$, since the absolute deadline of τ_1 is at $t = 6$.

4.3.1 Advantage of Subtask-Level Scheduling

Here, we provide an example showing how Subtask-Level scheduling can be suitable for DAG tasks since it takes into consideration their internal structure. We consider the processor load feasibility condition which is a common necessary condition for multiprocessor task sets. The processor load of a task set τ is defined as the maximum value of total demand bound function (DBF) (from Definition 1.2 on page 5) of all tasks in τ within an interval of $[0, t)$ divided by its length t.

$$load(\tau) = \max_{\forall t} \left(\frac{\sum_{\forall \tau_i \in \tau} max(0, DBF^i(t))}{t} \right) \tag{4.8}$$

where $DBF^i(t)$ is defined in Equation (1.2).

The necessary feasibility condition [19] based on processor load states that a task set cannot be schedulable on m identical processors using any scheduling algorithm if the execution demand of the task set exceeds the system's capacity within any interval:

$$load(\tau) \leq m$$

At a subtask level, let $DBF^{i,j}$ denote the demand bound function from jobs of subtask $\tau_{i,j}$ of executing DAG $\tau_i \in \tau$ in an interval $[0, t)$. Subtasks are assigned local offsets based on their precedence constraints and the execution order of their predecessors. We define $DBF^{i,j}$ as follows:

$$DBF^{i,j}(t) = \left(\left\lfloor \frac{t - O_{i,j} - D_{i,j}}{T_{i,j}} \right\rfloor + 1 \right)^+ \times C_{i,j} \tag{4.9}$$

where $x^+ = \max(0, x)$.

In time interval $[0, t)$, we suppose that each DAG task $\tau_i \in \tau$ is activated at time $t = 0$. The assigned local offset $O_{i,j}$ of subtask $\tau_{i,j}$ represents the minimum delay between the activation time r_i of its DAG job J_i and the activation time $r_{i,j}$ of subtask job $J_{i,j}$. Hence, a job of subtask $\tau_{i,j}$ is activated after at least $O_{i,j}$ time units of the release time of its DAG job, and in a time interval $[0, t)$, subtask $\tau_{i,j}$ is activated in $[O_{i,j}, t)$.

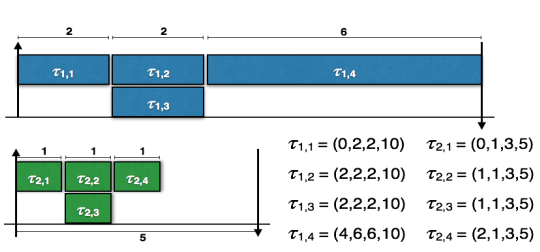
We modify the processor load by using the local timing parameters of subtasks and the DBF at subtask level (from Equation (4.9)). The necessary condition of processor load is defined as follows:

$$load(\tau) = \max_{\forall t} \left(\frac{\sum_{\forall \tau_i \in \tau} \sum_{\forall \tau_{i,j} \in \tau_i} max(0, DBF^{i,j}(t))}{t} \right) \leq m \tag{4.10}$$

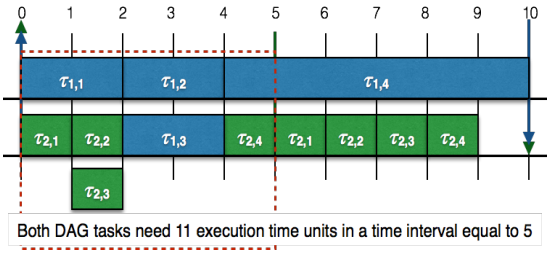
We believe that this necessary feasibility condition of processor load is more adapted to DAG tasks, since the DBF is calculated based on the execution requirements of subtasks rather than their DAGs. We provide an example to prove the correctness of this assumption.

Example 4.5 (Necessary condition of processor load at Subtask-Level)**.**
In Figure 4.8, we present an example to show the difference between the necessary condition of processor load at DAG-Level and at Subtask-Level. Figure 4.8(a) shows the DAG set τ which consists of 2 periodic implicit-deadline DAGs (τ_1 and τ_2). The figure shows the structure of DAG tasks and the local timing parameters of their subtasks. DAG τ_1 has a deadline and a period equal to 10. It consists of 4 subtasks $(\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,4})$ where subtasks have a WCET equal to 2 except for subtask $\tau_{1,4}$ whose WCET is equal to 6. DAG τ_2 consists of 4 subtasks which have identical WCETs equal to 1 and has a deadline and a period equal to 5.

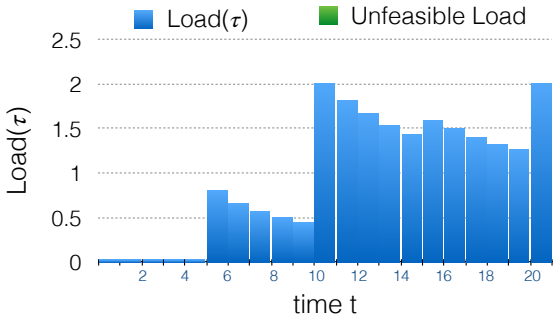
Based on the necessary multiprocessor feasibility condition of system utilization which determines the minimum number of processors on which a task set may be schedulable (from Equation (1.1) on page 5), we consider that DAG task τ (whose utilization is equal to 2) is executing on a platform of 2 homogeneous unit-speed processors ($m = 2$). By assuming synchronous activation of DAG set at time $t = 0$, we can notice that τ is not schedulable by any global preemptive scheduling algorithm on 2 processors, as shown in Figure 4.8(b). In order for the DAG set to be feasible, DAG τ_1 must execute without delaying any of its subtasks, because it has no slack time and its critical path length L_1 is equal to it relative deadline ($L_1 = D_1 = 10$). In the case of synchronous activation, the first job of DAG τ_2 is activated within time interval $[0, 5)$ and it executes for 5 time units within this interval. Based on the structure of both DAGs and the scheduling proposed in Figure 4.8(b), the DAG set needs to execute 11 time units within time interval $[0, 5)$ to meet all deadlines of the jobs. We can conclude that the DAG set is not feasible on a system of 2 unit-speed processors.



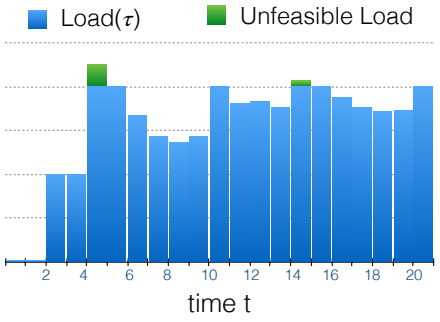
(a) DAG set τ.



(b) DAG set τ is not schedulable on a platform of 2 processors.



(c) Processor load of DAG-Level scheduling.



(d) Processor load of Subtask-Level Scheduling.

FIGURE 4.8: Processor load analysis of DAG-Level vs. Subtask-Level Scheduling.

t	1	2	3	4	5	6	7	8	9	10
$load(\tau)$	0.025	0.025	0.025	0.025	0.8	0.667	0.57	0.5	0.44	2
t	11	12	13	14	15	16	17	18	19	20
$load(\tau)$	1.818	1.667	1.538	1.43	1.6	1.5	1.41	1.33	1.26	2

TABLE 4.2: Processor load of DAG set τ from Figure 4.8 at DAG-Level Scheduling.

t	1	2	3	4	5	6	7	8	9	10
$load(\tau)$	0.02	1	1	2.25	2	1.667	1.43	1.375	1.44	2
t	11	12	13	14	15	16	17	18	19	20
$load(\tau)$	1.82	1.83	1.77	2.07	2	1.875	1.765	1.72	1.74	2

TABLE 4.3: Processor load of DAG set τ from Figure 4.8 at Subtask-Level Scheduling.

Since the necessary feasibility condition of task set utilization failed in identifying the infeasibility of the DAG set, we want to test the necessary condition of processor load at DAG-Level and at Subtask-Level and verify whether they can detect the infeasibility of the system or not. Starting by the DAG-Level scheduling, the global parameters of DAG tasks are used to calculate the processor load at DAG-Level (from Equation (4.8)) in different time intervals. The processor load for all time intervals $[0, t)$*, where* $1 \leq t \leq 20$*. The results are shown in Table 4.2, and they are represented graphically in Figure 4.8(c). We can notice that processor load at DAG-Level*

never exceeds 2 *for all time intervals, which is a contradiction with the scheduling example in Figure 4.8(c) that proves the infeasibility of the DAG set. Hence, the necessary condition of processor load at DAG-Level fails in identifying the exact execution requirement of DAG set in time interval* $[0, 5)$.

For Subtask-Level Scheduling, we calculate processor load of the DAG set based on Equation (4.3) *which calculates the DBF of subtasks by using their assigned local timing parameters (the quadruple from Figure 4.8(a)). Table 4.3 and Figure 4.8(c) show the processor load at Subtask-Level for time intervals* $[0, t)$ *where* $1 \leq t \leq 20$. *According to these results, the processor load at Subtask-Level is equal to* 2.25 *at time interval* $[0, 5)$ *(respectively,* 2.07 *at time interval* $[0, 15)$*). We detect that DAG set* τ *performs a total amount of execution time at specific time intervals that requires more than* 2 *processors. As a result, the feasibility condition of processor load at Subtask-Level states that* τ *needs at least* 3 *processors to execute on, which detects the infeasibility of* τ *on* 2 *processors as shown in Figure 4.8(b).*

4.3.2 Interference Analysis

In this section, we provide an interference analysis for any work conserving algorithm for the Subtask-Level Scheduling of DAG tasks. The interference on a subtask $\tau_{k,h}$ from DAG τ_k in a specific interval $[a, b)$ is defined as follows:

Definition 4.11. $I_{k,h}(a, b)$ is the length of all intervals where subtask $\tau_{k,h}$ is ready to execute but blocked by higher priority subtasks in time interval $[a, b)$.

Definition 4.12. $I^{i,j}_{k,h}(a, b)$ is the length of all intervals where subtask $\tau_{k,h}$ is ready to execute but blocked by subtask $\tau_{i,j}$ which has higher priority in time interval $[a, b)$.

In the case of sequential independent real-time tasks, the interference that a task can suffer in a specific time interval is the sum of interference of all other tasks (in the same interval) divided by the number of processors[5]. Our DAG model defines subtasks as real-time threads that execute sequentially. Hence, the same interference relation $I_{k,h}(a, b)$ and $I^{i,j}_{k,h}(a, b)$ can be applied as follows:

$$I_{k,h}(a, b) = \frac{1}{m} \times \sum_{\forall \tau_{i,j} \in \tau_i \in \tau} I^{i,j}_{k,h}(a, b) \tag{4.11}$$

[5]From Lemma (3) in [30].

As stated earlier, subtasks of the same DAG may be assigned specific priorities by scheduling algorithms at Subtask-Level, based on their local timing parameters. However, we consider that predecessor subtasks of a given subtask $\tau_{k,h}$ have implicitly higher priorities, because the execution of any jobs of $\tau_{k,h}$ is blocked until its predecessors jobs complete their own. Hence, successor subtasks of $\tau_{k,h}$ have lower priorities. The sibling subtasks of $\tau_{k,h}$, which execute independently and in parallel, are assigned specific priorities by the scheduling algorithm. In addition to the interference from other DAG tasks in the set, the interference on a subtask $\tau_{k,h}$ is divided into two sources: internal interference and external interference.

Let $Ie_{k,h}(a,b)$ denote the external interference from higher priority subtasks of DAG tasks other than τ_k in the set in time interval $[a,b)$. These external subtasks have no precedence constraints or execution dependencies with subtask $\tau_{k,h}$. The interference can be defined as follows:

$$Ie_{k,h}(a,b) = \frac{1}{m} \times \sum_{i \neq k, \forall \tau_{i,j} \in \tau_i \in \tau} I_{k,h}^{i,j}(a,b) \tag{4.12}$$

where some or all of the subtasks of DAG task τ_i (other than τ_k) can interfere with $\tau_{k,h}$ based on their priorities.

Let $Ii_{k,h}(a,b)$ denote the internal interference on subtask $\tau_{k,h}$ from other subtasks in τ_k in time interval $[a,b)$. Since we consider constrained-deadline DAG tasks and that subtasks of a given DAG share the same period of their DAG, one job at most from each subtask contributes in the internal interference. The internal interference depends on the type of interfering subtasks which are divided into categories based on their execution dependencies with subtask $\tau_{k,h}$:

- A predecessor subtask $\tau_{k,x} \in Pred(\tau_{k,h})$: this subtask delays the activation time of $\tau_{k,h}$, but once subtask $\tau_{k,x}$ completes its execution, there will be no further effect of $\tau_{k,x}$ on $\tau_{k,h}$.
- A sibling subtask $\tau_{k,x} \in Sibling(\tau_{k,h})$: for a specific job of subtask $\tau_{k,h}$, their sibling subtasks can interfere once with its execution, based on the priorities assigned to them by the scheduling algorithm.
- A successor subtask $\tau_{k,x} \in Succ(\tau_{k,h})$: this subtask cannot interfere with subtask $\tau_{k,h}$, because both subtasks cannot execute in parallel, and subtask $\tau_{k,x}$ starts its execution after $\tau_{k,h}$ completes its own. According to this, $I_{k,h}^{k,x}(a,b) = 0$,
- Subtask $\tau_{k,h}$ has no interference since we consider constrained-deadline DAG tasks, in which only one job from each DAG task is activate at any time t. Hence, $I_{k,h}^{k,h}(a,b) = 0$.

Based on the above definitions, the internal interference $Ii_{k,h}(a,b)$ on subtask $\tau_{k,h}$ is defined as:

$$Ii_{k,h}(a,b) = \frac{1}{m} \times \sum_{\forall \tau_{k,i} \notin succ(\tau_{k,h}); i \neq h} I_{k,h}^{k,i}(a,b) \tag{4.13}$$

Let $J_{k,h}^*$ be the job of subtask $\tau_{k,h}$ which suffers from maximum interference, and let $I_{k,h}(r_{k,h}^*, d_{k,h}^*)$ denote the worst-case interference for subtask job $J_{k,h}^*$ of $\tau_{k,h}$ in its activation interval $[r_{k,h}^*, d_{k,h}^*)$. For the sake of clarity, we use $\widehat{I}_{k,h}$ to denote the maximum interference on subtask $\tau_{k,h}$, where $\widehat{I}_{k,h} = I_{k,h}(r_{k,h}^*, d_{k,h}^*)$.

Lemma 4.13. *A DAG set τ of sporadic constrained-deadline DAGs is schedulable on m identical unit-speed processors, for any work conserving algorithm if:*

$$\forall \tau_{k,h} \in \tau_k \in \tau$$
$$\widehat{I}_{k,h} = \widehat{Ie}_{k,h} + \widehat{Ii}_{k,h} \leq (D_{k,h} - C_{k,h}) \tag{4.14}$$

Proof. The proof of this lemma is straightforward. The interference on a subtask job comes from its two main sources: internal and external interference. In order for any subtask job to be schedulable, its execution window (between its activation time and deadline) should be long enough to execute its execution requirement plus the interference from higher priority subtasks in the system. □

Interference from Predecessor Subtasks

In Section 4.1.2, we assigned a maximum release jitter for each subtask in the DAG. Based on Definition 4.3 (on page 96), the maximum release jitter $\widehat{j}_{k,h}$ represents the maximum length of the time interval in which any job of subtask $\tau_{k,h}$ can be activated due to its precedence constraints and the execution dependencies of its predecessor jobs. If we consider that all predecessors of $\tau_{k,h}$ have executed as late as possible, then subtask job $J_{k,h}^*$ cannot be delayed more than $\widehat{j}_{k,h}$ time units after its release time $r_{k,h}^*$. If we consider that job $J_{k,h}^*$ has a release jitter $j_{k,h} \leq \widehat{j}_{k,h}$, then we can exclude predecessor subtasks from the interference calculation in Equation (4.14), and Lemma 4.13 can be modified as follows:

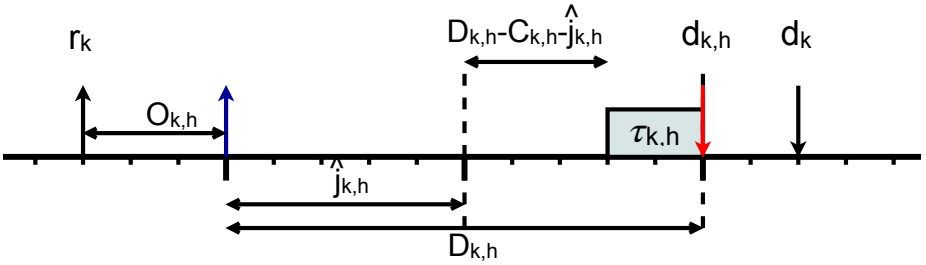


FIGURE 4.9: The interference window on subtask $\tau_{k,h}$ excluding the interference from its predecessor subtasks.

Lemma 4.14. *A DAG set τ is schedulable on m identical unit-speed processors using any work conserving algorithm, if:*

$$\forall \tau_{k,h} \in \tau_k \in \tau$$

$$\widehat{I}e_{k,h} + \widehat{I}i_{k,h} \leq (D_{k,h} - C_{k,h} - \widehat{j}_{k,h}) \leq (D_{k,h} - C_{k,h} - j_{k,h}) \quad (4.15)$$

where

$$\widehat{I}i_{k,h} = \frac{1}{m} \times \sum_{\forall \tau_{k,i} \in sibling(\tau_{k,h})} I_{k,h}^{k,i}(a,b)$$

Proof. As shown in Figure 4.9 and based on the definition of the maximum release jitter in Definition 4.3, if all predecessor jobs complete their execution just before their local deadlines, then any job of subtask $\tau_{k,h}$ will be activated no later than $t = (r_k + O_{k,h} + \widehat{j}_{k,h})$. In the interval $[t, d_{k,h})$, predecessors will have no further interference, and only sibling subtasks of $\tau_{k,h}$ will interfere with $\tau_{k,h}$. If subtask $\tau_{k,h}$ is feasible, the total interference on any of its jobs should not exceed its slack time $(D_{k,h} - C_{k,h} - \widehat{j}_{k,h})$. However, if we consider the actual release jitter $j_{k,h}$ which is less than or equal to its maximum value $\widehat{j}_{k,h}$, then the slack time of the job will be increased and we will have a better feasibility condition. □

As mentioned earlier, the maximum release jitter $\widehat{j}_{k,h}$ of subtask $\tau_{k,h}$ in interference analysis is too pessimistic, because it considers that predecessor subtasks execute as late as possible (just before their absolute deadlines). As a result, the upper bound on interference used in Lemma 4.14 is always zero for any critical subtask $\tau_{k,h}$ since their maximum release jitter $\widehat{j}_{k,h} = D_{k,h} - C_{k,h}$, and they have no slack time.

As shown in Figure 4.10, it is possible to optimize the release jitter of subtask $\tau_{k,h}$ to calculate its finish time when we consider the maximum interference on any parent subtask $\tau_{k,i}$. Any job

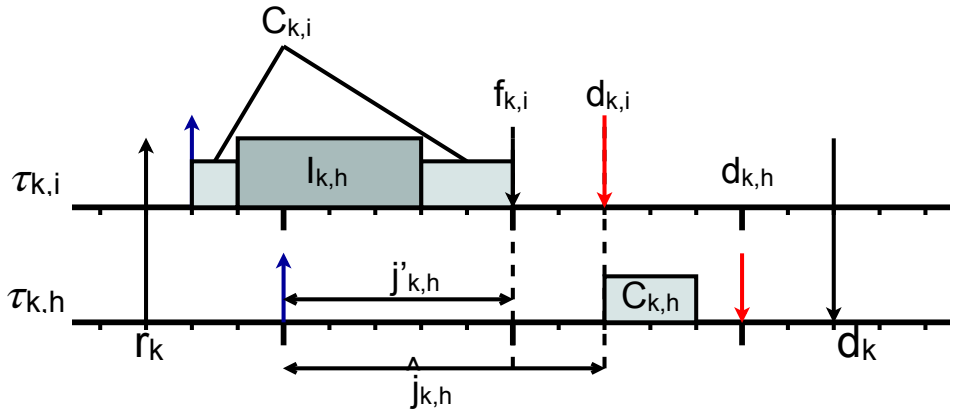


FIGURE 4.10: The optimized release jitter of subtask $\tau_{k,h}$ which has a sole parent $\tau_{k,i}$.

of subtask $\tau_{k,i}$ has a response time equal to $(\widehat{I}_{k,i} + C_{k,i})$ when any work conserving algorithm is used, and its latest finish time $f^*_{k,i}$ can be defined as:

$$f^*_{k,i} = C_{k,i} + \widehat{I}_{k,i} \leq D_{k,i}$$

The finish time $f_{k,i}$ of any job of a schedulable subtask $\tau_{k,i}$ should not be greater than its local deadline $D_{k,i}$. Otherwise, a deadline miss will occur.

By using the finish time of each parent subtask of $\tau_{k,h}$, we can calculate its optimized release jitter $j'_{k,h}$, which is defined as follows:

$$\begin{aligned} j'_{k,h} &= \max_{\forall \tau_{k,i} \in parents(\tau_{k,h})} (f^*_{k,i} - (O_{k,h} - O_{k,i})) \\ &= \max_{\forall \tau_{k,i} \in parents(\tau_{k,h})} (C_{k,i} + \widehat{I}_{k,i} - (O_{k,h} - O_{k,i})) \\ &\leq \widehat{j}_{k,h} \end{aligned} \tag{4.16}$$

Corollary 4.15. *A DAG set τ is schedulable on m identical unit-speed processors using any work conserving algorithm, if:*

$$\forall \tau_{k,h} \in \tau_k \in \tau \qquad\qquad \widehat{Ie}_{k,h} + \widehat{Ii}_{k,h} \leq (D_{k,h} - C_{k,h} - j'_{k,h})$$

where

$$\widehat{Ii}_{k,h} = \frac{1}{m} \times \sum_{\forall \tau_{k,i} \in sibling(\tau_{k,h})} I^{k,i}_{k,h}(a,b) \tag{4.17}$$

As a result, the use of the optimized release jitter of a subtask instead of its maximum release jitter improves our feasibility condition by considering a more accurate upper bound on interference.

4.3.3 Workload Analysis for Work Conserving Algorithms

In global multiprocessor systems, the problem of quantifying the exact interference (not the maximum interference) of a task on another one is a challenging problem when compared to uniprocessor systems, because jobs can execute on any processor of the system and it is hard to determine which specific job blocks the execution of another and for how many time units. The same problem can be applied to the interference analysis of DAG tasks at Subtask-Level, which we used in the schedulability condition of Corollary 4.15. However, calculating the workload of an interfering task in a given interval is much easier than its exact interference, but at the expense of pessimism. So, it is possible to use it as an upper bound on interference subtasks within the same interfering interval. It is based on the fact that the interference of a subtask on a lower priority subtask in a fixed interval cannot exceed its workload during the same interval. Let $W_{i,j}(a,b)$ be the amount of work performed by the jobs of subtask $\tau_{i,j}$ in time interval $[a,b)$, then:

$$I_{k,h}^{i,j}(a,b) \leq W_{i,j}(a,b)$$

In the remainder of this subsection, we provide workload analysis of interfering subtasks on any job of subtask $\tau_{k,h}$ in a time interval of length equal to its local deadline $D_{k,h}$. As stated earlier in the interference section, the workload analysis has two main sources: internal workload from sibling subtasks and external workload from subtasks from other DAGs in the set.

Workload Analysis from Sibling Subtasks

A sibling subtask $\tau_{k,i}$ of $\tau_{k,h}$ is a subtask that belongs to the same DAG task τ_k and it can execute in parallel with $\tau_{k,h}$. Moreover, subtask $\tau_{k,i}$ has no precedence relations with $\tau_{k,h}$ and it is not among its predecessor or successor subtasks.

For a given subtask, at most one job from each sibling subtask can interfere with any of its jobs, because these jobs share the same period and the release time of their DAG job is considered as their activation reference. In other words, one job from each sibling subtask of $\tau_{k,h}$ is released in the interval $[r_k + O_{k,h}, r_k + O_{k,h} + D_{k,h}]$. For any work conserving algorithm, the maximum

internal interference $\widehat{Ii}_{k,h}^{k,i}$ of a subtask $\tau_{k,i}$ on its sibling $\tau_{k,h}$ is calculated by identifying the length of the maximum interference interval $L_{k,h}^{k,i}$ of $\tau_{k,i}$ on $\tau_{k,h}$. It is defined as the length of the longest interval in which subtasks $\tau_{k,h}$ and $\tau_{k,i}$ can execute in parallel. It can be calculated as follows:

$$L_{k,h}^{k,i} = min(\overline{D}_{k,i}, \overline{D}_{k,h}) - max(O_{k,i}, O_{k,h}) \tag{4.18}$$

For the sake of clarity, we denote by $\overline{D}_{k,h}$ the relative deadline of subtask $\tau_{k,h}$ from the release time of the DAG task, where $\overline{D}_{k,h} = O_{k,h} + D_{k,h}$. Hence, the sibling interference interval is defined as $[max(O_{k,i}, O_{k,h}), max(O_{k,i}, O_{k,h}) + L_{k,h}^{k,i})$. In the example from Figure 4.2 (on page 93), sibling subtasks $\tau_{1,2}$ and $\tau_{1,3}$ from DAG τ_1 have an interference interval $[1, 5)$ whose length is equal to $L_{1,3}^{1,2} = 4$.

Lemma 4.16. *The maximum internal interference $\widehat{Ii}_{k,h}^{k,i}$ of subtask $\tau_{k,i}$ on a job of its sibling subtask $\tau_{k,h}$ is upper bounded as follows:*

$$\widehat{Ii}_{k,h}^{k,i} \leq min(C_{k,i}, L_{k,h}^{k,i}) = \widehat{Wi}_{k,i}$$

where $L_{k,h}^{k,i}$ is defined in Equation (4.18).

Proof. Based on the definition of the interference interval $L_{k,h}^{k,i}$, the maximum possible workload of subtask $\tau_{k,i}$ in the interval happens when $\tau_{k,i}$ executes as long as possible in the interference interval. However, a subtask job cannot execute more than its WCET in a given time interval, then the performed work within the interval is the minimum between its WCET $C_{k,i}$ and the length of the interference interval $L_{k,h}^{k,i}$. □

Workload Analysis for External Subtasks

For any work conserving algorithm, Bertogna et al. [32] identified the worst-case activation scenario of interfering jobs in a fixed interval (a, b) which generates the maximum possible workload. They considered a task model of independent constrained-deadline sequential tasks on multiprocessor systems. As shown in Figure 4.11, this scenario is defined when the carry-in job of an interfering task starts its execution at the beginning of the interference interval, and when it executes just before its deadline. The successive body jobs and at most one carry-out job execute as soon as possible until the end of the interval. This scenario is proved in [32] to generate the maximum workload in the interval.

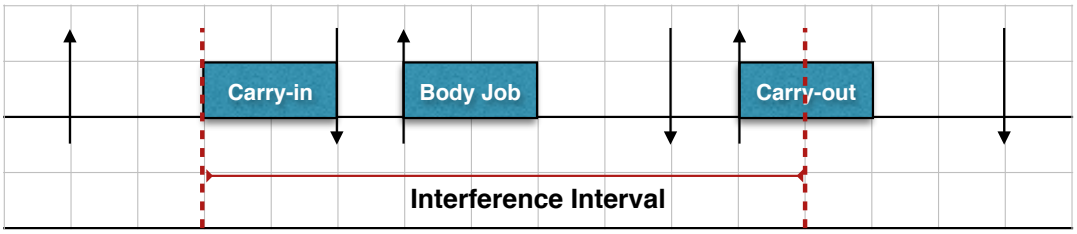
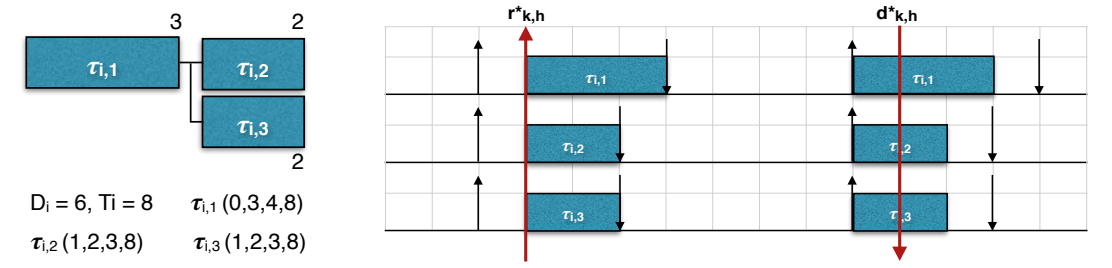


FIGURE 4.11: The densest possible packing of jobs in an interference interval for traditional task using any work conserving algorithm (from [32]).

We apply the same scenario on DAG tasks so as to calculate the performed workload of an interfering external subtask on another particular subtask $\tau_{k,h}$ in the system. This is possible because interfering subtask jobs do not belong to the same DAG τ_k and their execution is independent from $\tau_{k,h}$. In order to calculate the upper bound on interference from external subtasks on job $J^*_{k,h}$, we assume that all interfering subtask jobs from the same DAG ($\forall \tau_{i,j} \in \tau_i$ where $i \neq k$) are activated based on the worst-case activation scenario from Figure 4.11. As shown in the example from Figure 4.12, a DAG task τ_i consists of 3 subtasks with relative deadline $D_i = 6$ and a period $T_i = 8$. Subtask $\tau_{i,1}$ is the source subtask of the DAG and subtasks $\tau_{i,2}$ and $\tau_{i,3}$ are its successors. The local timing parameters of subtasks are shown in Inset 4.12(a) in which each subtask $\tau_{i,j}$ is characterized as $(O_{i,j}, C_{i,j}, D_{i,j}, T_{i,j})$. Inset 4.12(b) shows our considered worst-case activation scenario of subtask jobs in the interference interval $[r^*_{k,h}, d^*_{k,h})$. All carry-in jobs of subtasks start their execution at time $t = r^*_{k,h}$ while successive body jobs execute as soon as possible.

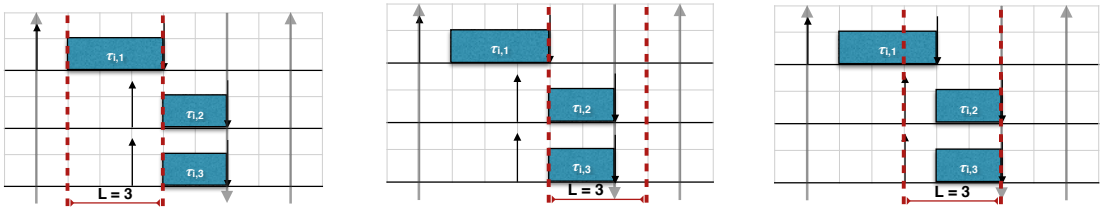
Using this worst-case activation scenario of subtask jobs is used to identify an upper bound on performed workload in the interference interval, although it is pessimistic. Because interfering subtasks from the same DAG task have precedence constraints that define their execution order and dependencies. For example, subtask $\tau_{i,1}$ from Figure 4.12(b) cannot execute in parallel with its successor subtasks $\tau_{i,2}$ and $\tau_{i,3}$. However, identifying the worst-case activation scenario of jobs, which is adapted to DAG tasks and which takes into consideration the precedence constraints between their subtasks, is not trivial. The example in Figure 4.12 describes the problem difficulty.

Example 4.6 (Workload of external subtasks)**.**
We consider the DAG task τ_i from Figure 4.12(a). According to the precedence constraints between its subtasks, the actual activation scenario of their jobs cannot be as shown in Figure 4.12(b), because subtask $\tau_{i,1}$ cannot execute in parallel with subtasks $\tau_{i,2}$ and $\tau_{i,3}$. In Figures 4.12(c), 4.12(d) and 4.12(e), we show the actual activation scenario of subtask jobs based on



(a) A DAG task τ_i which consists of 3 subtasks.

(b) The worst-case workload activation scenario for subtask jobs of τ_i for any work conserving scheduling algorithm.



(c) First possible activation scenario of subtasks of τ_i. Total performed workload is equal to 3.

(d) Second possible activation scenario of subtasks of τ_i. Total performed workload is equal to 4.

(e) Third possible activation scenario of subtasks of τ_i. Total performed workload is equal to 5.

FIGURE 4.12: An example of workload analysis of external subtasks.

their precedence constraints. Moreover, we consider an interference interval of length equal to 3 *which we shift forward and backward based on the local offset and deadline of each subtask in* τ_i *as shown in the Figures. As a result, we can identify the maximum workload of subtask jobs in the interference interval.*

First scenario (from Figure 4.12(c)) considers that subtask $\tau_{i,1}$ *starts at the beginning of the interference interval. The length of the interval is equal to the WCET of* $\tau_{i,1}$*, then its successors are activated after the end of the interval. The total performed workload in this interval is equal to* $C_{i,1} = 3$*. The second scenario is shown in Figure 4.12(d), in which we consider that the interference interval begins at the release time of subtasks* $\tau_{i,2}$ *and* $\tau_{i,3}$*, and subtask* $\tau_{i,1}$ *terminates its execution before the beginning of the interval. The total workload is equal to* 4 *which is the sum of their WCET. However, both scenarios are not the one that generates the maximum workload. We notice in Figure 4.12(e) that if the interference interval begins during the execution of subtask* $\tau_{i,1}$ *(*1 *time unit before its deadline* $d_{i,1}$*) and it ends at the absolute local deadline of subtask jobs* $\tau_{i,2}$ *and* $\tau_{i,3}$*,then the total performed workload is equal to* 5*.*

We conclude that the activation scenario of interfering DAG τ_i in an interval of length 3 at Subtask-Level, that generated the maximum workload, does not comply with the worst-case activation scenario from Figure 4.11 when it is applied on each subtask individually. Hence, to calculate the maximum workload of external subtasks from the same DAG task, either we

have to test all possible placements of interference interval and analyze their workload w.r.t. the release time of subtask jobs, or we consider an upper bound on workload as shown in Figure 4.12(b). The latter option is easier to be calculated despite of its pessimism.

Lemma 4.17. *The external interference $\widehat{Ie}_{k,h}^{i,j}$ of subtask $\tau_{i,j}$ on subtask $\tau_{k,h}$ in a time interval whose length is equal to the relative deadline $D_{k,h}$ of $\tau_{k,h}$, is bounded by:*

$$\widehat{Ie}_{k,h}^{i,j} \leq N_{i,j}(D_{k,h})C_{i,j} + min\left(C_{i,j}, D_{k,h} + D_{i,j} - C_{i,j} - N_{i,j}(D_{k,h})T_{i,j}\right) \tag{4.19}$$

where

$$N_{i,j}(D_{k,h}) = \left\lfloor \frac{D_{k,h} + D_{i,j} - C_{i,j}}{T_{i,j}} \right\rfloor$$

Proof. This lemma is a generalization of the one in [32] which considers independent sequential tasks. The maximum workload of an external interfering subtask $\tau_{i,j}$ on subtask $\tau_{k,h}$ is generated based on the worst-case activation scenario described in [32] and shown in Figure 4.11. The amount of workload is based on the number of interfering jobs $N_{i,j}$ which are entirely within the interference interval in addition to the carry-out job which executes at the end of the interval and may contribute partially to the interference (represented by the *min* operation in Equation (4.19)). More details about these equations can be found in [32]. □

A schedulability condition for DAG tasks using any work conserving algorithm on m identical processors, is provided by the following theorem:

Theorem 4.18. *A DAG set τ is schedulable on m identical processors using any work conserving algorithm if:*

$$\forall \tau_{k,h} \in \tau_k \in \tau$$
$$\sum_{\tau_{k,i} \in sibling(k,h)} min(\widehat{Ii}_{k,h}^{k,i}, D_{k,h} - C_{k,h} - j'_{k,h}) + \sum_{\tau_{i,j}\,;\, i \neq k} min(\widehat{Ie}_{k,h}^{i,j}, D_{k,h} - C_{k,h} - j'_{k,h}) \leq m(D_{k,h} - C_{k,h} - j'_{k,h})$$

where $\widehat{Ii}_{k,h}^{k,i}$ (respectively $\widehat{Ie}_{k,h}^{i,j}$) is provided in Lemma 4.16 (respectively Lemma 4.17).

Proof. By knowing that the interference of a subtask on another one in a given time interval can never exceed the workload of this subtask in the same interval, we transform the interference

schedulability bound on subtask $\tau_{k,h}$, which was described in Corollary 4.15 into a workload bound on schedulability of the same subtask. However, the internal and external interference are based on their respective workload calculations which are provided in Lemmas 4.16 and 4.17. □

The schedulability condition described in Theorem 4.18 can be used to optimize the release jitter of each subtask. For each successor subtask, its release jitter can be calculated based on its finish time which is derived from the maximum workload. If this new release jitter is greater than its default value (which is the maximum release jitter), then the new value is discarded and the default release jitter is not modified.

4.3.4 Global Earliest Deadline First Scheduling Algorithm

In this subsection, we extend the workload analysis of work conserving algorithm to consider the case of GEDF scheduling algorithm, which assigns ready subtask jobs priorities based on their absolute local deadlines. The subtask job whose absolute local deadline is the earliest has the highest priority. Similarly to previous analyses, we provide the workload of internal subtasks such as predecessor and sibling subtasks, then workload of external subtasks. This analysis leads to a GEDF schedulability condition.

Internal Interference from Predecessor Subtasks

In Corollary 4.15 (on page 119), we provided an interference upper bound for internal subtasks (predecessors and successors) in the case of work conserving algorithm. For internal interference on $\tau_{k,h}$ when GEDF is used, the upper bound considers that the interference from predecessor subtasks is included within the release jitter $j_{k,h}$ of the subtask. Also, predecessor subtasks have higher priorities than their successors because of their dependencies. However, the optimized release jitter from Equation (4.16) (on page 119) depends on the total interference imposed on predecessor subtasks from internal and external interfering subtasks, and this depends on the scheduling algorithm.

Internal Interference from Sibling Subtasks

In the case of work conserving algorithm, we considered that all jobs of sibling subtasks, which execute in parallel with a given job of subtask $\tau_{k,h}$, may have interference on it. So, in order to calculate an upper bound on workload, we assume that all subtasks participate in the workload within the interference interval. However, this bound can be optimized in the case of GEDF, since only sibling jobs with absolute deadlines no later than $d_{k,h}$ can participate in the interference. While sibling jobs with later absolute deadlines are assigned lower priorities than $J_{k,h}$ by GEDF. We can notice that GEDF algorithm can assign priorities to sibling subtask jobs based on their relative local deadlines, because these jobs share the same activation reference. In other words, each subtask job $J_{k,i} \in sibling(\tau_{k,h})$ is activated $O_{k,i}$ time units after the release time r_k of its DAG job J_k. Hence, we can conclude that subtask job $J_{k,h}$ has higher priority than job $J_{k,i}$ if $D'_{k,h} = (O_{k,h} + D_{k,h}) \leq D'_{k,i}$.

Corollary 4.19. *If subtask $\tau_{k,i}$ is a sibling subtask of $\tau_{k,h}$ and its local relative deadline $D'_{k,i}$ is not later than $D'_{k,h}$, then its worst-case internal interference $\widehat{I}i^{k,i}_{k,h}$ on a job $J^*_{k,h}$ of subtask $\tau_{k,h}$ is defined as:*

$$\widehat{I}i^{k,i}_{k,h} \leq \widehat{W}i_{k,i}$$

where $\widehat{W}i_{k,i}$ is provided in Lemma 4.16 (on page 121).

Interference from External Subtasks

As stated earlier in Lemma 4.4 (on page 100), Bertogna et al. [32] identified the activation scenario of interfering jobs which generates the maximum workload in a given interval for GEDF. As shown in Figure 4.4, the scenario is defined when the last body job of an interfering task has its deadline at the end of the interference interval and all body jobs execute as soon as possible, while carry-in jobs execute just before their deadline. In the case of DAG tasks and as shown earlier for work conserving algorithms in the example in Figure 4.12, it is not obvious which interfering subtask in each DAG generates the maximum workload when the worst-case activation scenario is applied. Due to precedence constraints between external interfering subtasks from the same DAG, any subtask can be the reference of the interference interval (its local deadline determines the end of the interval) while the remaining subtasks execute based on their precedence constraints within the interval. Hence, it is necessary to calculate the workload

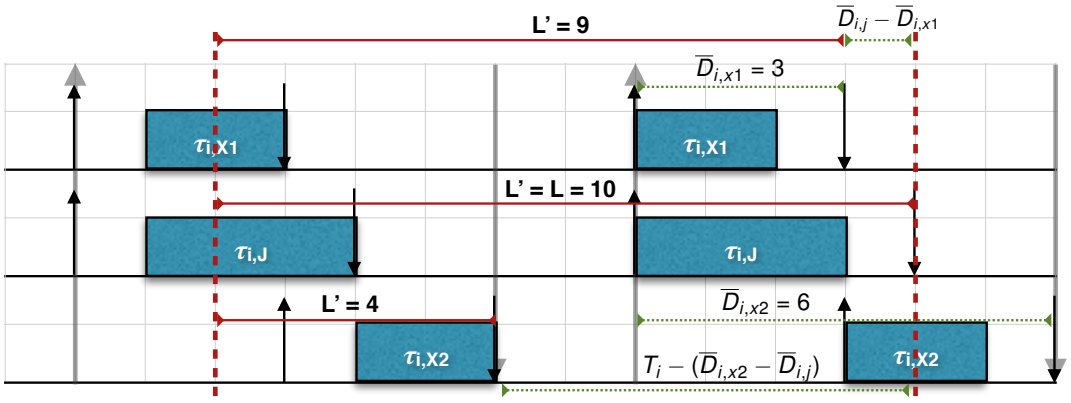


FIGURE 4.13: The workload performed by external subtasks of τ_i when subtask $\tau_{i,j}$ is the reference interval.

of each reference subtask and consider the maximum value as the worst-case workload of the interfering DAG. For a concrete example, please refer to Example 4.7.

In order to calculate the workload of an external interfering subtask in a given time interval for a given activation scenario, we have to calculate how many of its jobs execute within the interval. Figure 4.13 shows an example of the workload performed by interfering DAG τ_i on a subtask job from another DAG in an interference interval of length L. If we consider that subtask $\tau_{i,j}$ is the reference subtask in the activation scenario, then the interference interval ends at the local absolute deadline of the subtask. As shown in Figure 4.13, subtask $\tau_{i,j}$ has a body job which executes completely in the interval and a carry-in job that executes 2 time units within the interference interval. The workload $W_{i,j}$ of subtask $\tau_{i,j}$, which has a WCET $C_{i,j}$ and a period T_i, in interference interval of length L is calculated as the sum of the WCET of body jobs and carry-in job, as shown in the following equation:

$$W_{i,j}(L) = \left\lfloor \frac{L}{T_i} \right\rfloor C_{i,j} + min\left(C_{i,j}, L - \left\lfloor \frac{L}{T_i} \right\rfloor T_i\right) \tag{4.20}$$

For any subtask $\tau_{i,x1} \in \tau_i$ which has a local deadline $\overline{D}_{i,x1}$[6] that is earlier than deadline $\overline{D}_{i,j}$ of subtask $\tau_{i,j}$, its last body job in the interference interval has a local absolute deadline $(\overline{D}_{i,j} - \overline{D}_{i,x1})$ time units before the end of the interval. Any job of this subtask, whose absolute deadline is later than the end of the interference interval, is not considered in the workload analysis because the interfering job is assigned a lower priority by GEDF. In order to apply the same activation scenario of workload on $\tau_{i,x1}$, the length of the interference interval is reduced

[6]For any subtask $\tau_{i,x1}$, $\overline{D}_{i,x1} = O_{i,x1} + D_{i,x1}$.

to be $L' = L - (\overline{D}_{i,j} - \overline{D}_{i,x1})$. From Equation (4.20), the workload $W_{i,x1}(L')$ of subtask $\tau_{i,x1}$ is calculated based on the modified interference interval.

For any subtask $\tau_{i,x2} \in \tau_i$ which has a local deadline $\overline{D}_{i,x1}$ that is later than the deadline $\overline{D}_{i,j}$ of subtask $\tau_{i,j}$, the interference interval must be modified so as to apply the activation scenario. As shown in Figure 4.13, such subtask has a carry-out job that has an absolute deadline outside the interference interval and cannot participate in the workload. The last body job of $\tau_{i,x2}$ (first predecessor of its carry-out job) has an absolute deadline $T_i - (\overline{D}_{i,x2} - \overline{D}_{i,j})$ time units from the end of the original interference interval. Hence, the length of the interference interval is modified to be equal to $L' = L - (T_i - (\overline{D}_{i,x2} - \overline{D}_{i,j}))$. Similarly to the previous case, Equation (4.20) can be used to calculate the workload $W_{i,x2}(L')$ of subtask $\tau_{i,x2}$ when the interference interval is reduced.

Lemma 4.20. *When GEDF scheduling algorithm is used, the maximum workload of a DAG task τ_i on a job of subtask $\tau_{k,h}$, where $i \neq k$, in a time interval of length L, is calculated as follows:*

$$We_i(L) = \max_{\forall \tau_{i,j} \in \tau_i} \left(\sum_{\forall \tau_{i,x} \in \tau_i} \left(\left\lfloor \frac{L'}{T_i} \right\rfloor C_{i,x} + min(C_{i,x}, L' - \left\lfloor \frac{L'}{T_i} \right\rfloor * T_i) \right) \right) \tag{4.21}$$

$$\textit{where} \qquad L' = \max\left(L + [\overline{D}_{i,x} - \overline{D}_{i,j}] - \alpha * T_i, 0\right)$$

$$\alpha = \begin{cases} 0, & \textit{if } (\overline{D}_{i,j}) \geq (\overline{D}_{i,x}) \\ 1, & \textit{otherwise} \end{cases}$$

Proof. In order to calculate the maximum workload of interfering DAG task τ_i on a subtask job from another DAG task τ_k, we must apply the worst-case activation scenario on each subtask from the interfering DAG by considering the maximum workload. For each scenario, the subtasks execute based on their precedence constraints with the reference subtask of the scenario, and the workload is calculated based on the length of the considered interference interval. Equation (4.21) groups the three types of subtasks which are described above in Figure 4.13. The α parameter is defined to get rid of carry-out jobs from subtasks with local deadlines outside the interference interval. □

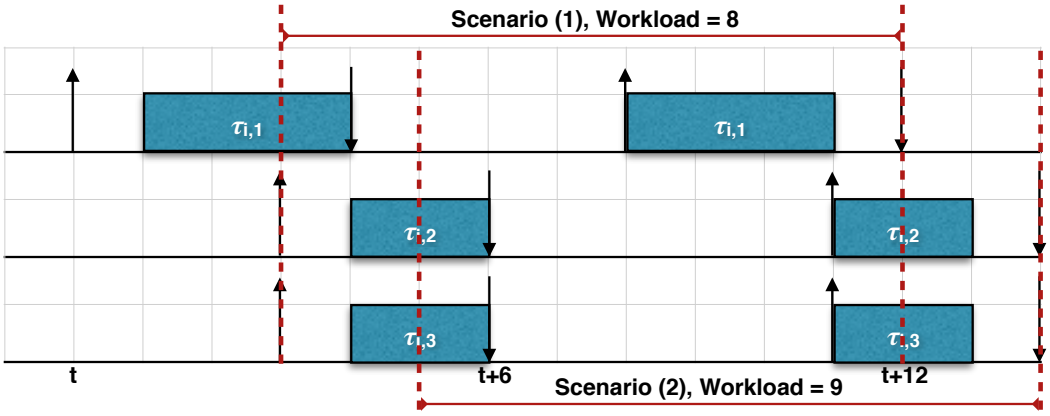


FIGURE 4.14: An example of workload analysis of external subtasks using GEDF scheduling algorithm.

Example 4.7 (Maximum workload of external subtasks)**.**

Figure 4.14 explains how to calculate the maximum workload of subtasks from an interfering DAG τ_i in an interval of length equal to $L = 9$. DAG task τ_i is presented earlier in Figure 4.12(a) (on page 123) and it consists of 3 subtasks in which two of them are identical w.r.t. their local timing parameters. We identify two activation scenarios of workload: first, the reference subtask is $\tau_{i,1}$, then we consider subtask $\tau_{i,2}$ (or $\tau_{i,3}$) to be the reference subtask.

Scenario 1: *the interference interval is defined as $[t+3, t+12)$, and it ends at the deadline of subtask job of $\tau_{i,1}$. This subtask has a carry-in and a body job in the interval. Based on Equation (4.21), subtask $\tau_{i,1}$ has a workload equal to 4. While subtasks $\tau_{i,2}$ and $\tau_{i,3}$ have carry-in jobs in the interval whose absolute deadline is at time $(t+6)$ and carry-out jobs whose absolute deadline is at time $(t+14)$. The carry-out jobs are not considered in the workload since they are assigned lower priorities by GEDF. The total workload of subtasks $\tau_{i,2}$ and $\tau_{i,3}$ is equal to 4 when we consider an interference interval of length $L' = 3$. As a result, the maximum workload of this scenario is equal to 8.*

Scenario 2: *the interfering interval is defined as $[t+5, t+14)$, and it ends at the deadline of subtasks $\tau_{i,2}$ and $\tau_{i,3}$. These subtasks are considered as the reference subtasks and they have 2 body jobs in the interval. Based on Equation (4.21), the workload of these subtasks is equal to 8. Concerning subtask $\tau_{i,1}$, its last body job in the interval has a deadline at time $(t+9)$, which is 2 time units earlier than the end of the interference interval. We modify the interval to $[t+5, t+9)$ whose length $L' = 7$. Based on Equation (4.21), the total workload of the subtask is generated from its body job and it is equal to 3. As a result, the maximum workload of this scenario is equal to 9.*

Conclusion: *the maximum workload of the interfering DAG τ_i in a time interval of length 9 is equal to 9 from Scenario 2.*

Corollary 4.21. *A DAG set τ is GEDF schedulable on m identical processors if:*

$$\forall \tau_{k,h} \in \tau_k \in \tau$$

$$\sum_{\tau_{k,i} \in sibling_{k,h}} min(\widehat{I}i_{k,h}^{k,i}, D_{k,h} - C_{k,h} - j'_{k,h}) + \sum_{\tau_i \,;\, i \neq k} min \quad (We_i(D_{k,h}), D_{k,h} - C_{k,h} - j'_{k,h})$$

$$\leq m(D_{k,h} - C_{k,h} - j'_{k,h})$$

where $\widehat{I}i_{k,h}^{k,i}$ is defined in Lemma 4.19 (on page 126).

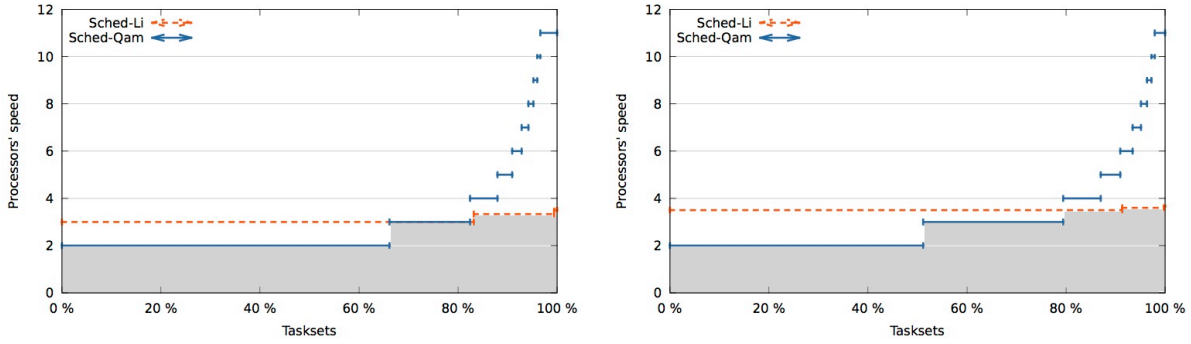
4.3.5 Simulation-Based Evaluation

In this subsection, we provide simulation results to analyze the performance of our DAG scheduling approaches and schedulability conditions when compared to other researches from the state-of-the-art. We start by analyzing the performance of internal structure GEDF schedulability condition at DAG-Level from Section 4.2 (on page 97). Then, we evaluate DAG scheduling approach at Subtask-Level for GEDF algorithm from Section 4.3. More details about our simulation tool and experimental results are provided in Chapter 5, where we compare the performance of DAG scheduling when Model Transformation and Direct Scheduling approaches are used.

Workload Analysis of DAG-Level Scheduling

In Theorem 4.7 (on page 104), we have presented a DAG-Level GEDF schedulability bound based on the maximum workload performed on a DAG task during a time interval equal to its relative deadline. This condition can be used to compute the processor speed (denoted by b in Theorem 4.7) that guarantees the schedulability of the task set. As mentioned earlier, for a particular task set, the speed of processors can be found by solving the inequality of Equation (4.6) for each DAG task. The minimum processor speed among all tasks is the speed that guarantees the schedulability of the system using GEDF.

We use simulation to evaluate the performance of our GEDF schedulability condition. Then, we compared the processor speed, obtained with our approach ,to the speed from [81] which proved



(a) Total task set utilization of 2.

(b) Total task set utilization of 4.

FIGURE 4.15: Simulation results analyzing the GEDF schedulability condition at DAG-Level.

a speed $b \geq 4 - \frac{2}{m}$ for GEDF scheduling algorithm without considering the internal structure of DAGs.

In our experiments, we considered that each DAG set consists of 50 periodic DAGs. We simulate periodic DAGs instead of sporadic DAGs because periodic activation of tasks generates the worst interference on a particular task based on the worst-case scenario. For each utilization from 1 to 4, we generate 1 million DAG sets to be simulated on m identical processors, where m is the smallest integer that is greater than system utilization U of task set τ ($m = \lceil U \rceil$).

First, we analyze the GEDF schedulability condition in Theorem 4.7. For the sake of clarity, let *Sched-Qam* denote our schedulability condition and *Sched-Li* be the condition from [81]. The simulation results for various utilization are shown in Figure 4.15. For a large range of task set utilization (more than 80% of the generated sets), task sets are schedulable using *Sched-Qam* on m processors with a lower speed than *Sched-Li*. We obtain a maximum gain is equal to 30% when $m = 2$, while the gain is between 10% and 40% when $m = 4$. We can notice that *Sched-Qam* calculates processor speed while considering a pessimistic worst-case workload scenario. Hence, high utilization task sets ($> 80\%$) require higher speed processors when compared to *Sched-Li*. For such task sets, we can use the bound from *Sched-Li* as an upper bound on workload so as to find the minimum processor speed that guarantees the schedulability of GEDF. As a result, all task sets are scheduled using GEDF on m processors with speed $\leq 4 - \frac{2}{m}$.

The results of the experimental simulations prove the importance of including the internal structure in the scheduling analysis of DAG tasks. On the average, our *Sched-Qam* condition has better performance than *Sched-Li*, and it schedules task sets on the same number of processors but with lower speed.

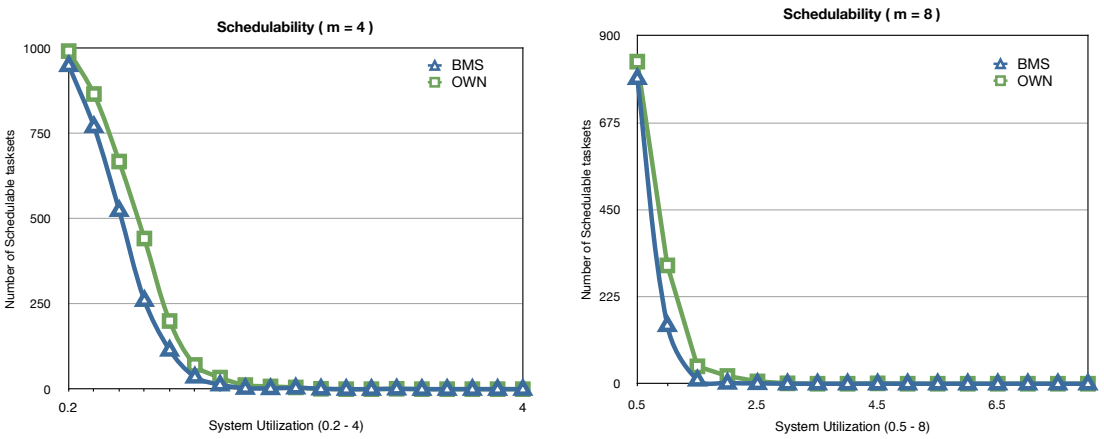


FIGURE 4.16: Simulation results analyzing the GEDF schedulability condition at Subtask-Level.

Workload Analysis of Subtask-Level Scheduling

In this subsection, we evaluate the performance of GEDF scheduling algorithm at Subtask-Level and the schedulability condition from Corollary 4.21 (on page 130). To the best of our knowledge, there is no Subtask-Level scheduling algorithms for DAGs in the state-of-the-art. Hence, we compare the schedulability of our approach with schedulability condition at DAG-Level. Bonifaci et al. in [36] provided a GEDF schedulability condition for DAG sets on m identical processors which is summarized in the following theorem:

Theorem 4.22 (from [36]). *A DAG set τ is EDF-schedulable on m unit-processors if:*

$$\forall \tau_k \in \tau \ , \ L_k \leq D_k$$

$$\left(\sum_{i:T_i \leq D_k} \frac{C_i}{T_i} + \sum_{i:T_i > D_k} \frac{C_i}{D_k} \right) \leq \frac{(m + 1/2)}{3}$$

Based on this theorem, the schedulability condition of DAG tasks depends on critical path length, total WCET, deadline and period. The structure of the subtasks is not included in the condition. Theorem 4.21 provides GEDF schedulability condition using the Subtask-Level approach. The simulation results are provided in Figure 4.16.

We generate a large number of random DAG sets, and we apply both GEDF-schedulability condition on the sets of utilization from 0 to 8. As shown in Figure 4.16, our own schedulability condition, which is denoted by *OWN*, performs better than the condition from [36], which is denoted by *BMS*. For each system utilization, our condition schedules more DAG sets than the

BMS test. The simulation results prove the importance of the internal structure of DAG tasks in the schedulability analysis.

4.4 Summary

In this chapter, we discussed the scheduling of parallel DAG tasks on multiprocessor systems using the Direct Scheduling approach. Our contributions of this chapter show the importance of the internal structure of DAGs and the execution order of subtasks in the scheduling process and analysis. We started by analyzing the subtasks and we added extra local timing parameters for each subtask based on global parameters of their DAGs and their precedence constraints. These parameters do not modify the DAG model but they are useful in adapting scheduling analysis for this model. We proved the usefulness of the internal structure of DAGs when we modified the necessary feasibility condition of processor load to consider local timing parameters of subtasks. As a result, the condition at Subtask-Level is adapted to DAG tasks.

Then, we presented a DAG-Level analysis for GEDF scheduling algorithm. Based on this approach, global parameters of DAGs are used in the scheduling process. In this work, we used the local parameters of subtasks in the schedulability analysis. Moreover, we provided a new DAG scheduling approach at Subtask-Level, in which we used the local timing parameters of subtasks in the scheduling process. As a result, a real-time scheduler becomes able to take scheduling decisions based on these parameters rather than the global parameters of the DAGs. We provided interference and workload analyses of DAGs which led to a schedulability condition for any work conserving algorithm and for GEDF scheduling algorithm in particular.

Finally, we provided simulation-based evaluations for Direct Scheduling approaches: at DAG-Level and at Subtask-Level. Mainly, we compared our GEDF schedulability conditions with the ones we found in the state-of-the-art, such as [81] and [36]. Simulation results showed the benefit of our schedulability conditions which considered internal structure of DAGs in their analyses.

Chapter 5

Experimental Analysis of DAG Task Scheduling

In this chapter, we aim at comparing both DAG scheduling approaches: the DAG stretching algorithm from the Model Transformation (from Chapter 3) and the Direct Scheduling at Subtask-Level and at DAG-Level (from Chapter 4). In Section 5.1, we provide some DAG scheduling examples so as to prove that these approaches are not comparable, i.e., there exist task sets that are schedulable using one scheduling approach and not schedulable by the other one, and vice versa.

Due to the incomparability of DAG scheduling approaches, we use extensive simulations to analyze and compare their performance to schedule random DAG tasks on multiprocessor systems. In Section 5.2, we describe our real-time simulation tool *YARTISS*, and we explain the different methods and techniques that we used to generate random DAG tasks w.r.t. timing parameters, internal structure and dependencies. This is important for performance evaluations of scheduling approaches based on the inter-subtask parallelism of DAGs. Finally in Section 5.3, we present simulation results when DAG-tasks are scheduled using the scheduling approaches with global multiprocessor algorithms. In this section, we consider two common scheduling algorithms: EDF algorithm from the Fixed-Job Priority (FJP) assignment family, and DM algorithm from the Fixed-Task Priority (FTP) assignment family.

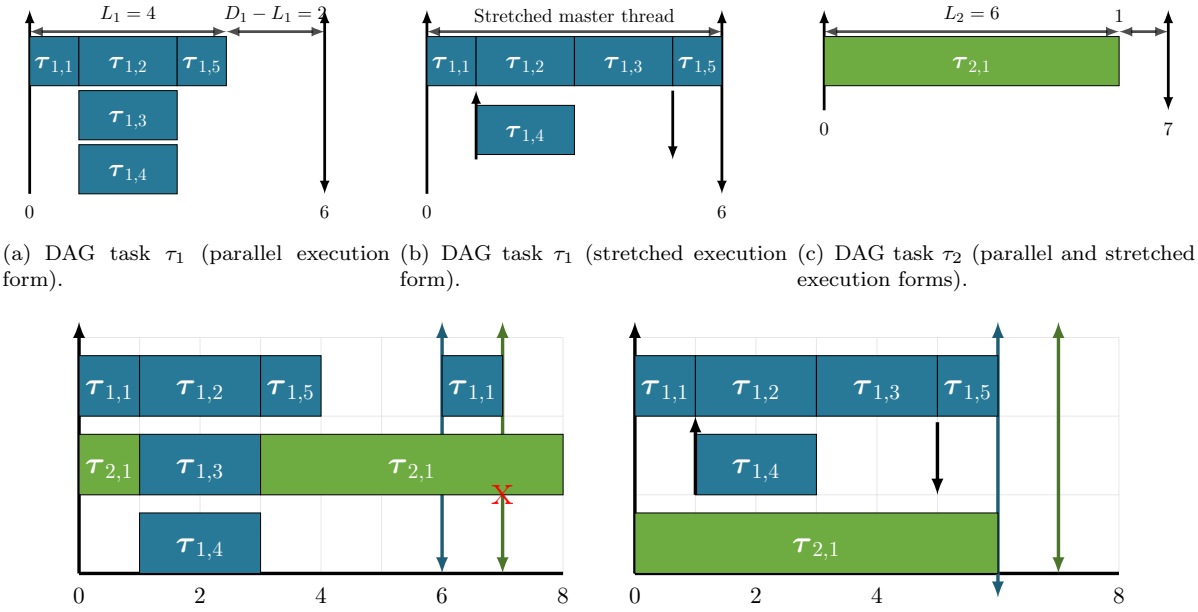
5.1 Incomparability of DAG Scheduling Approaches

Proving the incomparability of different scheduling approaches is important to show that both methods are acceptable for DAG scheduling and no one dominates the other. In this section, we provide four scheduling examples to prove the incomparability of the scheduling approaches presented in Chapters 3 and 4. Subsection 5.1.1 analyzes the DAG-Str algorithm with the Model Transformation approach, which is compared to the Direct Scheduling at DAG-Level. It contains two scheduling examples in which Example 1 is in favor of the DAG-Str algorithm, while Example 2 is in favor of the Direct Scheduling. In Subsection 5.1.2, we compare the two scheduling methods of the Direct Scheduling approach which are the DAG-Level and Subtask-Level scheduling. It contains two scheduling examples: Example 3 is in favor of DAG-Level scheduling in which scheduling decisions are taken based on global parameters of DAGs, and Example 4 is in favor of Subtask-Level scheduling in which scheduling decisions are taken based on local parameters of subtasks. In the scheduling examples, we consider two execution forms of DAG tasks based on the scheduling approach. The parallel execution form refers to the default execution scenario of a DAG, in which subtasks are activated as soon as their predecessors terminate their execution without considering any resource limitations. While the stretched execution form represents the sequential execution of subtasks after applying the DAG-Str algorithm.

In these examples, we consider global preemptive scheduling of periodic implicit-deadline DAGs on a platform of identical unit-speed processors, with DM and EDF scheduling algorithms. We consider synchronous activation scenario of task sets, in which the first jobs of tasks in the set are activated at the same time ($t = 0$ in the examples). We choose specific task sets in which the priority assignment of jobs is the same with EDF and DM during their hyper-period.

5.1.1 DAG Stretching Algorithm vs. Direct Scheduling

This subsection contains two examples, the first one shows that there exists a DAG set that is schedulable on multiprocessor systems using the DAG-Str algorithm from the Model Transformation approach, but it is not schedulable under the Direct Scheduling. As a reminder, the Model Transformation approach through stretching, forces DAG tasks to execute as sequentially as possible, then any multiprocessor scheduling algorithm can be applied afterward. While in



(a) DAG task τ_1 (parallel execution form).

(b) DAG task τ_1 (stretched execution form).

(c) DAG task τ_2 (parallel and stretched execution forms).

(d) EDF & DM scheduling of DAG set showing a deadline miss when Direct Scheduling is used.

(e) A successful EDF & DM scheduling when DAG-Str algorithm is used.

FIGURE 5.1: An example of scheduling incomparability in favor of DAG-Str when compared to Direct Scheduling.

the Direct Scheduling approach, we consider a DAG-Level scheduling in which scheduling algorithms are applied on DAG tasks based on their global parameters (such as DAG's deadline) without modifying the task model.

Example 5.1 (DAG Stretching algorithm can outperform Direct Scheduling at DAG-Level (Figure 5.1))**.**

Task Set: *In this example, we consider a DAG set τ that consists of two periodic implicit-deadline DAG tasks, where $\tau = \{\tau_1, \tau_2\}$. DAG τ_1 has 5 subtasks with a total WCET equal to 8 and a deadline equal to 6. The timing parameters of subtasks and the structure of DAG τ_1 are shown in Inset 5.1(a). It has a critical path length equal to 4 and a slack time equal to 2. Inset 5.1(a) shows its parallel form in which subtasks $\tau_{1,2}$, $\tau_{1,3}$ and $\tau_{1,4}$ execute in parallel, while Inset 5.1(b) shows the stretched structure of DAG τ_1. When DAG-Str algorithm is applied, the master thread of the DAG is stretched up to its deadline by forcing subtask $\tau_{1,3}$ to execute sequentially after subtask $\tau_{1,2}$ within the master thread. Subtask $\tau_{1,4}$ executes in parallel with an offset equal to 1 and a local relative deadline equal to 4.*

DAG task τ_2 is shown in Inset 5.1(c). It consists of a single subtask $\tau_{2,1}$ which has a WCET equal to 6 and a deadline equal to 7. Since DAG τ_2 is a sequential task, there is no difference between its parallel and stretched execution forms. The utilization of the DAG set $U(\tau) = \frac{8}{6} + \frac{6}{7} = 2.19$

which means that it needs a platform of at least 3 *unit-speed processors to execute on. In this example, we consider a platform of* 3 *processors.*

Priority Assignment: *If GDM scheduling algorithm is considered, then jobs of DAG* τ_1 *are assigned a higher priority than jobs of DAG* τ_2 *because* τ_1 *has a shorter relative deadline. In the case of GEDF, if we consider a synchronous scenario in which DAGs are released at time* $t = 0$*, then the first job of task* τ_1 *has an absolute deadline at* $t = 6$ *while the first job of* τ_2 *has an absolute deadline at* $t = 7$*. Hence, GEDF assigns the first job of* τ_1 *a higher priority than the job of* τ_2*. According to this priority assignment, jobs active in the time interval* $[0, 7)$ *have the same priorities according to GEDF and GDM.*

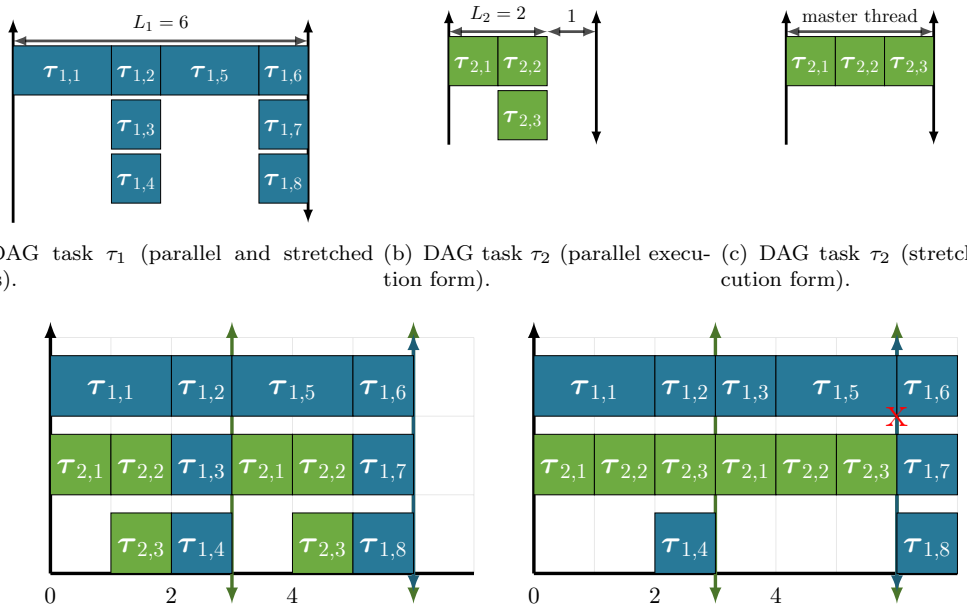
Direct Scheduling Approach: *The considered scheduling is done based on the parallel execution form of DAGs while considering priority assignment of GEDF and GDM. As shown in Inset 5.1(d), the first job of* τ_1 *executes without being interrupted since it has the highest priority. Its parallel subtasks* $\{\tau_{1,2}, \tau_{1,3}, \tau_{1,4}\}$ *occupy the* 3 *processors of the system for* 2 *time units in time interval* $[1, 3)$*. As a result, the execution of the first job of* τ_2 *is interrupted and it is delayed for* 2 *time units. Since its slack is equal to* 1 *time unit, a deadline miss occurs.*

Model Transformation Approach: *Inset 5.1(e) shows the scheduling of the same DAG set when the DAG-Str algorithm is used. Based on the structure of stretched DAG* τ_1 *from Inset 5.1(b), each job needs* 2 *unit-speed processors to execute successfully at all times, because the stretching algorithm forces subtask* $\tau_{1,4}$ *to execute sequentially within the master thread. For any given scheduling algorithm, the DAG set is schedulable on a system of* 3 *processors. DAG task* τ_1 *occupies* 2 *processors and the remaining processor is dedicated to the sequential DAG task* τ_2*.*

Conclusion: In the case of preemptive GEDF and GDM scheduling algorithms, there exists a DAG set that is schedulable on a multiprocessor system when the DAG-Str algorithm from the Model Transformation approach is used, while Direct Scheduling approach fails to schedule the same task set.

Example 5.2 (Direct Scheduling at DAG-Level can outperform DAG Stretching algorithm (Figure 5.2))**.**

Task Set: *In this example, we consider a DAG set* τ *that consists of two periodic implicit-deadline DAG tasks* $\{\tau_1, \tau_2\}$*. DAG task* τ_1 *has a deadline equal to* 6*, it consists of* 8 *subtasks and a total WCET equal to* 10*. The WCET of subtasks and the internal structure of the DAG are shown in Inset 5.2(a). The critical path length of DAG* τ_1 *is equal to* 6 *which is the same as its relative deadline. Thus, the DAG has no slack time and cannot be stretched. In order to*



(a) DAG task τ_1 (parallel and stretched forms).

(b) DAG task τ_2 (parallel execution form).

(c) DAG task τ_2 (stretched execution form).

(d) A successful EDF and DM scheduling of task set in case of Direct Scheduling.

(e) EDF and DM scheduling with a deadline miss when DAG-Str is used.

FIGURE 5.2: An example of DAG scheduling incomparability in favor of Direct Scheduling when compared to DAG-Str algorithm.

avoid a deadline miss, its subtasks must execute without any delay or interruption. DAG task τ_2 *consists of* 3 *subtasks, in which subtask* $\tau_{2,1}$ *is the source subtask of the DAG and subtasks* $\{\tau_{2,2}, \tau_{2,3}\}$ *are its successors, as shown in Inset 5.2(b). The default execution behavior of these subtasks is that both subtasks* $\tau_{2,2}$ *and* $\tau_{2,3}$ *execute in parallel. However, since DAG task* τ_2 *has a slack time equal to* 1 *time unit and its utilization is less than* 1*, then DAG-Str algorithm transforms DAG* τ_2 *into a sequential task in which all of its subtasks execute sequentially. The stretched form of DAG* τ_2 *is shown in Inset 5.2(c).*

The total utilization of the DAG set is equal to $U(\tau) = (\frac{10}{6} + 1) = 2.66$*. Hence, the task set requires an execution platform of at least* 3 *processors to be feasible.*

Priority Assignment: *with GDM scheduling, jobs of DAG* τ_2 *are assigned a higher priority than jobs of DAG* τ_1 *because* τ_2 *has a shorter relative deadline. In the case of GEDF, if we consider a synchronous scenario in which DAGs are released at time* $t = 0$*, the first job of task* τ_2 *has an absolute deadline at* $t = 3$ *while the first job of* τ_1 *has an absolute deadline at* $t = 6$*. As a result, GEDF assigns the job of DAG* τ_2 *a higher priority. Regarding the second job of DAG* τ_2*, its absolute deadline is equal to the deadline of the first job of DAG* τ_1*, hence, priorities are assigned arbitrarily. In this example, we consider that the DAG with the shortest relative*

deadline is assigned higher priority as tie breaking rule. According to this priority assignment, DAG jobs have the same priorities according to GEDF and GDM scheduling algorithms.

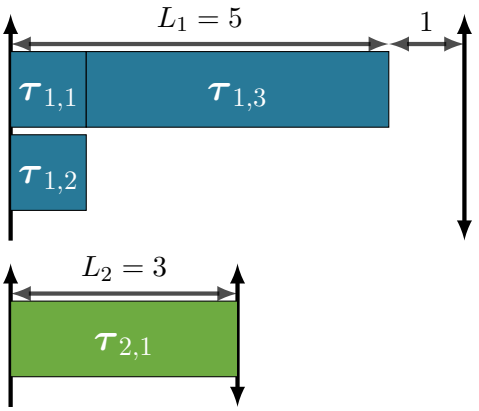
Direct Scheduling Approach: *In Inset 5.2(d), we consider the global preemptive scheduling of the DAG set on a system of* 3 *identical processors using the Direct scheduling approach. First DAG jobs are activated at time* $t = 0$ *and the job of DAG* τ_2 *has the highest priority. Subtask* $\tau_{2,1}$ *executes in interval* $[0, 1)$ *and its successors* $\tau_{2,2}$ *and* $\tau_{2,3}$ *execute in parallel and occupy two processors of the system in* $[1, 2)$*. Similarly, the second job of DAG* τ_2 *executes in parallel in time interval* $[3, 5)$*. According to this scheduling, the first job of DAG task* τ_1 *executes without interruption and all of its subtasks execute without any delay. As shown in Inset 5.2(d), the synchronous DAG set is schedulable using GDM and GEDF scheduling algorithms.*

Model Transformation Approach: *When DAG-Str algorithm is used, subtasks of DAG* τ_2 *are forced to execute sequentially as a sequential thread even if there are available processors in the system for the subtasks to execute in parallel. The first job of* τ_2 *has a higher priority according to GDM and GEDF, then it occupies a single processor by itself in time interval* $[0, 3)$*, as shown in Inset 5.2(e). As a result, subtasks of the first job of DAG* τ_1 *are blocked in this time interval and one of them (subtask* $\tau_{1,3}$ *in the example) is delayed for* 1 *time unit and is forced to execute in time interval* $[3, 4)$ *instead of* $[2, 3)$*. DAG task* τ_1 *has no slack time, then a deadline miss happens as shown in the figure.*

Conclusion: In the case of preemptive GEDF and GDM scheduling algorithms, there exists a DAG set that is schedulable on a multiprocessor system with the Direct Scheduling approach, while it is unschedulable with the DAG stretching algorithm. Based on Examples 5.1 and 5.2, both scheduling approaches are not comparable and no one dominates the other.

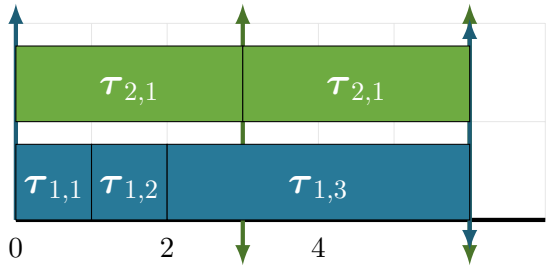
5.1.2 Direct Scheduling: DAG-Level vs. Subtask-Level

In this subsection, we compare the Direct Scheduling of DAG tasks on homogeneous multiprocessor systems at DAG and Subtask levels. As described in Chapter 4, the difference between these scheduling approaches is the timing parameters of DAGs and their subtasks which are used by the real-time scheduler. Here, we prove using scheduling examples that both Direct Scheduling approaches are not comparable and that there is no dominance relationship between them. As in the previous subsection, we consider GDM and GEDF scheduling algorithms to schedule DAG tasks.

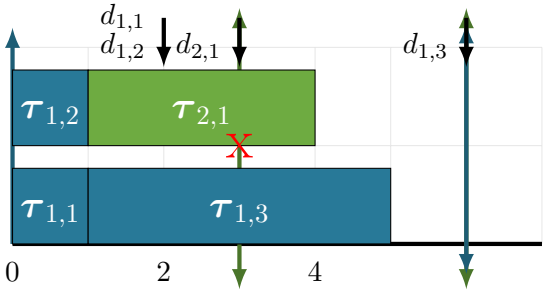


	DAG-Level	**Subtask-Level**
τ_1	$\tau_{1,1} = (0, 1, 6, 6)$	$\tau_{1,1} = (0, 1, 2, 6)$
	$\tau_{1,2} = (0, 1, 6, 6)$	$\tau_{1,2} = (0, 1, 2, 6)$
	$\tau_{1,3} = (0, 4, 6, 6)$	$\tau_{1,3} = (1, 4, 5, 6)$
τ_2	$\tau_{2,1} = (0, 3, 3, 3)$	$\tau_{2,1} = (0, 3, 3, 3)$

(a) DAG tasks τ_1 and τ_2 (with global and local parameters).



(b) A successful GEDF and GDM scheduling of task set at DAG-Level.



(c) GEDF and GDM scheduling fails at Subtask-Level.

FIGURE 5.3: An example of DAG scheduling incomparability in favor of DAG-Level scheduling when compared to Subtask-Level scheduling.

Example 5.3 (DAG-Level scheduling can outperform Subtask-Level Scheduling (Figure 5.3))**.**

Task Set: *In this example, we consider a DAG set τ that consists of two periodic implicit-deadline DAG tasks $\{\tau_1, \tau_2\}$. DAG task τ_1 has a deadline equal to 6 and consists of 3 subtasks whose total WCET is equal to 6. The critical path length of DAG τ_1 is equal to 5 and its slack time is equal to 1 time unit. Subtask $\tau_{1,1}$ and $\tau_{1,2}$ are source subtasks and subtask $\tau_{1,3}$ is their successor. DAG task τ_2 consists of a single subtask with a WCET equal to 3 which is the same as its deadline. The internal structure of the DAGs is shown in Inset 5.3(a).*

The total utilization of the DAG set is equal to $U(\tau) = (\frac{6}{6} + \frac{3}{3}) = 2$. Hence, the DAG set requires an execution platform of at least 2 processors to be feasible. The global and local timing parameters of subtasks of each DAG in the set are shown in Inset 5.3(a). A subtask $\tau_{i,j} \in \tau_i$ is characterized by an offset, a WCET, a relative deadline and a period which are represented respectively by a quadruple $(O_{i,j}, C_{i,j}, D_{i,j}, T_{i,j})$[1]. At DAG-Level, each subtask inherits the offset, deadline and period of its respective DAG task and is characterized by a specific WCET.

[1]Local timing parameters of subtasks are described in details in Section 4.1 (on page 89).

DAG-Level Scheduling Approach: *We consider a global preemptive scheduling of synchronous DAG set on a platform which consists of* 2 *unit-speed processors. If DM scheduling algorithm is considered, then jobs of DAG* τ_2 *are assigned a higher priority than jobs of DAG* τ_1 *because* τ_2 *has a shorter relative deadline. This priority is inherited by the subtasks, i.e., subtask* $\tau_{2,1}$ *has higher priority than subtasks* $\tau_{1,1}$, $\tau_{1,2}$ *and* $\tau_{1,3}$. *The same priority assignment is applied on the first job of DAGs in the case of EDF. When DAGs are released at time* $t = 0$, *then the first job of* τ_2 *has an absolute deadline at* $t = 3$ *and it has a higher priority than the first job of* τ_1 *whose absolute deadline is at time* $t = 6$. *We assume that the second job of* τ_2 *has higher priority with an absolute deadline at* $t = 6$ *(ties are broken arbitrarily).*

Inset 5.3(b) shows the scheduling of this DAG set on 2 *processors. According to the priority assignment, the first job of task* τ_2 *executes in time interval* $[0, 3)$ *without any interruption from the other active jobs. While subtasks* $\tau_{1,1}$ *and* $\tau_{1,2}$ *are forced to execute sequentially in time interval* $[0, 2)$ *due to their priorities. According to this scheduling, both DAGs execute sequentially without any deadline miss.*

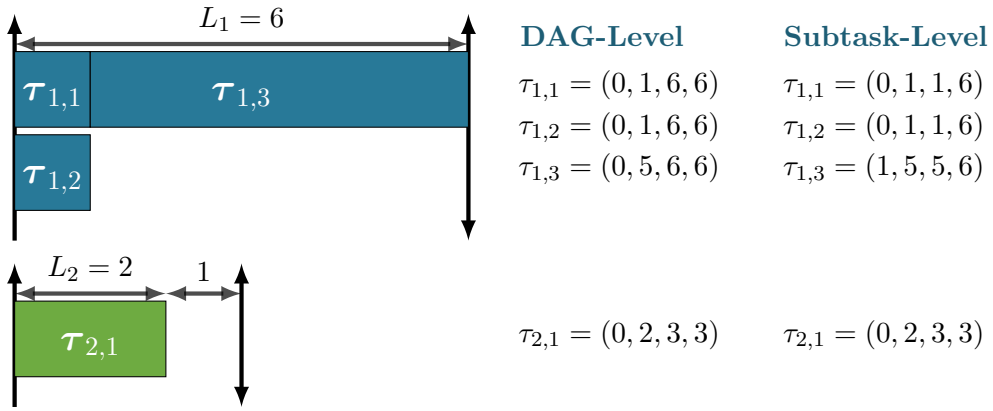
Subtask-Level Scheduling Approach: *At Subtask-Level scheduling, local timing parameters are assigned to subtasks and they are shown in Inset 5.3(a). Subtasks of the same DAG may be assigned different priorities based on their local parameters. In the case of DM, subtasks* $\tau_{1,1}$ *and* $\tau_{1,2}$ *are assigned the highest priorities since they have the smallest relative deadline. Subtask* $\tau_{2,1}$ *is assigned a lower priority and* $\tau_{1,3}$ *has the lowest priority. The same priority assignment is applied in the case of EDF to the first job of DAGs when they are released at time* $t = 0$.

In Inset 5.3(c), subtasks $\tau_{1,1}$ *and* $\tau_{1,2}$ *execute as soon as they are released (at* $t = 0$*) and they occupy both processors of the system in time interval* $[0, 1)$. *According to this, the first job of subtask* $\tau_{2,1}$ *is blocked and its execution starts at* $t = 1$ *which leads to a deadline miss.*

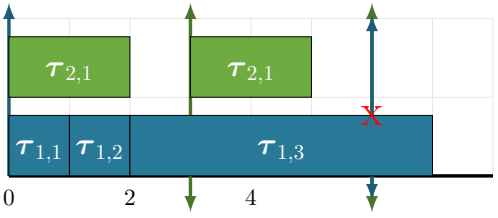
Conclusion: In the case of preemptive GEDF and GDM scheduling algorithms, there exists a DAG set that is schedulable on a multiprocessor system when it is scheduled at DAG-Level, while it is unschedulable on the same platform using Subtask-Level scheduling.

Example 5.4 (Subtask-Level scheduling can outperform DAG-Level Scheduling (Figure 5.4))**.**

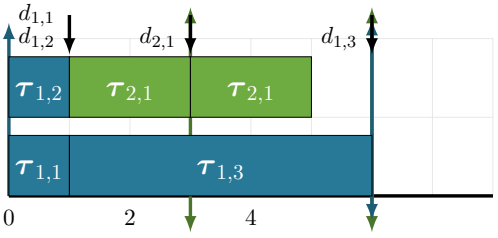
Task Set: *In this example, we consider a DAG set* τ *that consists of two periodic implicit-deadline DAG tasks* $\{\tau_1, \tau_2\}$. *DAG task* τ_1 *has a deadline equal to* 6, *and consists of* 3 *subtasks and a total WCET equal to* 7. *The critical path length of DAG* τ_1 *is equal to* 6, *hence the DAG has no slack time. Subtask* $\tau_{1,1}$ *and* $\tau_{1,2}$ *are the source subtasks and subtask* $\tau_{1,3}$ *is their*



(a) DAG task τ_1 and τ_2 (with global and local parameters).



(b) GEDF and GDM scheduling with a deadline miss when DAG-Level scheduling is used.



(c) A successful GEDF and GDM scheduling of DAG set when Subtask-Level scheduling is used.

FIGURE 5.4: An example of DAG scheduling incomparability in favor of Subtask-Level scheduling when compared to DAG-Level.

successor. DAG task τ_2 consists of a single subtask with a WCET equal to 2 and a deadline equal to 3. The internal structure of the DAGs in the set is represented in Inset 5.4(a).

The total utilization of the DAG set is equal to $U(\tau) = (\frac{7}{6} + \frac{2}{3}) = 1.83$. Hence, the set requires an execution platform of at least 2 processors to be feasible. The global and local timing parameters of subtasks of each DAG in the set are shown in the figure. As described in Example 3, a subtask $\tau_{i,j}$ is characterized by a quadruple $(O_{i,j}, C_{i,j}, D_{i,j}, T_{i,j})$.

DAG-Level Scheduling Approach: *We consider a global preemptive scheduling of synchronous DAG set on a system which consists of 2 unit-speed processors. If GDM scheduling algorithm is considered, then jobs of DAG τ_2 are assigned a higher priority than jobs of DAG τ_1 because τ_2 has a shorter relative deadline. This is applied on all subtasks of the DAG, i.e., subtask $\tau_{2,1}$ has higher priority than $\{\tau_{1,1}, \tau_{1,2}, \tau_{1,3}\}$. The same priority assignment is applied on the first job of DAGs when GEDF is used. When DAGs are released at time $t = 0$, then the first job of τ_2 has an absolute deadline at $t = 3$ and has a higher priority than the first job of τ_1 whose absolute deadline is at time $t = 6$. We assume that the second job of τ_2 has higher priority with absolute deadline at $t = 6$ arbitrarily.*

Inset 5.4(b) shows the scheduling of this DAG set on 2 processors. According to the assigned priorities, the first job of DAG τ_2 executes in time interval $[0, 2)$ without any interruption from other jobs, while subtasks $\tau_{1,1}$ and $\tau_{1,2}$ are forced to execute sequentially in time interval $[0, 2)$. According to this scheduling, subtask $\tau_{1,3}$ starts its execution at time $t = 2$ and cannot complete it before its deadline at time $t = 6$, which leads to a deadline miss.

Subtask-Level Scheduling Approach: *The local parameters which are used in Subtask-Level scheduling are shown in Inset 5.4(a). In the case of GDM, subtasks $\tau_{1,1}$ and $\tau_{1,2}$ are assigned the highest priorities since they have the smallest relative deadline value of 1. Then subtask $\tau_{2,1}$ has a lower priority and finally $\tau_{1,3}$ is assigned the lowest priority. The same priority assignment is applied in the case of GEDF on the first jobs of DAGs if they are released at time $t = 0$.*

As shown in Inset 5.4(c), subtasks $\tau_{1,1}$ and $\tau_{1,2}$ execute as soon as they are released and they occupy both processors of the system in time interval $[0, 1)$. Accordingly, jobs of subtask $\tau_{2,1}$ execute in time interval $[1, 3)$ and $[3, 5)$ and subtask $\tau_{1,3}$ executes in parallel on the other processor of the system in time interval $[1, 6)$. Hence, DAG task is scheduled successfully on 2 processors using Subtask-Level scheduling.

Conclusion: In the case of preemptive GEDF and GDM scheduling algorithms, there exists a DAG set that is schedulable on a multiprocessor system when Subtask-Level scheduling is used, but it is unschedulable when DAG-Level approach is used. We conclude from Examples 5.3 and 5.4 that both scheduling approaches are not comparable and no one dominates the other.

5.2 Simulation-Based Evaluation

Based on the previous scheduling examples, we concluded that the provided DAG scheduling approaches are not comparable, and it is not clear which approach outperforms the other. In order to evaluate their schedulability performance, we compare them through extensive simulations of randomly-generated DAG tasks on multiple processors. The use of simulation-based evaluations is common in real-time analysis to give an indication on the performance of the algorithms. Simulation is used to check whether a set of tasks respects its temporal constraints when a specific algorithm is used, or to evaluate the efficiency of a new approach when compared with other algorithms from the state of the art.

In real-time systems, there are many simulation tools that vary in their characteristics and features, such as MAST [70], Cheddar [110, 111], STORM [118] and FORTAS [47]. However,

many factors led researchers to implement their own simulation tools without depending on existing ones, such as the lack of a standard simulator, the difficulty of extending an existing tool to include new features and models and the lack of documentation. Moreover, our work required the implementation of specific real-time tasks of the DAG model.

In this section, we present *YARTISS*[39] which is a free open-source simulation tool written in Java for real-time multiprocessor scheduling. Similarly to other simulation tools, *YARTISS* is designed to simulate the scheduling of real-time algorithms and evaluate their performance. However, we focused during its design on providing a generic simulation tool and an easy-to-use modular design in which new modules can be easily added without the need to decompress, edit nor recompile existing parts. *YARTISS* can be used to simulate the execution of large scale data sets of different task models on multiprocessor platforms and show the scheduling results visually within the simulator. Moreover, many task models are already implemented in *YARTISS* including the DAG model with its associated scheduling algorithms and schedulability test. *YARTISS* provides energy-aware simulations of task sets in which energy consumption is one of the scheduling parameters of the tasks.

In Subsection 5.2.1, we start by providing a development history of *YARTISS*, and we introduce its previous versions that led to its development. Then in Subsection 5.2.2, we briefly describe the main features of the simulator while concentrating on the implementation of the DAG model and its scheduling simulation. Finally in Section 5.3, we present simulation-based evaluations of DAG scheduling approaches and we compare and evaluate their performance with GEDF and GDM scheduling algorithms.

5.2.1 Simulation Tool: *YARTISS*

YARTISS is a fourth simulation tool developed by our real-time research team at University of Paris-Est Marne-la-Vallée during the last few years. Each one of these tools was built for a specific purpose at that time and required a long period of time to be developed. In this subsection, we briefly present the development history of these tools and we show their features, functionality and even their limitations that led to the development of *YARTISS*. We gained experience from these previous developments to present a generic simulation tool that can be used in the future by other researchers.

Our first version of a real-time simulator was called RTSS [88] which was developed between 2005 and 2008. RTSS was initially developed to test some scheduling algorithms on uniprocessor

systems in order to handle temporal fault tolerance, such as preemptive FP, EDF and D^{OVER} [76] algorithms. It was extended to include sporadic tasks with *Polling and Deferrable* task servers and their algorithms [89, 90]. RTSS suffered from some problems: some modifications had been done based on certain assumptions and special execution behaviors without proper documentation. Also, RTSS was not easily extendable, i.e., modifying a class in the simulator results affected the execution of other classes. Moreover, although the tool was initially programmed in Java, bash scripts were regularly used to launch the simulation process and to generate readable files as outputs.

Between 2008 and 2011, RTMSim [52], which stands for **R**eal-**Time** **M**ultiprocessor **Sim**ulator, was developed based on RTSS. As the name indicates, RTMSim targeted multiprocessor simulations of restricted migration [53] scheduling. The decision of designing a new simulator was taken because of the poor documentation of RTSS. However, the core of RTSS was included in RTMSim but many implementations of scheduling algorithms were lost.

Third try was made in early 2011. RTSS v2 [88] was developed as a rebuild of RTSS so as to handle energy consuming tasks and energy-harvesting systems [42]. Unfortunately, RTSS v2 suffered from the same problems of the original tool RTSS such as the poor documentation and the lack of modularity and usability features which are necessary for large scale simulations. Moreover, RTSS v2 targeted uniprocessor systems and the extension towards multiprocessors was not trivial.

Based on all of these development experiences, *YARTISS* was designed as a generic modular tool that can be extended easily to include new models and algorithms. By default, it included different features such as generic task models (for both independent and dependent tasks), generic multiprocessor execution platforms and even scheduling simulation with energy constraints. Another important feature in *YARTISS* is the friendliness of its user interface which can be used to generate large data sets, launch the simulation process and produce scheduling results as readable files. In the following subsection, we provide a description of the main features of *YARTISS* followed by a comparison of DAG scheduling approaches.

5.2.2 Simulation Features and Functionality

YARTISS is designed to provide two main features that are necessary for real-time simulation. The first is the scheduling simulation of a given task set when a scheduling algorithm is used.

The second feature is the comparison of several scheduling algorithms (policies)[2] on a large scale and when different scheduling scenarios are considered. However, both features require a third important feature which is the generator of task sets that are used for simulation and they have to be random for reliability reasons.

In the remainder of this subsection, we list the main features of *YARTISS*, and we describe briefly their implementation and functionality.

Single Task set Simulation

This is a basic function which simulates the scheduling of a given task set when it executes on multiprocessor systems with a specific scheduling algorithm. The task set can be loaded as an Extensible Markup Language (XML) file into the simulator either through the Graphical User Interface (GUI) or by using task set generators. The simulation can be configured and launched using the GUI, then scheduling results are shown on the different views of *YARTISS*. The simulation parameters include the task set characteristics, the number of processors, the scheduling algorithm and the energy profile. Figure 5.5 shows the GUI of the simulator and its principle views: the scheduling diagram based on the available processors of the platform on the top-left side, and the time diagram of each task in the set on the bottom-left side.

[2]Both terms are used interchangeably in this subsection.

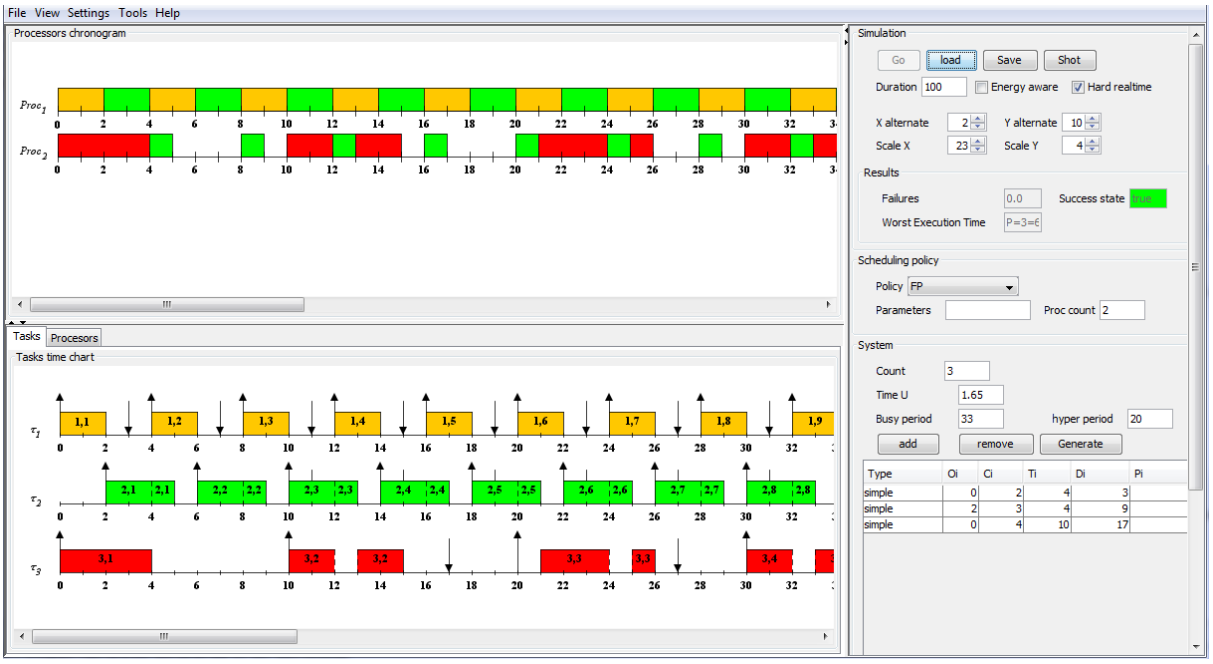


FIGURE 5.5: The multiprocessor view of YARTISS simulation tool.

By default, *YARTISS* considered global preemptive scheduling of tasks on homogeneous unit-speed multiprocessor systems. However, the extension to partitioned scheduling and/or non-preemptive scheduling can be easily done by adding their respective modules and classes in *YARTISS*.

Uniprocessor/Multiprocessor Platforms

Unlike previous simulation tools (RTSS and RTSS v2), *YARTISS* is designed to support multiprocessor scheduling, in which the number and type of processors are entered as simulation parameters. Uniprocessor systems, in which the number of processors is equal to 1, can be considered as a special case of multiprocessor systems. Task set generators are developed to be compatible with multiprocessor scheduling and they are randomly generated with system utilization greater than 1. Furthermore, each processor has a type which determines its speed (execution rate) as in the case of uniform or heterogeneous systems.

Energy Profile

YARTISS is designed to simulate energy production and consumption in real-time systems with different energy profiles. This feature, which is not common in regular real-time scheduling simulators, allows the user to model an energy harvester (such as a battery or a capacitor) with limited or unlimited capacity. A renewable energy source can be also modeled using a charging function.

Task Models and Generators

YARTISS offers an open architecture to facilitate the integration of different task models. Its current version contains two models, the first one is the independent sequential task model [85] with energy parameters. As described in Section 1.1.1 on page 3, each task τ_i is characterized by its WCET C_i, its period T_i and its relative deadline D_i, in addition to E_i which denotes its worst case energy consumption. The second model is the DAG task model which belongs to the dependent parallel task models. As described earlier in Section 1.3.4 on page 19, a graph task is characterized by a set of single-threaded subtasks with precedence constraints.

Performing large-scale scheduling simulations requires a large data set of tasks. In order to avoid biased results and ensure credibility, task sets should be randomly generated and their

Algorithm 5.1 The UUniFast Algorithm (from [33])

Input: $U(\tau), n$
Output: $vectU$ ▷ An array of utilization of each task τ_i in the set τ.

```
sumU = U(τ)
for i = 1 : n − 1 do
    nextSumU = sumU × rand^(1/(n−i))
    vectU(i) = sumU − nextSumU
    sumU = nextSumU
end for
vectU(n) = SumU
```

timing parameters should be varied. *YARTISS* provides the ability to choose a task set generator which defines the various timing parameters of tasks and whether they are periodic/sporadic and implicit/constrained/arbitrary deadline tasks. The default generator in *YARTISS* is based on the *UUniFast-Discard algorithm* [33] adapted to energy constraints and parallel tasks (utilization is greater than 1) coupled with a hyper-period limitation technique [62].

We start by explaining the generation of independent sequential tasks. For a given task set τ, the generator has two input parameters, the system utilization $U(\tau)$ and the number of tasks in the set denoted by n. The UUnifast-Discard algorithm uniformly distributes system utilization on all the tasks of the set with $\mathcal{O}(n)$ complexity. As shown in Algorithm 5.1, the UUniFast-Discard algorithm generates an array of n random task utilization, in which each element represents a task utilization U_i of $\tau_i \in \tau$, where $0 < U_i \leq 1$ (sequential tasks) and $\sum_{i=1}^{n} U_i \leq U(\tau)$.

After defining the task utilization, which is equal to $U_i = \frac{C_i}{T_i}$, the timing parameters of each task τ_i (WCET and period) can be derived based on the utilization. However, the period parameter is chosen based on a hyper-period limitation technique. It is known that a periodic task set repeats its task arrival pattern after an interval called the hyper period, which is equal to the Least Common Multiple (LCM) of task periods. For synchronous task sets (respectively asynchronous) with static scheduler, their schedulability is determined by verifying the response time of each job on a period of length equal to the hyper period (respectively slightly greater than the hyper period) [61]. For dynamic schedulers (synchronous or asynchronous), the period may go up to twice the hyper period [79]. Hence, the value of LCM is affected by the increase of task periods in the set. In order to limit the length of the hyper period during task generation in *YARTISS*, which results in a reduction in the simulation interval, we use the hyper-period limitation technique from [62]. The idea of the technique is to generate n periods $\{T_1, T_2, ..., T_n\}$ for each task in the set in a way to bound their resulting LCM. The algorithm uses a matrix $\mathcal{M}$ representing primes and their probabilistic distribution. A period is calculated as the multiple

of random number from each line in the matrix. For example, if we consider a matrix $\mathcal{M}$ which consists of 5 primes $(2, 3, 5, 7, 11)$, its structure and probabilistic distribution are provided as follows:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 & 2 & 4 & 4 & 4 & 8 & 16 & 16 \\ 1 & 3 & 3 & 9 & 9 & 9 & 27 & & \\ 1 & 5 & 5 & 25 & 25 & 25 & & & \\ 1 & 1 & 7 & 7 & 7 & 49 & & & \\ 1 & 1 & 1 & 11 & 11 & & & & \end{pmatrix}$$

The largest period, that can be possibly generated from $\mathcal{M}$, is equal to $(16 \times 27 \times 25 \times 49 \times 11 = 5821200)$, which represents the largest possible hyper period of the task set. Hence, by choosing the primes values of the matrix, we can limit the maximum hyper period of the generated task set, and respectively, the simulation interval.

Based on the UUniFast-Discard and the hyper-period limitation algorithms, the utilization and period of each task are derived. Based on these values, the relative deadline and WCET are randomly calculated. The deadline D_i is derived based on the type of generated task sets. In the case of implicit-deadlines, we consider that $D_i = T_i$. While the deadline of constrained-deadline tasks is less than or equal to the period, where $D_i \leq T_i$. There is no relation between the deadline and the period in the case of arbitrary-deadline tasks. Finally, the WCET C_i of task τ_i is calculated based on the utilization and period and it is strictly less than the deadline of the task $(1 \leq C_i \leq D_i)$.

As stated earlier, *YARTISS* is an energy-aware simulator in which task sets can be feasible regarding energy constraints as well as timing constraints. An extra parameter can be assigned to tasks to represent their energy consumption during execution, which is the Worst-Case Energy Consumption (WCEC) using UUniFast-Discard algorithm. *YARTISS* presents the generated data sets in XML files. An example of such files is shown in Listing 5.1 which contains a data set of a single task set (tag <`tasks`>) of two tasks (tag <`task`>). The timing parameters of each task are coded into the XML file. In *YARTISS*, there exist XML writer and reader classes which are responsible for encoding generated task sets in XML format and extract timing parameters from files to construct task sets.

The DAG task generator is more challenging than the one of the independent sequential tasks. Inter-subtask parallelism and dependencies between subtasks complicate the generation process. In our DAG model, subtasks are defined as sequential threads and their basic timing parameters

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <policy name="PFP_ASAP" nbParams="0"/>
  <simulation endValue="100" nbProc="1"/>
  <energyProfile E0="0" Emax="50" Emin="0" pr="3"/>
  <tasks nbTasks="2" type="Fixed Priority">
    <task deadline="3" firstRelease="0" period="8" priority="1" type="
    simple" wcec="10" wcet="2"/>
    <task deadline="9" firstRelease="0" period="10" priority="2" type="
    simple" wcec="3" wcet="3"/>
  </tasks>
</dataset>
```

LISTING 5.1: An example of XML code representing an independent sequential task set

are similar to the ones defined in the previous generated model. However, their precedence constraints are randomly generated to determine the execution structure of the DAG. The DAG generator uses the same generation techniques and algorithms of the independent sequential model which determine the utilization and period of each DAG. Based on the total system utilization and the number of tasks in the set, each DAG τ_i is assigned a utilization U_i which can be greater than 1 since it is a parallel task, and the total WCET C_i, the relative deadline D_i and the period T_i of the DAGs are defined randomly. WCET and precedence constraints of subtasks are defined based on the following parameters:

- **Maximum number of subtasks (`MAX_SUBTASKS`)**: it is defined for each DAG set as an upper bound on the number of subtasks in the DAG. This value is important to determine the size of DAGs which affects the probability of its inter-subtask parallelism. Generally, DAG tasks, whose number of subtasks is large, tend to have more internal parallelism and more precedence constraints between their subtasks than smaller DAGs.

  Additionally, the minimum number of subtasks (`MIN_SUBTASKS`) is calculated so as to ensure a feasible generation of DAGs. Its value is calculated when we consider that each subtask $\tau_{i,j}$ in DAG τ_i has a WCET $C_{i,j}$ equal to its relative deadline D_i, which is the maximum execution time that can be assigned to any subtask so as to be feasible. Then, the `MIN_SUBTASKS` is equal to $\left\lceil \frac{C_i}{D_i} \right\rceil$. For each DAG task τ_i in the set, its number of subtasks n_i is equal to rand(`MIN_SUBTASKS`, `MAX_SUBTASKS`). If random generation of subtask timing parameters leads to `MIN_SUBTASKS` greater than `MAX_SUBTASKS`, then the generation process is repeated until this relation becomes true.

- The **WCET** $C_{i,j}$ of each subtask $\tau_{i,j}$ is calculated using the UUniFast-Discard algorithm, where the total WCET C_i of the DAG and the number of subtasks n_i are its inputs. We

bound the value of $C_{i,j}$ of each subtask to ensure system feasibility by using the following $C_{i,j}^{max}$ and $C_{i,j}^{min}$ bounds:

- Any sequential subtask of DAG τ_i cannot exceed the deadline of the DAG. Hence, $C_{i,j}^{max} = D_i$.
- The minimum WCET $C_{i,j}^{min}$ of subtask $\tau_{i,j}$ is calculated when each subtask $\tau_{i,k}$ in the DAG, which is not assigned a WCET yet, is considered to have a WCET $C_{i,k}$ equal to $C_{i,k}^{max}$. This bound is necessary to ensure the feasibility of generated subtasks. For example, let us consider a DAG task τ_i with the following timing parameters: a total WCET $C_i = 6$, a relative deadline $D_i = 4$ and two subtasks $\{\tau_{i,1}, \tau_{i,2}\}$. If subtask $\tau_{i,1}$ is assigned a WCET equal to $C_{i,1} = 1$, then the remaining WCET available for subtask $\tau_{i,2}$ is equal to 5 which is greater than the deadline of the DAG and the subtask is not feasible on a unit-speed processor. Hence, $C_{i,1}^{min}$ has to be at least $(6 - 4) = 2$ time units to ensure feasibility.

- The **probability factor of directed relations** ρ between subtasks, where $0 < \rho < 1$. If ρ is close to 0, then the probability of creating a directed relation between any two subtasks in the DAG is large. This probability is reduced when ρ moves closer to 1. In order to get rid of cyclic dependencies between subtasks, we use a triangular matrix $\mathcal{R}$ whose entries of ones and zeros are randomly generated based on the probability factor ρ. For each DAG task τ_i, $\mathcal{R}$ is a square matrix of size n_i and all of its entries under the main diagonal are zeros. We consider that the main diagonal entries are zeros so that a subtask does not have a precedence relation with itself. The remaining entries represent the precedence relations between subtasks and they are either zeros or ones. For DAG τ_i, if entry $\mathcal{R}_{j,k} = 1$, then we create a precedence relation from subtask $\tau_{i,j}$ to $\tau_{i,k}$. If it is zero, then there is no relation between these two subtasks. An example of the triangular matrix $\mathcal{R}$ of a DAG τ_i of 4 subtasks is considered as follows:

$$\mathcal{R} = \begin{array}{c} \\ \tau_{i,1} \\ \tau_{i,2} \\ \tau_{i,3} \\ \tau_{i,4} \end{array} \begin{array}{c} \begin{array}{cccc} \tau_{i,1} & \tau_{i,2} & \tau_{i,3} & \tau_{i,4} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

```
<?xml version="1.0" encoding="UTF-8"?>
  ...
  <tasks nbTasks="1" type="Fixed Priority">
    <task deadline="10" firstRelease="0" nbSubtasks="2" period="10"
    priority="1" type="graph" wcee="0" wcet="5">
      <subtask children="1" deadline="10" firstRelease="0" index="0"
    localDeadline="7" nbProc="1" parents="" period="10" priority="1" type="
    subtask" wcet="2"/>
      <subtask children="-1" deadline="10" firstRelease="2" index="1"
    localDeadline="10" nbProc="1" parents="0" period="10" priority="1" type
    ="subtask" wcet="3"/>
    </task>
  </tasks>
```

LISTING 5.2: An example of an XML file describing the DAG Tasks

In this example, subtask $\tau_{i,1}$ has two successors $\tau_{i,2}$ and $\tau_{i,3}$ since $\mathcal{R}_{1,2} = \mathcal{R}_{1,3} = 1$, while there is no directed relations between subtask $\tau_{i,1}$ and subtask $\tau_{i,4}$. Similarly, subtask $\tau_{i,4}$ is the successor of subtasks $\tau_{i,2}$ and $\tau_{i,3}$.

Using these parameters, DAG sets are randomly generated and each DAG task is assigned a WCET, a period, a relative deadline and a set of random subtasks with precedence constraints. As in the case of the sequential task set generator, each DAG set can be encoded with an XML file so as to be used repeatedly in the simulation of different scheduling algorithms. An example of a DAG XML file is shown in Listing 5.2, which represents a data set of a single task set (tag `<tasks>`) which contains a single DAG task (tag `<task>`). This DAG consists of two subtasks (tag `<subtask>`), in which the first subtask is a parent of the second one (represented by attributes `parents` and `children`). The subtask tag has other attributes such the WCET, deadline, first release time so as to represent their timing parameters.

Scheduling Algorithms and Schedulability tests

The main purpose of a real-time simulator is to compare the performance and efficiency of different scheduling algorithms. During the design of *YARTISS*, great attention was paid to make it as generic as possible so as to facilitate the integration of new scheduling algorithms.

The addition of new scheduling algorithms to *YARTISS* follows the UML scheme described in Figure 5.6. The scheduling algorithm (policy) mainly depends on its corresponding `taskset` class which contains the necessary functions for priority assignment of the policy. In the case of GEDF scheduling policy for example, we added a new class called the `GlobalEDFTaskset` which extends the `ITaskSet` interface. Then, another class for the scheduling policy called

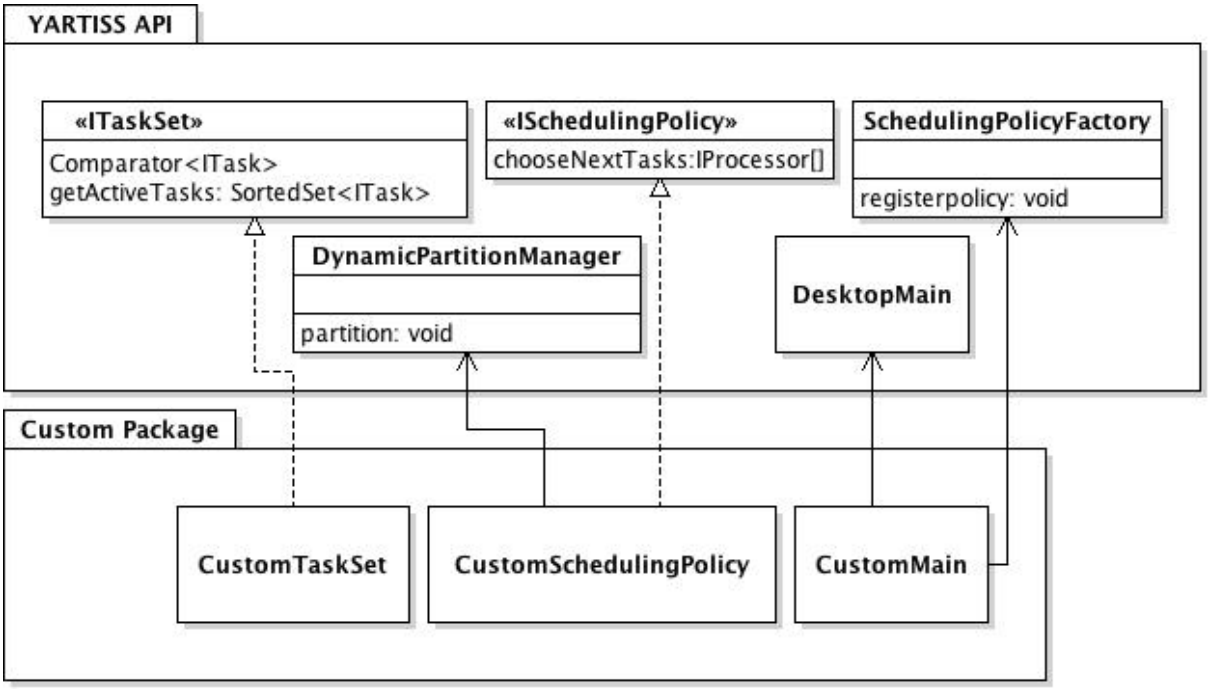


FIGURE 5.6: A UML diagram describing the addition of a new scheduling policy.

`GlobalEDFGraphPolicy` was implemented to extend the `ISchedulingPolicy` interface. This class has the sorted `taskset` class as input and is mainly responsible for choosing the m highest-priority active jobs to be executed in the system, where m is the number of processors. Selected jobs are assigned to processors with a desired scheduling characteristic, such as global or partitioned scheduling.

The same principle is applied in the case of schedulability tests, which are defined as schedulability conditions that determine the schedulability of a given task set when an algorithm is used. Usually, schedulability tests do not require simulating the scheduling of tasks in the system. The schedulability conditions are applied on generated task sets and the results are represented visually by a text file or a curve to show whether each task set is schedulable or not. The schedulability tests can be used as a performance indication of different real-time scheduling algorithms. *YARTISS* is designed to facilitate the addition of new schedulability tests, similar to the addition of scheduling algorithms.

5.3 Simulation Results of DAG Scheduling Approaches

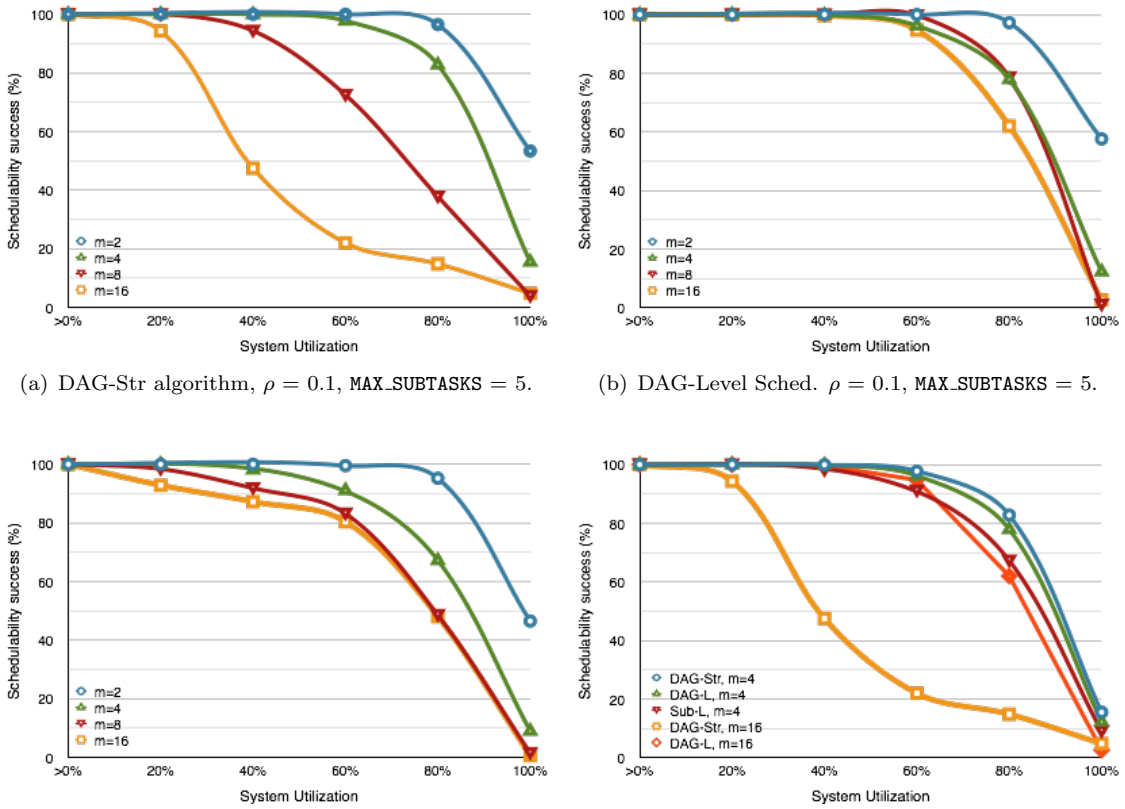
In this section, we describe the schedulability results of simulation experiments conducted to compare the performance of DAG scheduling approaches, which are the DAG Stretching algorithm, the Direct scheduling at DAG-Level and at Subtask-Level. We use *YARTISS* to generate large data sets of random DAGs and to simulate their scheduling during one hyper period in the synchronous scenario. We consider two preemptive algorithm to schedule DAG tasks, after applying the scheduling approaches, which are the GEDF and GDM algorithms. Simulation results are respectively shown in Subsections 5.3.1 and 5.3.2.

In order to generate random DAG tasks with various internal structure characteristics, we consider a probability factor ρ that varies from 0.1 to 0.9 in steps of 0.1 which affects the internal parallelism of DAGs. We also consider a `MAX_SUBTASKS` value that varies from 5 to 12 in steps of 1 which affects the size of the DAGs. Based on these parameters and for a given execution platform of m identical processors, where $m \in \{2, 4, 8, 16\}$, we consider task set utilization that varies from 0.2 to 1.0 times the number of processors in steps of 0.2. For each utilization value, we generate $50,000$ DAG sets each of 10 periodic implicit-deadline DAG tasks.

We schedule the generated data sets by varying the DAG scheduling approaches and algorithms. Then, we study simulation results to analyze the effect of the following variations on the schedulability of DAG tasks:

- The number of processors in the system.
- The size of DAG tasks.
- The probability of internal parallelism of DAG tasks.

We analyze these variations in the case of GEDF and GDM scheduling algorithms. Since these two algorithms belong to different priority assignment families, we compare their effect on the scheduling approaches. We identify through simulation results the compatible priority assignment for parallel or sequential form of execution for DAG tasks.



(a) DAG-Str algorithm, $\rho = 0.1$, `MAX_SUBTASKS` $= 5$.

(b) DAG-Level Sched. $\rho = 0.1$, `MAX_SUBTASKS` $= 5$.

(c) Subtask-Level Sched. $\rho = 0.1$, `MAX_SUBTASKS` $= 5$.

(d) All algorithms, $\rho = 0.1$, `MAX_SUBTASKS` $= 5$.

FIGURE 5.7: Effects of the number of processors on the performance of DAG scheduling approaches when GEDF is used.

5.3.1 Simulation Results for GEDF Scheduling Algorithm

The effect of the number of processors on DAG schedulability

In this experiment, we scheduled DAG sets using the three scheduling approaches with preemptive GEDF scheduling algorithm while concentrating on varying the number of processors in the system. The other simulation parameters regarding the structure of DAGs are fixed for each simulation sets.

The simulation results of different scheduling approaches are shown in Figure 5.7 which consists of 4 insets. The x-axis in the Figure denotes the percentage of task set utilization w.r.t. the number of processors in the system, while the y-axis denotes the percentage of schedulable task sets. The effect of varying the number of processors on the schedulability of DAGs was independent from the other simulation parameters. Hence, we show the simulation results when the probability factor ρ is equal to 0.1 (high probability of internal parallelism) and `MAX_SUBTASKS` is equal to 5. The same remarks and conclusions can be extended to other scenarios and variations.

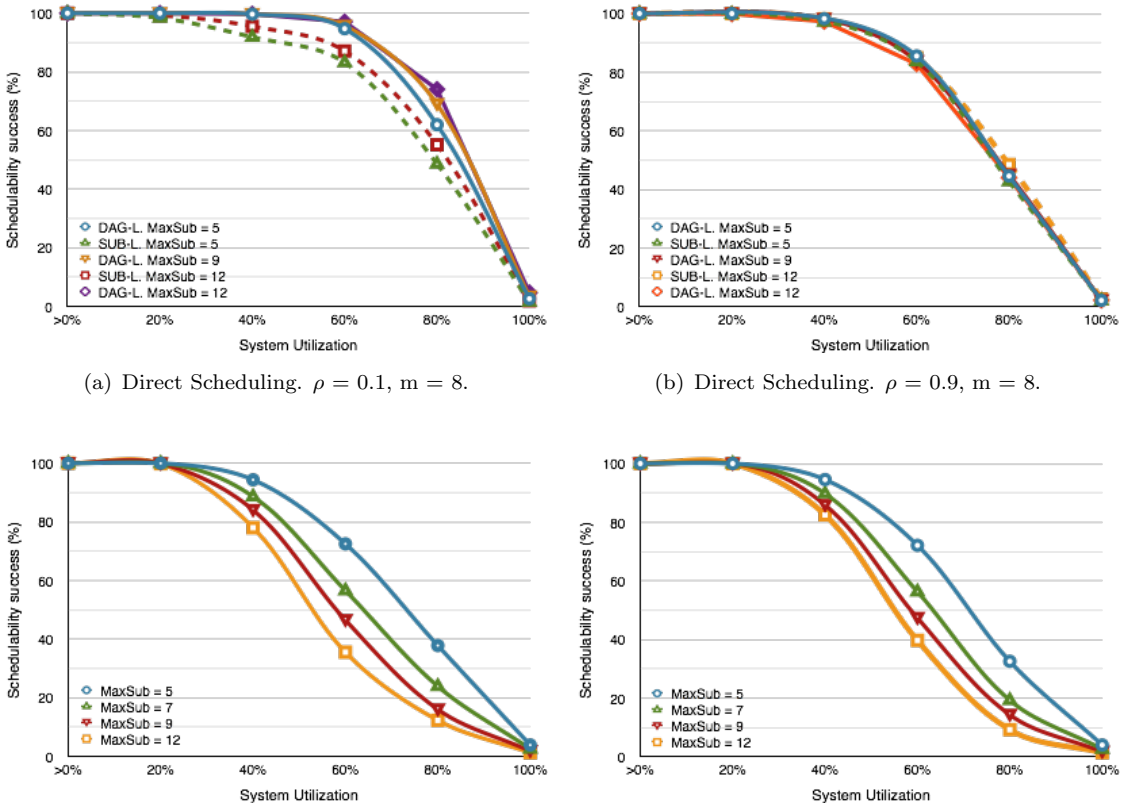
In general, we notice that the performance of all DAG scheduling approaches decreases when the number of processors of the system is increased. However, the performance of the DAG-Str algorithm is more affected by varying the number of processors than the direct scheduling approaches (at DAG-Level and Subtask-Level). Inset 5.7(a) shows the simulation results when DAG-Str is used. For $m = 2$, we notice that the schedulability percentage of stretched DAG sets is almost 100% for an utilization up to 80%. The schedulability percentage drops to around 50% for an utilization equal to 100%. However, the schedulability performance degrades when the number of processors is increased. When $m = 16$, less than 50% of task sets are schedulable when their utilization is greater than 40% of the number of processors.

Direct Scheduling approaches at DAG and subtask levels behave in the same manner but with better schedulability, as shown in Insets 5.7(b) and 5.7(c). The simulation results show that DAG-Level scheduling successfully schedules more than 60% of DAG sets whose utilization is no more than 80% of the number of processors, for any number of processors. The schedulability percentage drops to around 50% of schedulable DAG sets when Subtask-Level scheduling is used. As shown in Inset 5.7(d), the performance of all scheduling approaches is relatively similar when the number of processors is small (although the DAG-Str algorithm has the best performance). However, when we consider $m = 16$, there is a big difference in performance between the DAG-Str algorithm and the Direct scheduling. In conclusion, when GEDF is used to schedule DAG tasks on execution platforms of large number of processors, it is better to consider Direct scheduling approaches rather than DAG-Str algorithm for large number of processors.

The effect of the size of DAGs on DAG schedulability

Figure 5.8 shows the simulation results of scheduling approaches as a function of the number of processors. We consider that `MAX_SUBTASKS` varies from 5 to 12 while the other simulation parameters are fixed. In these experiments, we choose an average number of processors equal to 8 and we analyze the simulation results based on the minimum and maximum probability of internal parallelism of DAG tasks ($\rho = 0.1$ and $\rho = 0.9$ respectively).

Generally, we notice that when the size of DAGs is increased, the schedulability performance decreases with different rates based on the considered scheduling approach. In Inset 5.8(a) where $\rho = 0.1$, the performance of DAG-Level scheduling (solid lines in the figure) degrades when the size of DAGs is increased, but it is better than the schedulability of Subtask-Level (dashed lines). When $\rho = 0.9$ in Inset 5.8(b), the schedulability of Direct scheduling approach is similar to both DAG and subtask levels and it is not affected by the variation of DAG size.



(a) Direct Scheduling. $\rho = 0.1$, m = 8.

(b) Direct Scheduling. $\rho = 0.9$, m = 8.

(c) DAG-Str algorithm. $\rho = 0.1$, m = 8.

(d) DAG-Str algorithm. $\rho = 0.9$, m = 8.

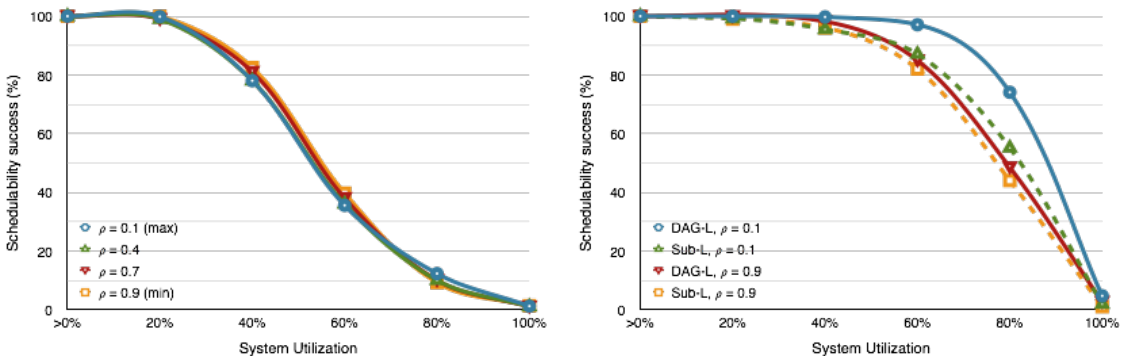
FIGURE 5.8: The effect of size of DAG tasks on the performance of DAG scheduling approaches when GEDF is used.

In both cases, Direct scheduling approach successfully schedules more than 40% of DAG sets whose utilization is less than or equal to 80% of the number of processors.

Unfortunately, the performance of the DAG-Str algorithm degrades more than the Direct Scheduling approach when DAG sizes are increased. As shown in Insets 5.8(c) and 5.8(d), the DAG-Str algorithm schedules more DAG sets of small size. When `MAX_SUBTASKS` is equal to 5, more than 30% of DAG sets, whose utilization is not greater than 80%, are schedulable. The schedulability percentage drops to around 10% of the same DAG sets when `MAX_SUBTASKS` is equal to 12. Moreover, we notice that the performance of DAG-Str algorithm is not much affected by the level of internal parallelism of DAG tasks which is represented by the probability factor ρ.

The effect of internal parallelism of DAGs on schedulability

In this subsection, we analyze the effect of parallelism probability on the schedulability of DAGs using different scheduling approaches. We present simulation results after varying the probability factor ρ from 0.1 to 0.9 in steps of 0.1. When ρ is close to zero, DAG tasks are more



(a) DAG-Str algorithm, `MAX_SUBTASKS` = 12, m = 8.

(b) Direct Scheduling, `MAX_SUBTASKS` = 12, m = 8.

(c) DAG-Level Sched. `MAX_SUBTASKS` = 12, m = 8.

(d) Subtask-Level Sched. `MAX_SUBTASKS` = 12, m = 8.

FIGURE 5.9: The effect of probability of internal parallelism on the performance of DAG scheduling approaches when GEDF is used.

likely to have many dependencies between their subtasks, while a factor close to 1 means that subtasks tend to execute independently within their DAGs. In this experiment set, we choose execution platforms with number processors equal to 8 and large DAG tasks with `MAX_SUBTASKS` equal to 12. The simulation results are shown in Figure 5.9.

As shown in Inset 5.9(a), the performance of DAG-Str algorithm is not affected by the ρ factor. This can be explained by the scheduling of stretched DAG tasks transformed into independent sequential threads without considering the internal parallelism. Similarly, the performance of Direct Scheduling approach on a subtask-level is not much affected as shown in Inset 5.9(d). However, the performance of DAG-Level scheduling improves when the probability of internal dependencies increases. As shown in Inset 5.9(c), around 50% of DAG tasks with utilization less than 80% are schedulable when parallelism parameter $\rho = 0.9$. The percentage of schedulable DAG sets raises up to more than 70% of schedulable DAG tasks with the same utilization when $\rho = 0.1$.

In Inset 5.9(b), we compare the performance of Direct Scheduling approach at DAG-Level and

Subtask-Level. The simulation results show that DAG-Level scheduling has better performance than Subtask-Level scheduling specially when the probability of internal dependencies is high ($\rho = 0.1$). However, when the probability drops to 0.9, then the performance of both approaches is close, although that DAG-Level is slightly better.

Conclusion:

Based on the simulation results, we conclude that Direct Scheduling approach performs better than DAG-Str algorithm when preemptive GEDF algorithm is used. Also, DAG-Level scheduling has better schedulability than Direct Scheduling at Subtask-Level.

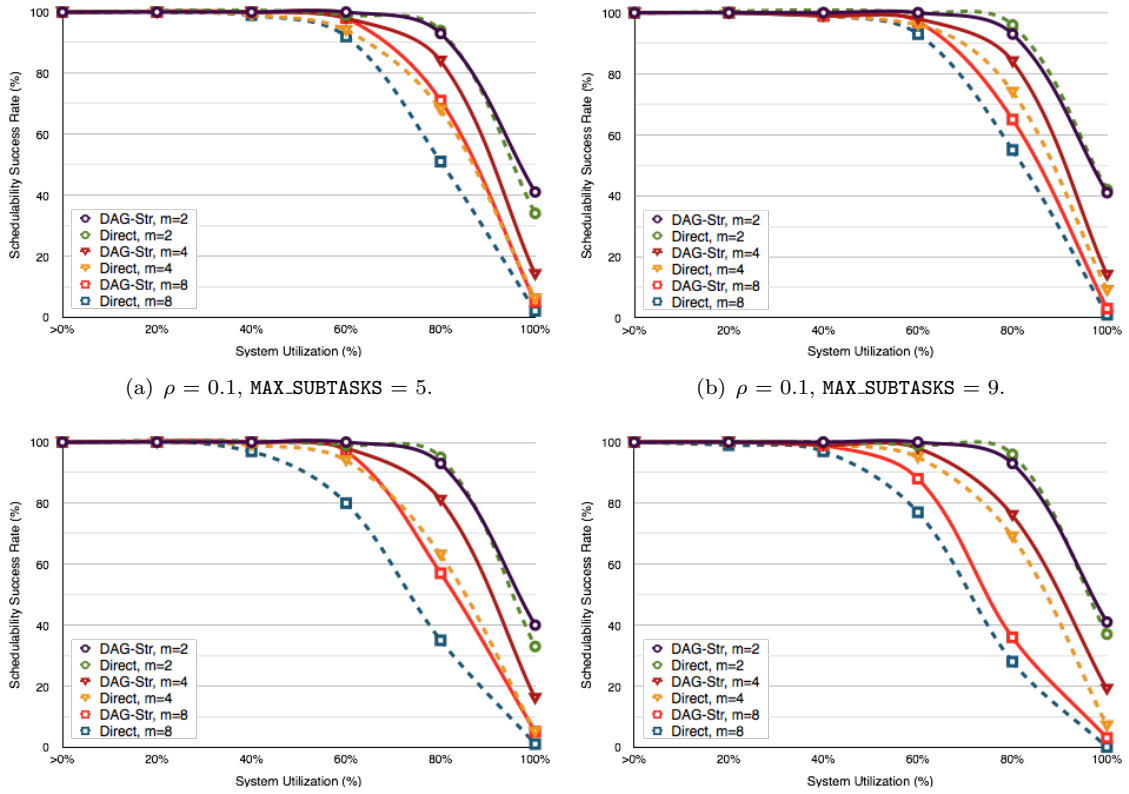
5.3.2 Simulation Results for GDM Scheduling Algorithm

In this subsection, we study the performance of priority assignment on the scheduling algorithm used for the DAG-scheduling approaches. We present simulation results of preemptive GDM scheduling algorithm and we compare the performance of DAG-Str algorithm and the Direct Scheduling approach at DAG-Level, by varying the number of processors in the system, the size of DAG tasks and the probability of inter-subtask parallelism.

The effect of the number of processors on DAG schedulability

Figure 5.10 shows simulation results which compare the DAG-Str algorithm (solid lines) and DAG-Level scheduling (dashes lines) when changing the number of processors in the systems. In general, the schedulability rates of both algorithms decrease with the increase of number of processors in the system. Also, we notice that the schedulability of both algorithms is almost identical when $m = 2$. However, DAG-Str algorithm performs better than DAG-Level scheduling when the execution platform consists of larger number of processors. This indicates that GDM is more compatible with DAG-Str algorithm and performs better than GEDF scheduling algorithm.

Moreover, we notice that changing the size of DAGs (`MAX_SUBTASKS`) does not affect the schedulability of DAG scheduling approaches. For example, the schedulability performance of approaches in Inset 5.10(a), where `MAX_SUBTASKS` is equal to 5, is almost identical to the results when `MAX_SUBTASKS` is increased to 9 in Inset 5.10(b). The size of DAG tasks does not seem to affect much the schedulability performance of algorithms. However, when ρ factor is equal to 0.1, the scheduling approaches has better schedulability on large number of processors when compared to the case where ρ is equal to 0.9.



(a) $\rho = 0.1$, `MAX_SUBTASKS` $= 5$.

(b) $\rho = 0.1$, `MAX_SUBTASKS` $= 9$.

(c) $\rho = 0.9$, `MAX_SUBTASKS` $= 5$.

(d) $\rho = 0.9$, `MAX_SUBTASKS` $= 9$.

FIGURE 5.10: Simulation results comparing the performance of DAG-Str algorithm and DAG-Level scheduling while varying the number of processors in the system.

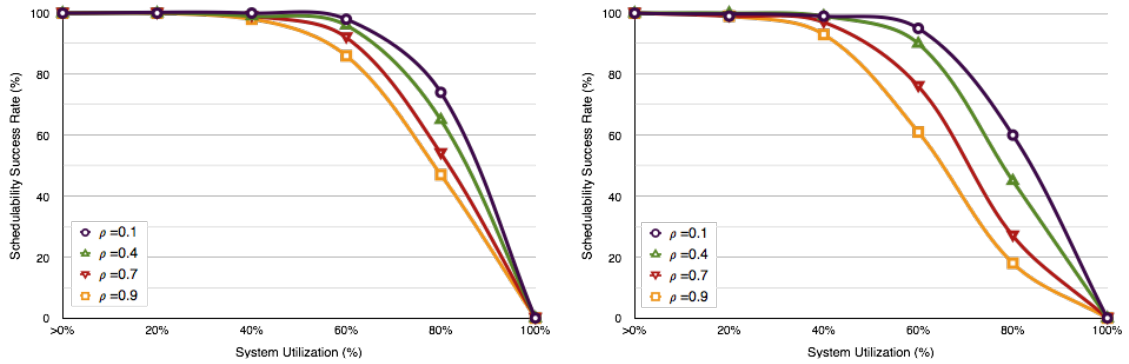
Simulation analysis for DAG-Level scheduling

In the case of Direct Scheduling approach at DAG-Level, the schedulability performance is better for high probability of parallelism ($\rho = 0.1$) rather than for low probability, as shown in Figure 5.12. However, the performance degrades more in the case of DAG tasks with large size, where `MAX_SUBTASKS` is equal to 12.

Insets 5.12(a) and 5.12(b) show the degradation of schedulability performance of Direct scheduling approach when we vary the number of processors in the system. Similarly, the schedulability rates degrade more in the case of low probability of internal parallelism.

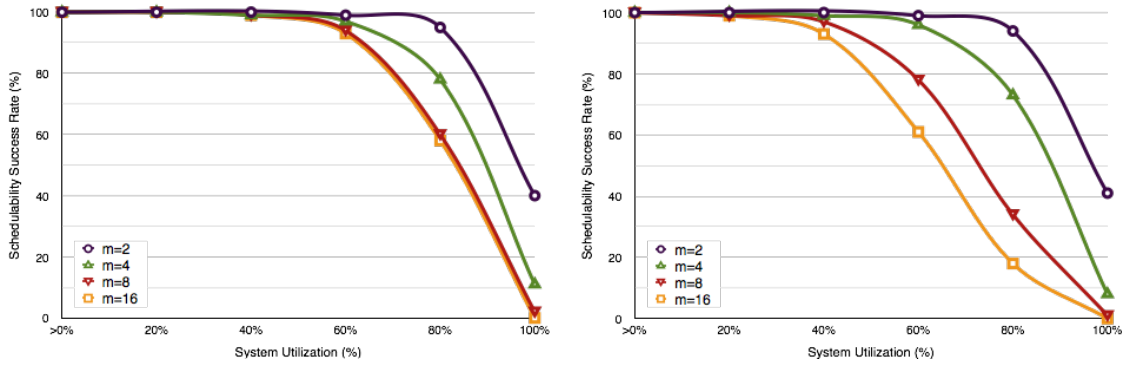
Simulation analysis for DAG-Str algorithm

As shown in Inset 5.13(a), the schedulability of DAG-Str algorithm is not affected by the change of probability of parallelism in DAG tasks when their sizes are relatively small. However, when DAG-Str is applied on DAGs with larger sizes, where `MAX_SUBTASKS` is equal to 12, the



(a) m = 16, `MAX_SUBTASKS` = 5. (b) m = 16, `MAX_SUBTASKS` = 12.

FIGURE 5.11: Simulation results comparing the schedulability performance of DAG-Level scheduling while varying the probability of internal parallelism.



(a) $\rho = 0.1$, `MAX_SUBTASKS` = 12. (b) $\rho = 0.9$, `MAX_SUBTASKS` = 12.

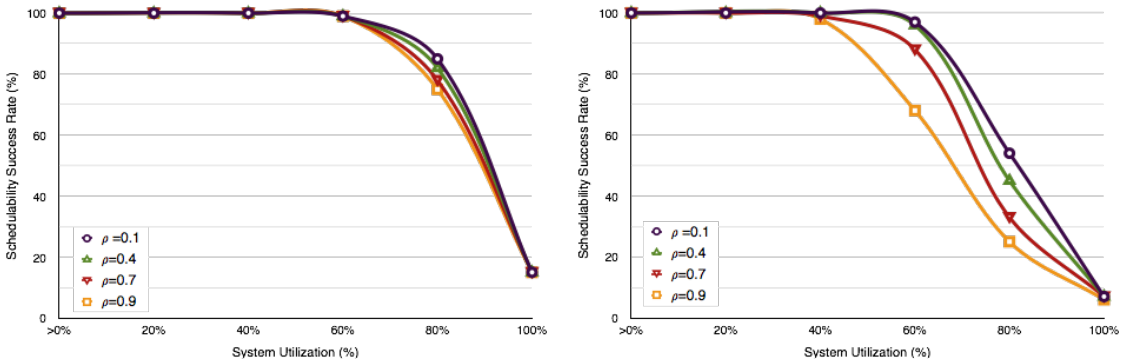
FIGURE 5.12: Simulation results comparing the schedulability performance of DAG-Level scheduling while varying the number of processors in the system.

schedulability gap increases when the number of processors is increased, as shown in Inset 5.13(b).

Similarly to the simulation results of DAG-Level scheduling, the performance of the DAG-Str algorithm on DAG tasks with probability factor ρ is equal to 0.1, is better than the ones with low probability, as shown in Insets 5.14(a) and 5.14(b).

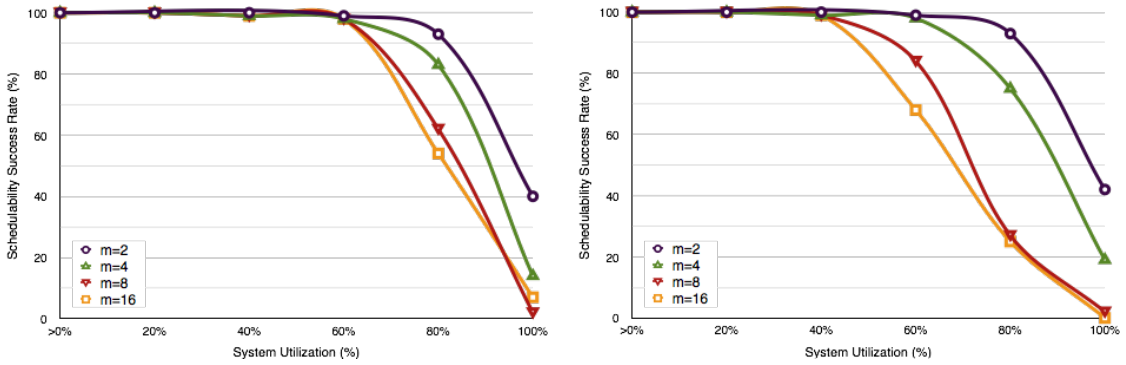
5.4 Summary

In this chapter, we presented scheduling examples of different DAG approaches to prove their incomparability. We compared DAG-Str algorithm from the Model Transformation approach with Direct Scheduling at DAG-Level. Then, we compared DAG-Level and Subtask-Level scheduling from the Direct Scheduling approach. Due to this incomparability, we performed extensive simulations so as to study and analyze their schedulability performance. We presented *YARTISS*,



(a) m = 16, `MAX_SUBTASKS` = 5. (b) m = 16, `MAX_SUBTASKS` = 12.

FIGURE 5.13: Simulation results comparing the schedulability performance of DAG-Str algorithm while varying the probability of internal parallelism.



(a) $\rho = 0.1$, `MAX_SUBTASKS` = 12. (b) $\rho = 0.9$, `MAX_SUBTASKS` = 12.

FIGURE 5.14: Simulation results comparing the schedulability performance of DAG-Str algorithm while varying the number of processors in the system.

which is a generic real-time simulation tool for multiprocessor systems that contains parallel DAG task generator. In order to ensure reliability of our simulation results, we generated a large number of DAG sets that vary in size and in the probability of internal parallelism.

The simulation results are provided when global preemptive EDF and DM scheduling algorithms are used while varying the number of processors of the systems. We analyzed these results in the different scheduling scenarios and approaches. In general, we concluded that Direct Scheduling approach is more compatible with GEDF scheduling algorithm, while DAG-Str algorithm is compatible with GDM.

Chapter 6

Conclusion and Perspectives

Hard real-time scheduling has been studied thoroughly for many years and it is still an interesting problem. On uniprocessor systems, the main objective of a real-time scheduler is to determine a feasible allocation of task sets on their execution platform without any deadline miss. Due to some physical constraints, manufacturers tend nowadays to enhance the performance of execution platforms by increasing the number of their processing units, and multiprocessor systems recently became an acceptable choice in industrial applications. However, multiprocessor scheduling problem is more challenging than uniprocessors since it concerns with optimizing the utilization of several processors during scheduling.

Moreover, parallel task models are of growing importance. They are introduced as a development of software design which takes advantage of multiprocessor architecture. Many parallel task models have been introduced in real-time systems. In this thesis, we are interested in the Multi-threaded model, in which a parallel task consists of a number of threads that execute either in parallel or sequentially based on the decisions of the real-time scheduler. The Directed Acyclic Graph (DAG) is an example of such model in which a DAG task consists of a set of subtasks that execute under precedence constraints. Managing the internal dependencies of DAG tasks adds another difficulty to the multiprocessor real-time scheduling problem.

6.1 List of Contributions

In this thesis, we are interested in the global preemptive scheduling problem of real-time DAG tasks on multiprocessor systems, and we aim at answering the following question:

How DAG tasks can be scheduled on multiprocessor real-time systems? Should they be directly scheduled on multiprocessor systems while considering their internal dependencies? Or is it better to avoid this parallel model and transform DAGs into independent sequential forms that are easier to schedule?

To this end, we provided the following contributions:

Model Transformation Approach: DAG Stretching Algorithm

In Chapter 3, we provided a DAG Stretching (DAG-Str) algorithm based on the Model Transformation approach for periodic implicit-deadline DAG tasks. In this algorithm, the parallel DAG model is converted into independent sequential tasks so as to avoid the internal dependencies during the scheduling process. Briefly, DAG tasks are transformed into a collection of independent sequential threads, and each thread is assigned an intermediate offset and deadline to ensure its independent execution. The real-time scheduling is done based on these intermediate parameters using any common multiprocessor scheduling algorithm.

In order to analyze the schedulability performance of the DAG-Str algorithm, we proved that preemptive GEDF scheduling algorithm has a resource augmentation bound of $\frac{3+\sqrt{5}}{2}$ for any stretched task set of $n < \varphi.\overline{m}$, where φ is the golden ratio, n is the number of DAGs in the set and $\overline{m}$ is the number of the available processors in the system after stretching. Recently, the value has been proved to be the resource augmentation bound of GEDF when it is used to schedule DAG tasks directly[82].

Additionally, we proposed a modified version of the DAG-Str algorithm which is called the Segment Stretching (Seg-Str) algorithm. It aims at reducing the number of job migration and preemption resulting from the stretching process when compared to the DAG-Str algorithm. We analyzed the performance of the Seg-Str algorithm and we proved that it has the same resource augmentation bound as for the DAG-Str algorithm.

Direct Scheduling Approach at DAG-Level

In Chapter 4, we analyzed the importance of the internal structure of DAG tasks when common scheduling algorithms are used to schedule them on multiprocessor systems. Moreover, we considered sporadic constrained-deadline DAG tasks that are scheduled on multiprocessor systems with any work conserving algorithm in general, and mainly with GEDF. Our scheduling analyses are done based on the internal dependencies that determine the execution of subtasks. In order

to achieve this, we added extra timing parameters to subtasks which are derived from the global parameters of their DAGs, such as local offset, local relative deadline and release jitter. These timing parameters are different from intermediate parameters assigned by the Model Transformation approach, because local parameters tend to define the maximum execution interval of each subtask based on its precedence constraints without imposing any external intermediate parameters.

We studied the scheduling of DAG tasks at DAG-Level, in which real-time algorithms schedule DAGs based on their global timing parameters. This approach is common in real-time researches regarding DAG scheduling but they mainly concentrate on timing parameters of DAGs such as their total WCET, critical path length and slack time. However, we provided scheduling analyses which take into consideration the internal structure of DAGs and the precedence constraints between subtasks. We provided an adapted schedulability condition for DAG scheduling for GEDF algorithm. The scheduling analyses are based on identifying upper bounds of interference and workload of DAG tasks.

Direct Scheduling Approach at Subtask-Level

Moreover in Chapter 4, we proposed a Subtask-Level scheduling of DAGs in which real-time algorithms schedule DAGs based on assigned local parameters of subtasks rather than on global parameters of DAGs. We argued the advantage of Subtask-Level scheduling on feasibility analysis of DAG tasks by adapting the necessary feasibility condition based on the load function. We provided a modification to the condition, at Subtask-Level, that is more accurate. Then, we provided interference and workload analyses for this scheduling, and schedulability conditions for any work conserving algorithm and for GEDF.

Simulation tool: *YARTISS*

We started Chapter 5 by proving the incomparability of DAG scheduling approaches which were presented in the previous chapters. This was done by presenting scheduling examples to show that a given DAG set is successfully scheduled by one approach and not by the other, and vice versa. Due to this incomparability, we compared the performance of the DAG scheduling approaches using extensive simulations. We simulated the global preemptive scheduling of random DAG sets on homogeneous multiprocessor systems when EDF and DM scheduling algorithms are used.

In order to perform the simulation experiments, we presented *YARTISS*, which is a free open-source simulation tool written in Java for real-time multiprocessor scheduling. During its design, we focused on providing a generic simulation tool and an easy-to-use modular design which can be easily used and extended by fellow researchers. *YARTISS* can be used to simulate the execution of large scale data sets of different task models on multiprocessor platforms and to show the scheduling results visually within the simulator. We included the DAG model within the simulator and we designed a DAG generator that is able to produce many variations to the structure of DAGs, including the size of tasks (by changing the number of their subtasks) and the probability of internal dependencies between subtasks.

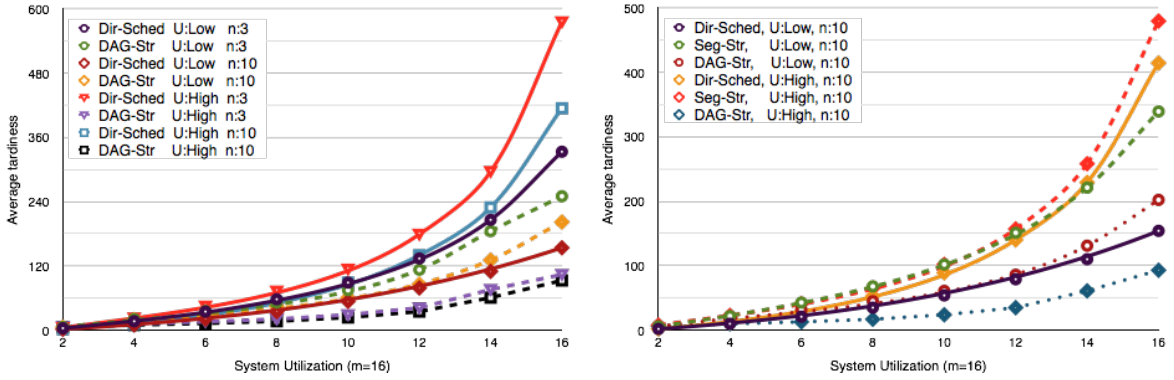
6.2 Future Work and Perspectives

Scheduling parallel real-time tasks on multiprocessor systems is an interesting problem, and we believe that there are many open problems and aspects that need to be studied and analyzed. Here, we provide our main perspectives of future work:

Real-time DAG scheduling on soft real-time systems

Usually, applications in hard real-time systems are critical in a way that deadline misses are not acceptable. Until now, their domains are limited and dependent on certain reliable scheduling algorithms and platforms. The switching into more advanced platforms could happen but slowly due to potential execution risks. However, soft real-time application, such as multimedia and communication services, are diverse. The main objective of soft real-time scheduler is to reduce the response time of such applications and it can tolerate certain deadline miss rate. The use of parallelism in soft real-time applications can be useful so as to improve their tardiness bounds when compared to sequential design. For example, the execution of a real-time video processing application can be executed in parallel on multiprocessor systems so as to improve its response time while taking advantage of the available processors in the system.

Hence, we believe that studying the scheduling problem of parallel real-time tasks on multiprocessor systems is an interesting problem. Similarly to sequential model, soft real-time scheduling of DAG tasks on multiprocessor systems can be analyzed and tardiness bounds for scheduling algorithms can be provided. Towards this perspective, we already started investigating this scheduling problem in [97]. We provided experimental evaluations of tardiness bounds of DAG Stretching algorithm and Direct Scheduling approaches for GEDF. The simulation results are



(a) Average tardiness for GEDF scheduling algorithm while varying the system utilization (U) and the number of tasks (n).

(b) Average tardiness for GEDF scheduling algorithm while the varying system utilization (U).

FIGURE 6.1: Simulation results showing tardiness bounds of DAG scheduling approaches.

shown in Figure 6.1 while varying the utilization of DAGs (*High* or *Low*) and the size of DAG sets (3 or 10 DAGs per set).

A more general DAG model with multi-threaded subtasks

In this thesis, we considered a parallel model of DAG tasks in which subtasks are dependent sequential threads with precedence constraints. In a previous work [101, 102], we explored the possibility of considering a DAG model that is more general, in which subtasks consist of multiple-threads. We provided two algorithms to determine the execution order of threads and whether they should execute in parallel or sequentially based on the structure of DAGs. We considered a moldable parallel scheduling in which the real-time scheduler determines the level of parallelism of subtasks before the beginning of the scheduling. We believe that generalizing the DAG model is an interesting open problem, in which more schedulability analysis can be provided. Furthermore, rigid and gang scheduling approaches for this parallel model can be considered.

Résumé de la thèse en français

Ordonnancement Temps Réel de Graphes de Tâches Parallèles sur Systèmes Multiprocesseurs

Dans cette thèse, nous étudions l'ordonnancement temps réel de graphes de tâches parallèles sur plateformes multiprocesseurs. Le manuscrit est organisé de la manière suivante :

Introduction Générale

Systèmes Temps Réel

En informatique, un système temps réel est un système qui exige une exécution correcte de ses tâches dans des délais imposés. Les différents domaines d'applications de ce type de système sont dans les transports, le multimédia et les systèmes de communication.

Les systèmes temps réel sont divisés en deux catégories en fonction de la criticité de leurs tâches : temps réel strict (dur) et temps réel souple. Dans un système temps réel dur, toutes les échéances doivent être respectées pour éviter des conséquences catastrophiques. Pour un système temps réel souple, le dépassement de certaines contraintes temporelles réduit la qualité du service fourni par l'application.

Nous supposons qu'un système temps réel τ est constitué de n tâches où $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Chaque tâche correspond à l'exécution d'un ensemble d'instructions (travail) qui se répète. Dans le modèle de tâches temps réel classique, une tâche τ_i est caractérisée par :

- Durée d'exécution C_i : c'est la durée d'exécution dans le pire cas (la plus longue durée) pendant laquelle un travail de τ_i peut s'exécuter. Il est important de considérer cette durée du pire cas pour garantir l'ordonnancement.
- Période T_i : c'est la durée entre les différents travaux de la même tâche. Pour une tâche périodique, la période est constante. Pour une tâche sporadique, la période représente la durée minimale entre deux instances.
- Échéance relative D_i : c'est la durée pendant laquelle la tâche doit s'exécuter. Si $D_i = T_i$, une tâche est dite à échéance implicite, si $D_i \leq T_i$, une tâche est dite à échéance contrainte, elle est dite à échéance arbitraire si D_i n'est pas contrainte par T_i.
- Instant d'activation (offset) O_i : c'est le décalage de la date d'activation du premier travail de la tâche par rapport à une origine des temps ($t = 0$).

Nous notons J_i^j le $j^{\text{ième}}$ travail de la tâche τ_i et il est caractérisé par deux paramètres : un instant d'activation r_i^j et une échéance absolue d_i^j. La figure 6.2 représente les caractéristiques temporelles de la tâche τ_i. L'utilisation U_i est le quotient C_i/T_i et l'utilisation $U(\tau)$ du système est définie par la somme $\sum_{i=1}^{n} U_i$.

Plateformes Multiprocesseurs

Dans cette thèse, nous considérons des systèmes temps réel multiprocesseurs dont les plateformes sont constituées de plus d'une unité d'exécution. L'ordonnancement multiprocesseurs n'est pas une simple généralisation du cas monoprocesseur, la problématique d'ordonnancement devient plus complexe. La responsabilité de l'ordonnanceur multiprocesseurs est de choisir, à chaque instant, les tâches devant s'exécuter sur les processeurs en respectant leurs contraintes temporelles. Pour un système multiprocesseurs, un ensemble de tâches ne peut pas être ordonnancé si son utilisation est supérieure au nombre de processeurs.

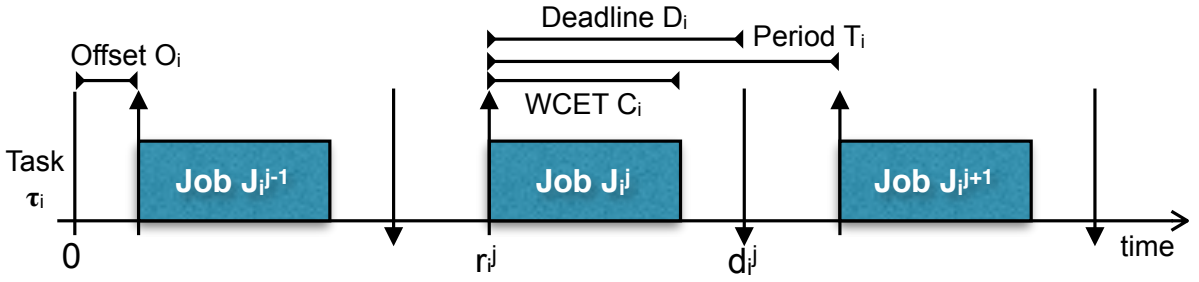


FIGURE 6.2: Modèle de tâches indépendantes séquentielles.

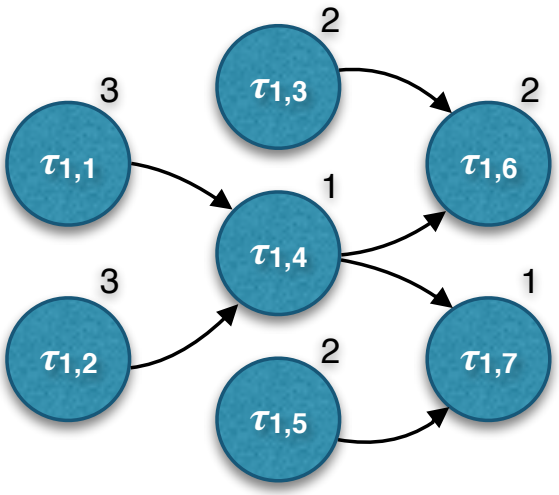


FIGURE 6.3: Un exemple d'une tâche du modèle DAG.

Applications Parallèles le modèle de graphe

Récemment, l'utilisation d'architectures multiprocesseurs a fortement augmenté dans les systèmes industriels. De ce fait, le nombre d'applications parallèles va également fortement progresser. Nous intéressons aux contraintes temps réel pour ces applications parallèles.

Dans cette thèse, nous considérons un modèle général de tâche parallèle représenté par un graphe orienté acyclique (ou les DAG). Dans ce modèle, un graphe τ_i est composé d'un ensemble de sous-tâches dépendantes. Les dépendances sont exprimées par des relations de précédence. Le graphe τ_i est caractérisé par $(\{\tau_{i,j}|1 \leq j \leq n_i\}, G_i, O_i, D_i, T_i)$, où le premier paramètre représente l'ensemble des n_i sous-tâches de τ_i. Chaque sous-tâche $\tau_{i,j}$, où $1 \leq j \leq n_i$, est caractérisée par sa durée d'exécution dans le pire cas $C_{i,j}$. Le paramètre G_i est l'ensemble des relations entre les sous-tâches, O_i est l'instant d'activation du graphe, D_i est son échéance relative et T_i est sa période. On notera que les sous-tâche partagent la même période et la même échéance.

Dans un graphe τ_i, une relation de précédence entre deux sous-tâches $\tau_{i,j}$ et $\tau_{i,k}$ signifie que le dernier commence son exécution quand $\tau_{i,j}$ se termine. Le délai critique du graphe (critical path length) L_i représente la longueur du plus long chemin dans le graphe (chemin critique). Une sous-tâche dans le graphe sera activée à la date de terminaison de ses prédécesseurs. Donc, l'ordre d'exécution des sous-tâches est dynamique, les sous-tâches pouvant s'exécuter en parallèle ou en séquence par rapport aux décisions de l'ordonnanceur. Dans la figure 6.3, nous montrons l'exemple d'un graphe τ_i composé de sept sous-tâches. Les relations de précédence sont représentées par des flèches. Le chemin critique du graphe se compose de $\tau_{1,1}$ (ou $\tau_{1,2}$), $\tau_{1,4}$ et $\tau_{1,6}$ (ou $\tau_{1,7}$).

Nous proposons deux méthodes pour ordonnancer ce graphe en tenant compte des contraintes de précédence : une transformation de modèle et l'ordonnancement direct.

Ordonnancement des Graphes par Transformation de modèle

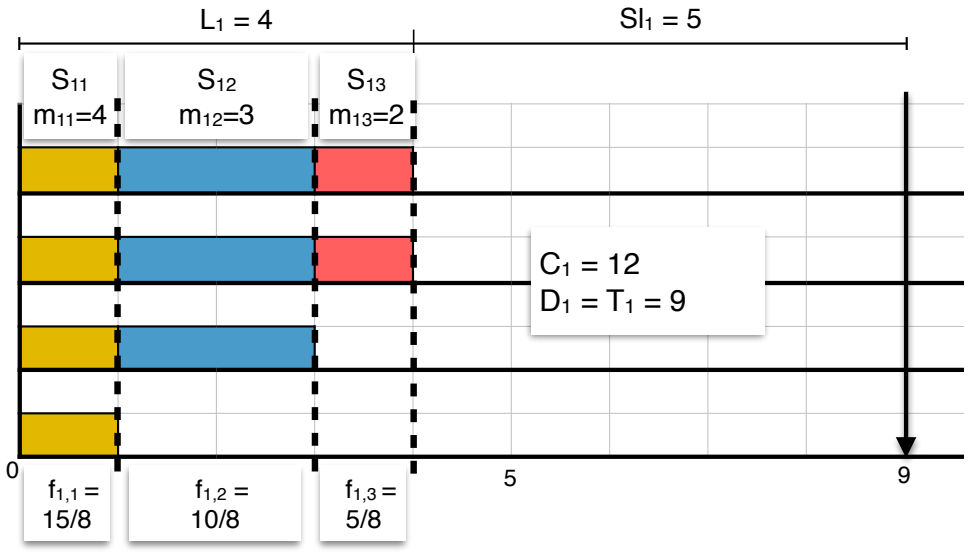
La méthode de transformation de modèle est utilisée dans l'état de l'art pour l'ordonnancement des tâches parallèles sur les systèmes multiprocesseurs, comme dans [77, 108]. Cette méthode simplifie l'ordonnancement des graphes en transformant le modèle parallèle en un modèle séquentiel de sous-tâches indépendantes afin de supprimer les dépendances internes. Les tâches ainsi transformées pourront être traitées par un algorithme d'ordonnancement standard.

Dans cette thèse, nous nous intéressons à l'ordonnancement des graphes périodiques à échéance implicite et nous proposons deux algorithmes de transformations (DAG-Str et Seg-Str) pour forcer l'exécution séquentielle des sous-tâches.

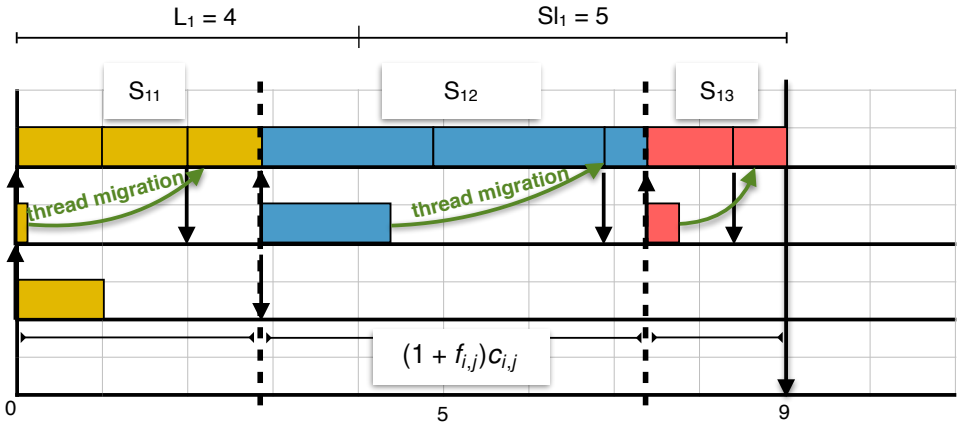
Algorithme DAG-Str

L'algorithme DAG-Str (**DAG-Str**etching) vise à forcer une exécution séquentielle des graphes. Il consiste à étirer le chemin critique de chaque graphe τ_i jusqu'à son échéance et à créer un thread (fil d'exécution) séquentiel d'utilisation égale à 1. Une tâche d'utilisation inférieure ou égale à 1 sera transformée en un seul thread séquentiel. Par contre, si la tâche a une utilisation supérieure à 1, elle est transformée en une séquence de segments S_i. Chaque segment $S_{i,j}$ est composé de threads non-critiques indépendants a qui sont affectés des offsets et des échéances relatives locales. Nous présentons dans la figure 6.4(b) un exemple montrant l'algorithme DAG-Str appliqué au graphe τ_1 de la figure 6.4(a). En utilisant l'algorithme DAG-Str, le graphe τ_1 est transformé en un thread principal de pire coût d'exécution et échéance relative égaux à 9 plus 3 segments s'exécutant en séquence. La durée d'exécution de chaque segment est utilisée pour affecter des offsets et échéances relatives à ces threads indépendants. Par exemple, un thread du segment $S_{1,1}$ a un coût d'exécution égal à 1 et une échéance égale à 3. Par contre, l'autre thread a une échéance plus courte (égale à 2) pour éviter une exécution parallèle avec sa deuxième partie qui s'exécute dans le thread principal.

Dans cette thèse, nous utilisons l'algorithme d'ordonnancement Global EDF pour ordonnancer les threads indépendants générés après transformation par DAG-Str. Nous proposons une analyse d'ordonnançabilité pour calculer le facteur d'expansion (speedup) dans le cas d'ordonnancement



(a) Un exemple d'un DAG τ_1.



(b) L'algorithme DAG-Str.

FIGURE 6.4: La méthode de transformation du modèle.

EDF. Nous prouvons que l'algorithme EDF avec DAG-Str a un facteur d'expansion égal à $\frac{3+\sqrt{5}}{2}$ pour tous les ensemble de tâches transformées $n < \varphi \times \overline{m}$, où n est le nombre de tâches dans l'ensemble, $\overline{m}$ est le nombre des processeurs disponibles après la transformation et φ le nombre d'or (valant exactement $\frac{1+\sqrt{5}}{2}$). Récemment dans [82], le même facteur a été prouvé dans le cas des tâches DAG sans transformation avec l'algorithme d'ordonnancement Global EDF. Donc, l'algorithme DAG-Str simplifie l'ordonnancement des DAGs sans perde de performance et il ne nécessite pas d'un ordonnanceur Global EDF adapté a la prise en compte des dépendances.

Algorithme Seg-Str

Nous proposons l'algorithme SEG-Str (**Seg**ment-**Str**etching) qui est une version modifiée de l'algorithme de transformation de modèle (DAG-Str) visant à réduire le nombre de migrations et préemptions des threads résultants de la transformation. La différence entre l'algorithme Seg-Str et l'algorithme DAG-Str est que ce dernier fait migrer un thread de chaque segment vers le thread principal, tandis que l'algorithme Seg-Str force un seul thread de tous les segments à migrer.

Dans la figure 6.5, nous montrons le résultat de l'algorithme Seg-Str quand il est appliqué au graphe τ_1 de la figure 6.4(a). Nous avons montré dans la figure 6.4(b) que la transformation en utilisant l'algorithme DAG-Str du graphe τ_1 a généré 3 migrations de thread. Dans le cas de l'algorithme Seg-Str, le thread principale est rempli par des threads entiers de segments, comme le thread du segment $S_{1,1}$. Par conséquence, un thread au plus migrera dans le thread principal (le thread du segment $S_{1,2}$).

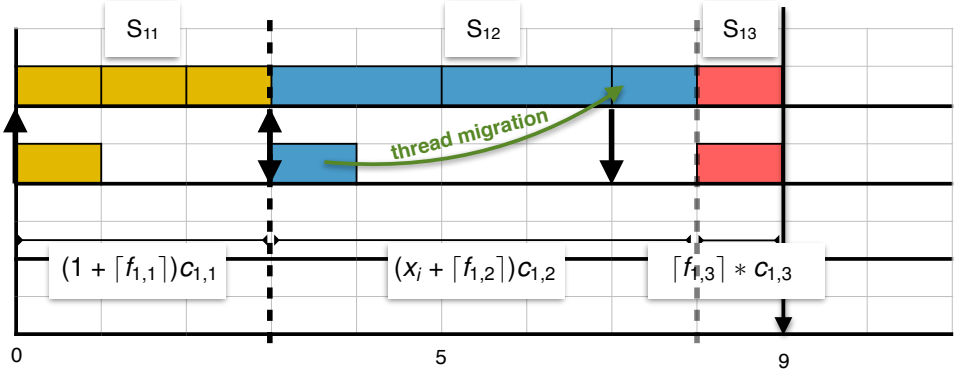


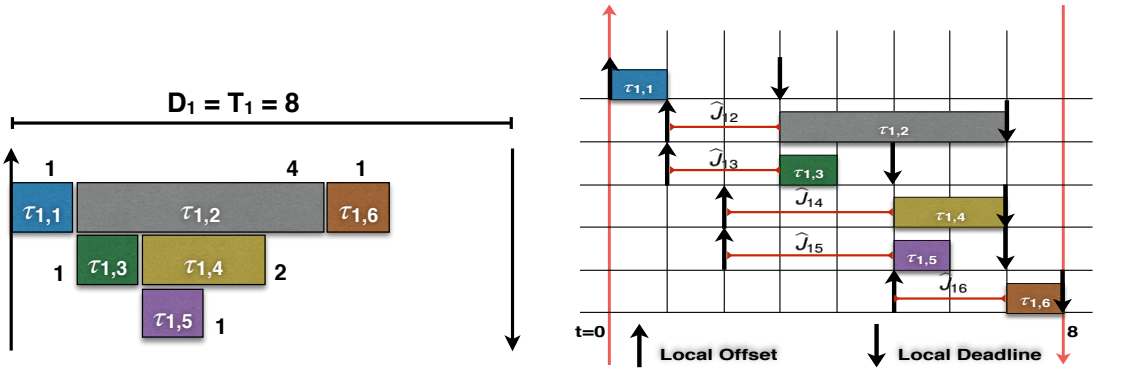
FIGURE 6.5: Seg-Str algorithm.

Nous montrons que l'algorithme Seg-Str a le même facteur d'expansion (speedup) que l'algorithme DAG-Str. Il offre les mêmes performances avec moins de migrations.

Ordonnancement Direct des Graphes

Nous nous intéressons à l'ordonnancement des graphes sporadiques à échéance contrainte. La méthode d'ordonnancement direct conserve les caractéristiques générales des graphes et considère que l'ordonnanceur prendra en compte les contraintes de précédence pour décider qu'un tâche est prête à être exécuter. Par conséquent, nous pouvons utiliser des algorithmes temps

réel standards pour l'ordonnancement des graphes. Les algorithmes EDF et Deadline Monotonic (DM) initialement proposés pour des tâches séquentielles indépendantes sont largement utilisés dans de nombreuses recherches d'ordonnancement de graphes de tâches parallèles tels que [27, 36, 44, 81, 82]. Toutefois, ces recherches ne considèrent que la structure externe des graphes pour l'ordonnancement et l'analyse d'ordonnançabilité. Les paramètres considérés sont le coût d'exécution total des sous-tâches, l'échéance, la période et la longueur du chemin critique pour le graphe. Dans cette thèse, nous montrons l'importance de la structure interne des graphes sur l'ordonnancement et l'analyse comme les relations entre les sous-tâches et ces coût d'exécutions. Nous proposons donc des algorithmes simples pour définir les paramètres temporels locaux des sous-tâches, comme les échéances, les périodes et les gigues d'activation. L'affectation des priorités pourra être faite au niveau d'une tâche (graphe), dans ce cas toutes les sous-tâches du graphe ont la même priorité. L'affectation des priorités pourra être faite au niveau des sous-tâches et dans ce cas les sous-tâches ont leur propre priorité. L'analyse sera toujours effectuée au niveau des sous-tâches en considérant les paramètres locaux. L'analyse est basé sur le calcul de la charge maximale des tâches ou des sous-tâches (workload). La connaissance de la structure interne d'un graphe permettra, lors de l'analyse, une estimation plus précise du workload.



(a) Un graphe τ_1 qui se compose de 6 sous-tâches.

(b) Les paramètres temporels locales de chaque sous-tâche dans τ_1.

FIGURE 6.6: Les paramètres temporels locales des sous-tâches.

Dans la figure 6.6, nous montrons l'affectation des paramètres locaux aux sous-tâches de graphe τ_1 qui contient 6 sous-tâches. La structure de la tâche et le coût d'exécution de ses sous-tâches sont présentés dans la figure 6.6(a). Dans la figure 6.6(b), chaque sous-tâche $\tau_{i,j}$ est caractérisée par trois paramètres : un offset local $O_{i,j}$ qui détermine le premier instant d'activation de la sous-tâche par rapport à l'activation de son graphe (offset au plus tôt), une échéance locale $D_{i,j}$ qui détermine le dernier instant d'exécution autorisé pour la sous-tâche (échéance au plus tard)

pour permettre l'exécution du plus long successeur et une gigue d'activation $\overline{j}_{i,j}$ qui détermine la longueur de l'intervalle d'activation de la sous-tâche. Par exemple, la sous-tâche $\tau_{1,1}$ a un offset local égal à 0, une échéance locale égale à 3 et une gigue d'activation égale à 0. Ces paramètres locaux sont différents des paramètres intermédiaires attribués par les algorithmes de la méthode de transformation de modèle. Les paramètres locaux définissent l'intervalle maximal d'exécution de chaque sous-tâche.

Nous proposons une analyse d'ordonnançabilité au niveau des graphes sur multiprocesseurs quand l'algorithme EDF est utilisé. L'ordonnanceur EDF affecte des priorités aux graphes par rapport aux échéances des graphes, mais l'analyse d'ordonnançabilité se base sur les paramètres locaux. Nous proposons un test d'ordonnancabilité pour EDF au niveau des graphes en calculant l'interférence entre les sous-tâches.

Ensuite, nous proposons un ordonnancement EDF des graphes au niveau des sous-tâches, dans lequel l'algorithme d'ordonnancement prend des décisions par rapport aux paramètres locaux (paramètres temporel des sous-tâches) plutôt que par rapport aux paramètres des graphes. Dans le cas d'EDF, l'algorithme attribue des priorités différentes pour les sous-tâches par rapport aux échéances locales. L'ordonnancement direct au niveau des sous-tâches est une nouvelle approche qui n'avait pas été utilisée dans le domaine de l'ordonnancement temps réel des graphes. Nous montrons que la condition nécessaire de faisabilité par rapport à la charge du processeur est plus précise lorsque l'on considère l'ordonnancement des graphes au niveau des sous-tâches. Nous fournissons l'analyse d'interférences pour l'ordonnancement EDF au niveau des sous-tâches comme dans le cas d'ordonnancement direct au niveau des graphes.

Résultats Expérimentaux

Dans cette partie nous montrons que les deux méthodes d'ordonnancement de graphes ne sont pas comparables du point de vue de l'ordonnançabilité. C'est à dire qu'aucune des deux méthodes ne domine l'autre. Toutefois, nous fournissons des résultats de simulation pour comparer leur comportement en moyenne en utilisant les algorithmes d'ordonnancement globaux EDF et DM. Nous avons développé un logiciel nommé *YARTISS* pour générer des graphes aléatoires et réaliser des simulations d'ordonnancement en multiprocesseurs.

Incomparabilité des méthodes d'ordonnancement de graphes

Nous comparons l'algorithme DAG-Str (transformation de modèle) et l'ordonnancement direct au niveau des graphes et au niveau des sous-tâches. Nous fournissons quatre exemples d'ordonnancement afin de prouver que ces méthodes ne sont pas comparables. C'est à dire qu'il existe des ensembles de graphes qui sont ordonnançable en utilisant une méthode et non ordonnançable par l'autre, et vice versa.

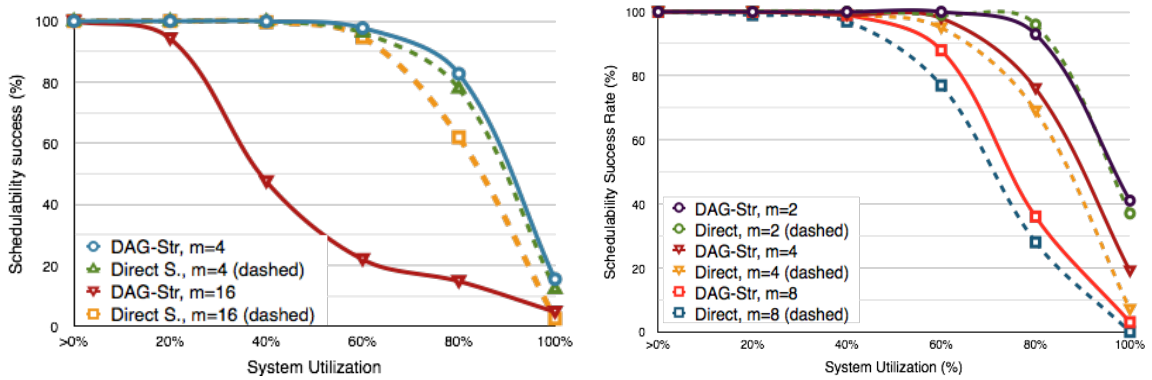
Les deux premiers exemples d'ordonnancement permet de comparer l'algorithme DAG-Str avec l'ordonnancement direct au niveau des graphes. Les deux autres exemples permet de comparer l'ordonnancement direct au niveau des graphes et au niveau des sous-tâches. Dans tous les exemples, nous considérons l'ordonnancement préemptif global des graphes périodiques à échéances implicites et activation synchrone sur une plate-forme multiprocesseurs.

Simulateur temps réel : YARTISS

En raison de l'incomparabilité des méthodes d'ordonnancement des graphes, nous utilisons des simulations pour comparer leurs performances en moyenne.Nous avons développé un outil de simulation temps réel (YARTISS) contenant un générateur aléatoire de graphes. Nous générons les paramètres temporels et les structures internes des graphes (dépendances). Pour comparer les différentes méthodes étudiées, nous avons simulé l'ordonnancement des graphes en faisant varier trois paramètres : le nombre de processeurs dans le système, la taille des graphes en fonction du nombre de sous-tâches et le parallélisme interne des graphes grâce aux relations de précédence.

Résultats de simulation

Les simulations effectués montre que l'ordonnancement direct des graphes donne des meilleurs résultats avec l'algorithme d'ordonnancement EDF (figure 6.7(a)). Pour sa part, la méthode de transformation de modèle (algorithme DAG-Str) donne des meilleurs résultats avec l'algorithme DM (figure 6.7(b)). Nous montrons les résultats des simulations dans la figure 6.7. L'augmentation du nombre de processeurs (passage à l'échelle) est mieux supportée par l'algorithme DAG-Str en ordonnancement DM. Par contre, l'ordonnancement direct supporte le passage à l'échelle



(a) Les résultats de simulations de l'algorithme EDF. (b) Les résultats de simulations de l'algorithme DM.

FIGURE 6.7: Les résultats de simulations.

indifféremment de l'ordonnancement.

Appendix A

DAG TASK Generator in YARTISS Simulator

```
1 /**
2  * Real-time task of Directed Acyclic Graph (DAG) model. A DAG task
3  * consists of a set of real-time subtasks with precedence constraints
4  * which determine the execution flow of its subtasks.
5  * @author Manar Qamhieh
6  */
7 public class GraphTask extends PeriodicTask implements ITaskElement {
8     private ArrayList<SubTask> subtasks;
9     public GraphTask(long firstrelease, long wcee, long period, long
      deadline,
10             int priority, ArrayList<SubTask> subtasks,
11             IEnergyConsumptionProfile p) {
12         super(firstrelease, getTotolWcet(subtasks), wcee, period, deadline
      , priority, p);
13         this.subtasks = subtasks;
14     }
15     public GraphTask(GraphTask graphTask) {
16         this(graphTask.getOffset(), graphTask.getWcee(), graphTask.
      getPeriod(), graphTask.getDeadline(), graphTask.getPriority(),
      graphTask.getSubtasks(), graphTask.getEnergyConsumptionProfile());
17     }
18
19     /**
```

```
20      * Gets the total worst-case execution time (WCET) of a graph,
21      * which is the sum of execution time of its subtasks.
22      * @param subtasks the set of subtasks of the graph
23      * @return total WCET of a graph
24      */
25     private static long getTotolWcet(ArrayList<SubTask> subtasks) {
26         long wcets = 0;
27         for (SubTask subTask : subtasks) {
28             wcets += subTask.getWcet();
29         }
30         return wcets;
31     }
32
33     /**
34      * Gets the subtasks of the graph
35      * @return list of subtasks
36      */
37     public ArrayList<SubTask> getSubtasks() {
38         return subtasks;
39     }
40
41     /**
42      * Gets the source subtask/s which are the subtask with predecessors.
43      *  Each graph has at least one source subtask.
44      * @return list of source subtasks
45      */
46     public ArrayList<SubTask> getSourceSubtasks() {
47         ArrayList<SubTask> source = new ArrayList<SubTask>();
48         for (SubTask subtask : subtasks) {
49             if (subtask.getParentTasks().isEmpty())
50                 source.add(subtask);
51         }
52         return source;
53     }
54
55     /**
56      * Gets the sink subtask/s of a graph which are the subtasks with no
57      * successors. Each graph has at least one sink subtask.
```

```
58      * @return a list of sink subtasks
59      */
60     public ArrayList<SubTask> getSinkSubtasks() {
61         ArrayList<SubTask> sink = new ArrayList<SubTask>();
62         for (SubTask subtask : subtasks) {
63             if (subtask.getChildrenTasks().isEmpty())
64                 sink.add(subtask);
65         }
66         return sink;
67     }
68
69     /**
70      * Removes a subtask from the graph and its corresponding relations
71      * with the remaining subtasks.
72      * @param subtask the subtask to be removed
73      */
74     public void removeSubtask(SubTask subtask) {
75         for (SubTask child : subtask.getChildrenTasks()) {
76             removeEdge(subtask, child);
77         }
78         for (SubTask parent : subtask.getParentTasks()) {
79             removeEdge(parent, subtask);
80         }
81         subtasks.remove(subtask);
82     }
83
84     /**
85      * Adds a subtask to the graph.
86      * @param subtask subtask to be added
87      */
88     public void addSubtask(SubTask subtask) {
89         subtask.setIndex(subtasks);
90         subtasks.add(subtask);
91     }
92
93     /**
94      * Adds a directed edge between two subtasks in the graph
95      * @param parent predecessor subtask
```

```
96      * @param child successor subtask
97      */
98     public void addEdge(SubTask parent, SubTask child) {
99         if (!parent.getChildrenTasks().contains(child))
100            parent.getChildrenTasks().add(child);
101        if (!child.getParentTasks().contains(parent))
102            child.getParentTasks().add(parent);
103    }
104
105    /**
106     * Removes a directed edge between two subtasks in the graph
107     * @param parent predecessor subtask
108     * @param child successor subtask
109     */
110    public void removeEdge(SubTask parent, SubTask child) {
111        if (parent.getChildrenTasks().contains(child))
112            parent.getChildrenTasks().remove(child);
113        if (child.getParentTasks().contains(parent))
114            child.getParentTasks().remove(parent);
115    }
116
117    /**
118     * Verify if an edge can be created between 2 subtasks in a graph. An
119     * edge can't be created if 2 subtasks share the same parent.
120     * @param parentSub predecessor subtask
121     * @param childSub successor subtask
122     * @return true if the edge can be created, false otherwise
123     */
124    public boolean canAddEdge(SubTask parentSub, SubTask childSub) {
125        boolean addEdge = true;
126        for (SubTask parent : parentSub.getParentTasks()) {
127            if (childSub.getParentTasks().contains(parent))
128                addEdge = false;
129        }
130        long cpl = childSub.getWcet() + getPathLength(parentSub);
131        if (cpl > getDeadline()) {
132            addEdge = false;
133        }
```

```
134         return addEdge;
135     }
136 
137     /**
138      * Recursive function calculates the length of the longest path from
139      * the source of the graph to the SubTask 'subtask'
140      * @param subtask a given subtask in the graph
141      * @return the length of the path
142      */
143     private long getPathLength(SubTask subtask) {
144         long cpl = 0;
145         for (SubTask sub : subtask.getParentTasks()) {
146             long tmp = getPathLength(sub);
147             if (tmp > cpl)
148                 cpl = tmp;
149         }
150         return cpl + subtask.getWcet();
151     }
152 
153     /**
154      * Calculates the length of the critical path of the graph, which is
155      * the longest path from a source to a sink subtask. Accordingly,
156      * the earliest and latest release time and deadline are calculated
157    * for each subtask.
158      */
159     public void calculateCriticalPath() {
160         initializeGraph();
161         for (SubTask sourceSubtask : getSourceSubtasks()) {
162             sourceSubtask.setEarliestFinishTime(sourceSubtask.getWcet());
163             calculateEarliestTime(sourceSubtask);
164         }
165         for (SubTask sinSubtask : getSinkSubtasks()) {
166             sinSubtask.setLatestFinisheTime(getDeadline());
167             calculateLatestTime(sinSubtask);
168         }
169         for (SubTask sub : subtasks) {
170             sub.setFirstRelease((sub.getEarliestFinishTime() - sub.getWcet
    ()));
```

```
171             sub.setLatestFinisheTime(sub.getLatestFinishTime()
172                     - sub.getOffset());
173         }
174     }
175
176     /**
177      * A function for DAGs with parallel subtasks, i.e., a subtask is
178      * a set of threads that can execute in parallel or sequentially.
179      * This function parallelizes a graph to the maximum
180      * based on critical path length (necessary feasibility condition).
181      */
182     public void parallelizeGraph() {
183         boolean repeat;
184         do {
185             calculateCriticalPath();
186             repeat = parallelizeSubtasks();
187         } while (repeat);
188     }
189
190     /**
191      * Initializes the first release and deadline of the subtasks of the
192      * graph.
193      */
194     private void initializeGraph() {
195         for (SubTask subTask : subtasks) {
196             subTask.setEarliestFinishTime(0);
197             subTask.setLatestFinisheTime(0);
198         }
199     }
200
201     /**
202      * Calculates the earliest release time of a subtask based on the
203      * depth-first algorithm. This is a recursive function used to
204      * traverse the subtasks of the graph from source to sink, in order
205      * to find the critical path of a graph.
206      * @param subtask a given subtask in the graph
207      */
208     private void calculateEarliestTime(SubTask subtask) {
```

```
209         long earliestTime;
210         if (subtask.isSinkSubtask())
211             return;
212         for (SubTask child : subtask.getChildrenTasks()) {
213             earliestTime = subtask.getEarliestFinishTime() + child.getWcet
        ();
214             if (child.getEarliestFinishTime() < earliestTime)
215                 child.setEarliestFinishTime(earliestTime);
216             calculateEarliestTime(child);
217         }
218     }
219
220     /**
221      * Calculates the latest deadline of a subtask based on the depth-
222      * first algorithm. This is a recursive function traverses the graph
223      * from sink to source, in order to find the critical path of a graph.
224      * @param subtask a given subtask in the graph
225      */
226     private void calculateLatestTime(SubTask subTask) {
227         long latestTime;
228         if (subTask.isSourceSubtask())
229             return;
230         for (SubTask parent : subTask.getParentTasks()) {
231             latestTime = subTask.getLatestFinishTime() - subTask.getWcet()
        ;
232             if (parent.getLatestFinishTime() == 0
233                     || parent.getLatestFinishTime() > latestTime)
234                 parent.setLatestFinisheTime(latestTime);
235             calculateLatestTime(parent);
236         }
237     }
238
239     /**
240      * Prints the local parameters of subtasks.
241      */
242     public void printCriticalInfo() {
243         for (SubTask subTask : subtasks) {
244             System.out.println("[" + subTask.getId() + "]: wcet= "
```

```
245                     + subTask.getWcet() + "   "
246                     + subTask.getEarliestFinishTime() + "/"
247                     + subTask.getLatestFinishTime() + " --- nbProc: "
248                     + subTask.getNumOfProc());
249         }
250     }
251
252     /**
253      * A function for DAGs with parallel subtasks, i.e., a subtask is
254      *  defined as a set of threads that can execute in parallel or
255      * sequentially. This function parallelizes subtasks to the maximum.
256      * @return True if a graph is parallelizable, False otherwise.
257      */
258     private boolean parallelizeSubtasks() {
259         boolean parallelized = false;
260         CopyOnWriteArrayList<SubTask> modList = new CopyOnWriteArrayList<
    SubTask>();
261         modList.addAll(subtasks);
262         for (SubTask subTask : modList) {
263             if (subTask.isCriticalTask() && subTask.getNumOfProc() > 1) {
264                 parallelized = true;
265                 for (int i = 0; i < subTask.getNumOfProc(); i++) {
266                     SubTask tmpSub = (SubTask) SchedulableFactory.
    newInstance("subtask", subTask.getOffset(), subTask.getWcet()/subTask.
    getNumOfProc(), subTask.getWcee(), subTask.getPeriod(), subTask.
    getDeadline(), subTask.getPriority(), 1);
267                     tmpSub.setIndex(modList);
268                     modList.add(tmpSub);
269                     for (SubTask parent : subTask.getParentTasks()) {
270                         parent.getChildrenTasks().remove(subTask);
271                         addEdge(parent, tmpSub);
272                     }
273                     for (SubTask child : subTask.getChildrenTasks()) {
274                         child.getParentTasks().remove(subTask);
275                         addEdge(tmpSub, child);
276                     }
277                 }
278                 modList.remove(subTask);
```

```
279             }
280         }
281         subtasks.clear();
282         subtasks.addAll(modList);
283         return parallelized;
284     }
285 
286     /**
287      * Prints the timing parameters of a graph.
288      * @return the local parameters of the graph as a string.
289      */
290     public String toString() {
291         return "[" + getId() + "] wcet:" + getTotolWcet(subtasks)
292                 + "\t nbSubTasks:\t" + subtasks.size() + "\tdeadline: \t"
293                 + getDeadline() + " \tperiod:\t " + getPeriod();
294     }
295 
296     /**
297      * Prints the timing parameters of the graph and its subtasks.
298      */
299     public void printGraph() {
300         String str = this.toString() + "\n";
301         for (SubTask sub : subtasks) {
302             str += "\t Subtask " + sub + "\n";
303         }
304         System.out.println(str);
305     }
306 
307     @Override
308     public ITask cloneTask() {
309         return new GraphTask(this);
310     }
311 
312     /**
313      * An 'accept' function for the visitable element
314      */
315     @Override
316     public void accept(ITaskElementVisitor visitor) {
```

```
317         visitor.visitGraphTask(this);
318     }
319
320     @Override
321     public String getType() {
322         return "graph";
323     }
324
325     @Override
326     public Job activate(long time) {
327         Job j = null;
328         for (SubTask subtask : getSubtasks()) {
329             j = subtask.activate(time);
330         }
331         return j;
332     }
333
334     @Override
335     public long getMaxTardiness() {
336         long max = 0;
337         for (SubTask subtask : getSubtasks()) {
338             if (subtask.getMaxTardiness() > max)
339                 max = subtask.getMaxTardiness();
340         }
341         return max;
342     }
343
344     /**
345      * Transforms the graph task into a multi-Threaded Segment (MTS) task,
346      * which consists of a sequence of parallel segments each consists
347      * of identical threads.
348      * @return transformed task of a MTS model.
349      */
350
351     public ArrayList<ParallelSegmentInfo> transformToMTS() {
352         calculateCriticalPath();
353         ArrayList<ParallelSegmentInfo> psList = new ArrayList<
        ParallelSegmentInfo>();
```

```
354         int count;
355         long index = 0, segOffset;
356         boolean isOver = false;
357 
358         while (!isOver) {
359             count = 0;
360             /*
361               minOffset is the minimum offset greater than index. Its
362               default value is the deadline of the graph.
363              */
364             segOffset = getDeadline();
365             for (SubTask st : subtasks) {
366                 if (index >= st.getOffset()
367                         && index < (st.getEarliestFinishTime())) {
368                     count++;
369                     if (st.getEarliestFinishTime() < segOffset)
370                         segOffset = st.getEarliestFinishTime();
371                 }
372             }
373             if (count != 0) {
374                 psList.add(new ParallelSegmentInfo(count, (segOffset -
    index), index));
375                 index = segOffset;
376             } else {
377                 isOver = true;
378             }
379         }
380         return psList;
381     }
382 
383     /**
384      * Stretches the graph task to execute as sequentially as possible.
385      * The stretching is done by transforming the graph into a MTS task
386      * first.
387      * @return The master stretched thread with utilization <= 1 (based
388      *         on the utilization of stretched graph) and the resulting
389      *         constrained-deadline tasks.
390      */
```

```
391
392     public ArrayList<SubTask> stretchGraph() {
393         ArrayList<SubTask> list = new ArrayList<SubTask>();
394         if (getWcet() <= getDeadline()) {
395             list.add(new SubTask(getOffset(), getWcet(), getWcee(),
    getPeriod(), getDeadline(), getPriority(), 1,
    getEnergyConsumptionProfile()));
396         } else {
397             ArrayList<ParallelSegmentInfo> psInfo = transformToMTS();
398             int i = 1;
399             long seqWCET = 0;
400             boolean isOver = false;
401             long slack = getDeadline();
402             while (getDeadline() == getSeqWCETParaSegs(psInfo, i)) {
403                 i++;
404                 slack = 0;
405             }
406             int fullyStretched = i - 1;
407             if (slack != 0) {
408                 do {
409                     if (slack < seqWCET)
410                         isOver = true;
411                     else if (slack == 0) {
412                         isOver = true;
413                         fullyStretched++;
414                     } else {
415                         slack -= seqWCET;
416                         i++;
417                     }
418                 } while (!isOver);
419             }
420             for (int j = 0; j < fullyStretched; j++) {
421                 list.add(new SubTask(getOffset(), getDeadline(), getWcee()
    , getPeriod(), getDeadline(), getPriority(), 1,
    getEnergyConsumptionProfile()));
422             }
423             int count, extra = 0;
424             long currentSlack, nextOffset = psInfo.get(0).getOffset();
```

```
425             double wcetDist = ((double) slack) / seqWCET;
426             if (slack != 0) {
427                 /*
428                   Stretching is applied. Add resulting fully-stretched
429                   thread to list before performing the stretching.
430                  */
431                 list.add(new SubTask(getOffset(), getDeadline(), getWcee()
    , getPeriod(), getDeadline(), getPriority(), 1,
    getEnergyConsumptionProfile()));
432             }
433             for (ParallelSegmentInfo psi : psInfo) {
434                 count = i;
435                 currentSlack = 0;
436                 if (i <= psi.getThreadCount()) {
437                     if (slack != 0) {
438                         count++;
439                         extra = 0;
440                         long wcetToFull = (long) Math.ceil(wcetDist
441                                 * psi.getThreadWCET());
442                         if (wcetToFull > slack)
443                             wcetToFull = slack;
444                         long remWCET = psi.getThreadWCET() - wcetToFull;
445                         if (remWCET != 0) {
446                             SubTask subtask = new SubTask(nextOffset,
    remWCET, getWcee(), getPeriod(), (psi.getThreadWCET() * (i - 1)),
    getPriority(), 1, getEnergyConsumptionProfile());
447                             for (SubTask sub : list) {
448                                 if (sub.getDeadline() <= subtask.getOffset
    ())
449                                     subtask.addParentTask(sub);
450                             }
451                             list.add(subtask);
452                         }
453                         currentSlack = wcetToFull;
454                         slack -= wcetToFull;
455                     } else {
456                         extra = 0;
457                         currentSlack = 0;
```

```
458                     }
459                     long segDeadline = psi.getThreadWCET() * (i - 1 +
    extra)
460                             + currentSlack;
461                     for (int c = count; c <= psi.getThreadCount(); c++) {
462                         SubTask st = new SubTask(nextOffset, psi.
    getThreadWCET(), getWcee(), getPeriod(), segDeadline, getPriority(), 1,
     getEnergyConsumptionProfile());
463                         for (SubTask sub : list) {
464                             if (sub.getDeadline() <= st.getOffset())
465                                 st.addParentTask(sub);
466                         }
467                         list.add(st);
468                     }
469                     nextOffset += segDeadline;
470                 } else {
471                     nextOffset += psi.getThreadWCET() * (i - 1);
472                 }
473             }
474         }
475         return list;
476     }
477
478     /**
479      * Calculates the sequential length of WCET of parallel segments at
480      * level i (while considering the segments with number of threads
481      * greater than i).
482      * @param psInfo list of parallel segments which form the MTS task
483      * @param i the parallel level
484      * @return sequential length of WCET of MTS task at level i
485      */
486     private long getSeqWCETParaSegs(ArrayList<ParallelSegmentInfo> psInfo,
     int i) {
487         long seqWCET = 0;
488         for (ParallelSegmentInfo psi : psInfo) {
489             if (psi.getThreadCount() >= i)
490                 seqWCET += psi.getThreadWCET();
491         }
```

```
492             return seqWCET;
493         }
494 }
495
496
497 /**
498  * GraphTaskGenerator is a class that extends AbstractTaskGenerator and
499  * generates new instance of graph task. A Graph task consists of a set of
500  * subtasks and random directed relations.
501  * @author Manar Qamhieh
502  */
503
504 public class GraphTaskGenerator extends AbstractTaskGenerator {
505     private static final int MAX_SUBTASKS = 10;
506     private long start;
507     private long wcet;
508     private long deadline;
509     private long period;
510     private int priority;
511     private GraphTask graph;
512     private int MAX_PROC;
513     private double edges = 0.4;
514
515     @Override
516     protected ITask generate(long start, long wcet, long period, long
        deadline,
517             int priority, int maxProc) {
518         this.start = start;
519         this.wcet = wcet;
520         this.deadline = deadline;
521         this.period = period;
522         this.MAX_PROC = maxProc;
523         this.priority = priority;
524         ArrayList<SubTask> subtasks = new ArrayList<SubTask>();
525         subtasks = generateSubtasks();
526         graph = (GraphTask) SchedulableFactory.newInstance("graph", start,
527                 period, deadline, priority, subtasks, null);
528         generateDirectedRelations(graph);
```

```
529         graph.calculateCriticalPath();
530         return graph;
531     }
532 
533     /**
534      * Gets the probability of creating edges between subtasks in the
535      * generated graph
536      * @return the probability of edge creation
537      */
538     public double getEdgeFrequency() {
539         return edges;
540     }
541 
542     public int getMaxSubtaskCount() {
543         return MAX_SUBTASKS;
544     }
545 
546     /**
547      * Creates random edges between subtasks using a triangle matrix to
548      * ensure acyclic relations
549      * @param graph input graph task with a set of subtasks
550      */
551     private void generateDirectedRelations(GraphTask graph) {
552         int n = graph.getSubtasks().size();
553         int[][] matrix = new int[n][n];
554         /*
555          * Fill the triangle matrix randomly by zeros (no edges) and ones
556          * (edges) based on the probability of edge creation
557          */
558         for (int i = 0; i < n; i++) {
559             for (int j = (i + 1); j < n; j++) {
560                 double randVal = (double) random.nextInt() / Integer.
    MAX_VALUE;
561                 if (randVal < edges)
562                     matrix[i][j] = 0;
563                 else
564                     matrix[i][j] = 1;
565                 if (matrix[i][j] == 1) {
```

```
566                     /* check and remove any edge redundancies between
    subtasks */
567                     for (int k = 0; k < matrix.length; k++) {
568                         if (matrix[k][i] == 1 && matrix[k][j] == 1) {
569                             matrix[k][j] = 0;
570                         }
571                     }
572                 }
573             }
574         }
575 
576         /* create edges between subtasks based on the triangle matrix */
577         for (int i = 0; i < matrix.length; i++) {
578             for (int j = 0; j < matrix.length; j++) {
579                 if (matrix[i][j] == 1) {
580                     if (graph.canAddEdge(graph.getSubtasks().get(i), graph
581                             .getSubtasks().get(j))) {
582                         graph.addEdge(graph.getSubtasks().get(i), graph
583                                 .getSubtasks().get(j));
584                     }
585                 }
586             }
587         }
588     }
589 
590     /**
591      * Generates a set of subtasks with uniformed random distribution of
592      * WCET of graph task and number of subtasks
593      * @return a set of generated real-time subtasks
594      */
595     private ArrayList<SubTask> generateSubtasks() {
596         boolean discard;
597         ArrayList<SubTask> list = new ArrayList<SubTask>();
598         do {
599             list.clear();
600             discard = false;
601             /* The generated subtasks should respect the necessary
602       feasibility condition of graphs regarding their number and WCET.
```

```
603         The sequential worst-case execution behavior of subtasks should
604         not exceed its deadline. */
605               int numOfSubtasks;
606               int minNumSubtasks = (int) Math.ceil((double) wcet / deadline)
        ;
607               if (minNumSubtasks >= MAX_SUBTASKS)
608                   numOfSubtasks = minNumSubtasks;
609               else
610                   numOfSubtasks = nextInt(minNumSubtasks, MAX_SUBTASKS);
611               /* WCET execution time of a subtask should be at least 1 time
        unit */
612               if (wcet <= numOfSubtasks)
613                   numOfSubtasks = minNumSubtasks;
614               if (numOfSubtasks == 1) {
615                   SubTask subTask = (SubTask) SchedulableFactory.newInstance
616             ("subtask", start, wcet, 0, period, deadline,
617             priority, 1, null);
618                   subTask.setIndex(list);
619                   list.add(subTask);
620                   if (subTask.getUtilization() < 1)
621                       return list;
622               } else {
623                   /* uniform distribution of WCET when subtasks > 1 */
624                   long sumWCET = wcet;
625                   long subWCET;
626                   long maxSub = deadline;
627                   long minSub = 0;
628                   double rand;
629                   for (int i = 1; i < numOfSubtasks; i++) {
630                       minSub = sumWCET - (maxSub * (numOfSubtasks - i));
631                       /* WCET of any subtasks should be at least 1 unit */
632                       if (sumWCET == 1)
633                           break;
634                       do {
635                           do {
636                               rand = random.nextDouble();
637                           } while (rand == 0);
638                           double tmp = sumWCET
```

```
639                                     * Math.pow(rand,
640                                             ((double) 1 / (numOfSubtasks - i))
    );
641                             if (tmp < 1)
642                                 tmp++;
643                             subWCET = sumWCET - (long) Math.floor(tmp);
644                         } while (subWCET == 0 || subWCET > maxSub
645                                 || subWCET < minSub);
646                 }
647                 SubTask subTask = (SubTask) SchedulableFactory.newInstance
    (
648                         "subtask", start, sumWCET, 0, period, deadline,
649                         priority, 1, null);
650
651                 subTask.setIndex(list);
652                 list.add(subTask);
653             }
654             for (SubTask subTask : list) {
655                 if (subTask.getUtilization() > 1) {
656                     discard = true;
657                     break;
658                 }
659             }
660         } while (discard);
661         return list;
662     }
663 }
```

Appendix B

Subtasks in YARTISS Simulator

```
1 /**
2  * Real-time subtask for the graph model. Mainly, subtasks share a lot of
3  * timing features of the classical sequential real-time tasks . However,
4  * they contain extra information regarding inter-subtask dependencies.
5  * @author Manar Qamhieh
6  */
7 public class SubTask extends PeriodicTask {
8     private ArrayList<SubTask> parentSubtasks;
9     private ArrayList<SubTask> childrenSubtasks;
10    private long earliestFinishTime;
11    private long latestFinishTime;
12    private int nbProc;
13    private List<Integer> indices;
14    private long graphLaxity;
15    private long jitter = 0, modJitter = 0;
16
17    public SubTask(long localOffset, long wcet, long wcee, long period,
      long deadline, int priority, int nbProc, IEnergyConsumptionProfile p) {
18        super(localOffset, wcet, wcee, period, deadline, priority, p);
19        this.nbProc = nbProc;
20        parentSubtasks = new ArrayList<SubTask>();
21        childrenSubtasks = new ArrayList<SubTask>();
22        indices = new ArrayList<Integer>();
23    }
24
25    public SubTask(SubTask subtask) {
```

```
26          this(subtask.getOffset(), subtask.getWcet(), subtask.getWcee(),
        subtask.getPeriod(), subtask.getDeadline(), subtask.getPriority(),
        subtask.getNumOfProc(), subtask.getEnergyConsumptionProfile());
27          setMessage(subtask.getMessage());
28      }
29
30      /**
31       * Sets the index of a subtask in the graph according to the index of
32       * the last subtask in the list.
33       * @param list list of all subtasks
34       */
35      public void setIndex(ArrayList<SubTask> list) {
36          if (list.isEmpty())
37              setId(0);
38          else
39              setId(list.get(list.size() - 1).getId() + 1);
40      }
41
42      public void setIndex(CopyOnWriteArrayList<SubTask> list) {
43          setId(list.get(list.size() - 1).getId() + 1);
44      }
45
46      /**
47       * Gets the parent subtasks of the subtask
48       * @return parent subtasks
49       */
50      public ArrayList<SubTask> getParentSubtasks() {
51          return parentSubtasks;
52      }
53
54      /**
55       * Sets the parent subtasks of the subtask
56       * @param parents parent subtasks
57       */
58      public void setParentSubtasks(ArrayList<SubTask> parents) {
59          for (SubTask subTask : parents) {
60              this.parentSubtasks.add(subTask);
61          }
```

```
62      }
63
64      /**
65       * Gets the children subtasks of the subtask
66       * @return children subtasks
67       */
68      public ArrayList<SubTask> getChildrenSubtasks() {
69          return childrenSubtasks;
70      }
71
72      /**
73       * Sets the children subtasks of the subtask
74       * @param children children subtasks
75       */
76      public void setChildrenTasks(ArrayList<SubTask> children) {
77          this.childrenSubtasks = children;
78      }
79
80      /**
81       * Adds a 'child' subtask as a successor to the current subtask
82       * @param child successor subtask
83       */
84      public void addChildTask(SubTask child) {
85          childrenSubtasks.add(child);
86          child.getParentSubtasks().add(this);
87      }
88
89      /**
90       * Adds a 'parent' subtask as a predecessor of the current subtask
91       * @param parent predecessor subtask
92       */
93      public void addParentTask(SubTask parent) {
94          parentSubtasks.add(parent);
95          parent.getChildrenSubtasks().add(this);
96      }
97
98      /**
99       * Gets the earliest time of a subtask to finish its execution
```

```
100      * @return earliest execution time
101      */
102     public long getEarliestFinishTime() {
103         return earliestFinishTime;
104     }
105 
106     /**
107      * Sets the earliest time of a subtask to finish its execution
108      * @param time earliest execution time
109      */
110     public void setEarliestFinishTime(long time) {
111         this.earliestFinishTime = time;
112     }
113 
114     /**
115      * Gets the latest time of a subtask to finish its execution
116      * @return latest execution time
117      */
118     public long getLatestFinishTime() {
119         return latestFinishTime;
120     }
121 
122     /**
123      * Sets the latest time of a subtask to finish its execution
124      * @param time latest execution time
125      */
126     public void setLatestFinisheTime(long time) {
127         this.latestFinishTime = time;
128     }
129 
130     /**
131      * Gets the number of processors on which a subtask can execute
132      * @return number of processors
133      */
134     public int getNumOfProc() {
135         return nbProc;
136     }
137 
```

```
138     /**
139      * Sets the number of processors on which a subtask can execute
140      * @param numOfProc number of processors
141      */
142     public void setNumOfProc(int nbProc) {
143         this.nbProc = nbProc;
144     }
145 
146     /**
147      * Verifies if the subtask is one of critical path subtasks
148      * @return true if it is critical subtask, false otherwise
149      */
150     public boolean isCriticalTask() {
151         if (earliestFinishTime == latestFinishTime)
152             return true;
153         else
154             return false;
155     }
156 
157     /**
158      * Checks if the subtask is the starting subtask of the graph, which
159      * is the only subtask in the graph which has no parents.
160      * @return true if it is the starting of a graph, false otherwise
161      */
162     public boolean isSourceSubtask() {
163         return parentSubtasks.isEmpty();
164     }
165 
166     /**
167      * Checks if the subtask is the ending subtask of the graph, which
168      * is the only subtask in the graph which has no children.
169      * @return true if it is the ending of a graph, false otherwise
170      */
171     public boolean isSinkSubtask() {
172         return childrenSubtasks.isEmpty();
173     }
174 
175     public String toString() {
```

```
176         return "[" + getId() + "] wcet:\t" + getWcet() + "\toffset:\t"+
        getOffset() + "\tlatestDeadline:\t" + getLatestFinishTime() + " \
        tdeadline:\t" + getDeadline() + "\trelease Jitter:\t" +
        getReleaseJitter();
177     }
178
179     public void accept(ITaskElementVisitor visitor) {
180         visitor.visitSubTask(this);
181     }
182
183     @Override
184     public ITask cloneTask() {
185         return new SubTask(this);
186     }
187
188     public void setChildrenIndices(String[] indices) {
189         if (indices == null)
190             this.indices = null;
191         for (String childId : indices) {
192             this.indices.add(Integer.parseInt(childId));
193         }
194     }
195
196     public List<Integer> getChildrenIndices() {
197         return indices;
198     }
199
200     /**
201      * Laxity of a subtask is the difference between the finish time of
202      * this subtask after calculating the critical path.
203      */
204     @Override
205     public long getLaxity() {
206         long laxity = getLatestFinishTime() - getOffset() -
        getRemainingCost();
207         if (latestFinishTime == 0 || laxity < 0)
208             throw new IllegalArgumentException(
```

```
209                     "Critical path should  be calculated before using
    finish time");
210         return laxity;
211     }
212
213     /**
214      * @return the laxity of the graph to which this subtask belongs
215      */
216     public long getGraphLaxity() {
217         return graphLaxity;
218     }
219
220     /**
221      * @param graphLaxity the graph's laxity to set
222      */
223     public void setGraphLaxity(long graphLaxity) {
224         this.graphLaxity = graphLaxity;
225     }
226
227     @Override
228     public long getNextAbsoluteDeadline(long time) {
229         long currentPeriod = (time - getOffset()) / getPeriod();
230         return currentPeriod * getPeriod() + getLatestFinishTime();
231     }
232
233     /**
234      * Gets the release jitter of the current subtasks, i.e., the length
235      * of the activation interval based on its precedence constraints.
236      * @return the release jitter of the subtask
237      */
238     public long getReleaseJitter() {
239         if (this.isSourceSubtask())
240             return 0;
241         if (this.jitter == 0)
242             calculateReleaseJitter();
243         return this.jitter;
244     }
245
```

```
246     /**
247      * Calculates the release jitter of the current subtask based on its
248      * local timing parameters and its precedence constraints.
249      */
250     private void calculateReleaseJitter() {
251         this.jitter = 0;
252         for (SubTask parent : getParentSubtasks()) {
253             long tmp = parent.getLatestFinishTime() - getOffset();
254             if (tmp > this.jitter)
255                 this.jitter = tmp;
256         }
257     }
258 
259     public long getModifiedJitter() {
260         return modJitter;
261     }
262 
263     public void setModifiedJitter(long jitter) {
264         this.modJitter = jitter;
265     }
266 
267     /**
268      * Checks if subtask "possiblePred" is a predecessor of the current
269      * subtask or not.
270      * @param possiblePred the verified subtask
271      * @return True if possiblePred is a predecessor, False otherwise
272      */
273     public boolean isPredecessor(SubTask possiblePred) {
274         if (this.getParentSubtasks().contains(possiblePred))
275             return true;
276         for (SubTask parent : this.getParentSubtasks()) {
277             if (parent.isPredecessor(possiblePred))
278                 return true;
279         }
280         return false;
281     }
282 
283     /**
```

```
284      * Checks if subtask "possibleSucc" is a successor of the current
285      * subtask or not.
286      * @param possibleSucc the verified subtask
287      * @return True if possibleSucc is a successor, False otherwise.
288      */
289     public boolean isSuccessor(SubTask possibleSucc) {
290         if (this.getChildrenSubtasks().contains(possibleSucc))
291             return true;
292         for (SubTask child : this.getChildrenSubtasks()) {
293             if (child.isSuccessor(possibleSucc))
294                 return true;
295         }
296         return false;
297     }
298 
299     public long getNextLocalDeadline(long time, boolean nextPeriod) {
300         if (time < getOffset())
301             return getOffset() + getLatestFinishTime();
302         long currentPeriod = (time - getOffset()) / getPeriod();
303         if (nextPeriod && getRemainingCost() == 0)
304             currentPeriod++;
305         return getOffset() + currentPeriod * getPeriod()
306                 + getLatestFinishTime();
307     }
308 }
```

Author's publication list

Journals and Reviews

ACR'14 M. Qamhieh and S. Midonnet. Simulation-Based Evaluations of DAG Scheduling in Hard Real-time Multiprocessor Systems. *Applied Computing Review*, 14(4):27–39, 2014

International Conference Papers

RTNS'14 M. Qamhieh, L. George, and S. Midonnet. A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 13:13–13:22. ACM, 2014

RACS'14 M. Qamhieh and S. Midonnet. An Experimental Analysis of DAG Scheduling Methods in Hard Real-Time Multiprocessor Systems. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pages 284–290. ACM, 2014

ADA'14 M. Qamhieh and S. Midonnet. Schedulability Analysis for Directed Acyclic Graphs on Multiprocessor Systems at a Subtask Level. In *Proceedings of the 19th International Conference on Reliable Software Technologies*, Ada-Europe, 2014

RTNS'13 M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 287–296. ACM, 2013

Workshop and WIP[1] Papers

JSSPP'14 M. Qamhieh and S. Midonnet. Experimental Analysis of the Tardiness of Parallel Tasks in Soft Real-time Systems. In *18th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, JSSPP, 2014

PETARS'12 M. Qamhieh, S. Midonnet, and L. George. Graph-to-Segment Transformation Technique minimizing the number of processors for Real-time Multiprocessor Systems. In *Workshop on Power, Energy, and Temperature Aware Real-time Systems (PETARS)*, page 6pp, San Juan, Peurto Rico, Dec. 2012

ECRTS'12-WiP M. Qamhieh, S. Midonnet, and L. George. Dynamic Scheduling Algorithm for Parallel Real-time Graph Tasks. *SIGBED Rev.*, 9(4):25–28, Nov. 2012. ISSN 1551-3688

WATERS'12 Y. Chandarli, F. Fauberteau, M. Damien, S. Midonnet, and M. Qamhieh. YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms. In *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2012

RTAS'12-WiP M. Qamhieh, S. Midonnet, and L. George. A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model. In *The 18th IEEE Real-Time and Embedded Technology and Applications Symposium Work-in-Progress (WiP) Proceedings*, pages 45–48, Beijing, Chine, Apr. 2012

JRWRTC'11 M. Qamhieh, F. Fauberteau, and S. Midonnet. Performance Analysis for Segment Stretch Transformation of Parallel Real-time Tasks. In *Proceedings of the 2th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, page 4pp, Nantes, France, Sept. 2011

ECRTS'11-WiP F. Fauberteau, S. Midonnet, and M. Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. *SIGBED Rev.*, 8(3):28–31, Sept. 2011

[1] Work In Progress

Technical Reports

Tech'14 Y. Chandarli, M. Qamhieh, F. Fauberteau, and M. Damien. YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems. Technical report, UPE LIGM ESIEE, 2014

Bibliography

[1] OpenMP Application Program Interface version 3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf. (Cited on pages 22 and 39.)

[2] K. Agrawal, C. Gill, J. Li, M. Mahadevan, D. Ferry, and C. Lu. A Real-time Scheduling Service for Parallel Tasks. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS, pages 261–272. IEEE Computer Society, 2013. (Cited on page 39.)

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. (Cited on page 17.)

[4] J. H. Anderson and A. Srinivasan. Early-Release Fair Scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems (ECRTS)*, pages 35–43. IEEE Computer Society, June 2000. (Cited on page 30.)

[5] J. H. Anderson and A. Srinivasan. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85, 2001. doi: 10.1109/EMRTS.2001.934004. (Cited on page 30.)

[6] B. Andersson and D. D. Niz. Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks. In *Proceedings of the 16th International Conference On Principles Of DIstributed Systems (OPODIS)*, pages 16–30, 2012. (Cited on page 45.)

[7] B. Andersson, S. K. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193–202, Dec 2001. doi: 10.1109/REAL.2001.990610. (Cited on pages 29 and 30.)

[8] N. C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report YCS 164, University of York, Department of Computer Science, York, UK, Dec. 1991. (Cited on pages 28 and 44.)

[9] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(May 1999):39–44, 2001. (Cited on page 28.)

[10] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 215–224, July 2013. doi: 10.1109/ECRTS.2013.31. (Cited on pages 44 and 45.)

[11] T. P. Baker. Multiprocessor EDF and Deadline Monotonic Schedulability Analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 120–129. IEEE Computer Society, Dec. 2003. (Cited on page 45.)

[12] T. P. Baker. An Analysis of EDF Schedulability on a Multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768, Aug 2005. ISSN 1045-9219. doi: 10.1109/TPDS.2005.88. (Cited on page 30.)

[13] T. P. Baker and S. K. Baruah. Sustainable Multiprocessor Scheduling of Sporadic Task Systems. In *21st Euromicro Conference on Real-Time Systems*, pages 141–150. Ieee, July 2009. (Cited on pages 105, 106, 107, 108, and 109.)

[14] S. K. Baruah. Feasibility analysis of recurring branching tasks. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems, 1998*, pages 138–145, 1998. (Cited on page 19.)

[15] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003. (Cited on page 19.)

[16] S. K. Baruah. Optimal Utilization Bounds for the Fixed-Priority Scheduling of Periodic Task Systems on Identical Multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, June 2004. (Cited on page 30.)

[17] S. K. Baruah. Techniques for Multiprocessor Global Schedulability Analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 119–128. Ieee, Dec. 2007. (Cited on pages 45 and 68.)

[18] S. K. Baruah and A. Burns. Sustainable scheduling analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168. IEEE Computer Society, Dec. 2006. (Cited on pages 9 and 105.)

[19] S. K. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 9 pp.–329, Dec 2005. doi: 10.1109/RTSS.2005.40. (Cited on page 112.)

[20] S. K. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 360–366. IEEE Computer Society, May 2003. (Cited on page 105.)

[21] S. K. Baruah, A. K.-L. Mok, and L. E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190. IEEE Computer Society, Dec. 1990. (Cited on page 5.)

[22] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time tasks on one processor. *Real-Time Systems*, 2(4):301–324, Nov. 1990. (Cited on page 5.)

[23] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, pages 280–288. IEEE Computer Society 1995, Apr. 1995. (Cited on page 30.)

[24] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6):600–625, June 1996. (Cited on page 30.)

[25] S. K. Baruah, D. Chen, S. Gorinsky, and A. K.-L. Mok. Generalized Multiframe Tasks. *Real-Time Systems*, 17(1):5–22, July 1999. (Cited on page 18.)

[26] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved Multiprocessor Global Schedulability Analysis. *Real-Time Systems*, 46(1):3–24, Sept. 2010. (Cited on pages 45 and 49.)

[27] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A Generalized Parallel Task Model for Recurrent Real-time Processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, pages 63–72. IEEE Computer Society, Dec. 2012. (Cited on pages 47, 49, 87, 90, and 179.)

[28] M. Bertogna. *Real-time Scheduling Analysis for Multiprocessor Platforms*. PhD thesis, Scuola Superiore Sant'Anna, 2007. (Cited on page 30.)

[29] M. Bertogna and M. Cirinei. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 149–160. Ieee, Dec. 2007. (Cited on page 45.)

[30] M. Bertogna, M. Cirinei, and G. Lipari. Improved Schedulability Analysis of EDF on Multiprocessor Platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218, July 2005. doi: 10.1109/ECRTS.2005.18. (Cited on pages 45 and 115.)

[31] M. Bertogna, M. Cirinei, and G. Lipari. New Schedulability Tests for Real-time Task Sets Scheduled by Deadline Monotonic on Multiprocessors. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 306–321, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36321-1, 978-3-540-36321-7. doi: 10.1007/11795490_24. (Cited on page 29.)

[32] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, 2009. (Cited on pages 100, 101, 121, 122, 124, and 126.)

[33] E. Bini and G. C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, May 2005. (Cited on pages 81 and 149.)

[34] V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. A Constant-Approximate Feasibility Test for Multiprocessor Real-Time Scheduling. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, volume 5193/2008, pages 210–221. Springer, Sept. 2008. (Cited on page 45.)

[35] V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. A Constant-Approximate Feasibility Test for Multiprocessor Real-Time Scheduling. *Algorithmica*, 62(3-4):1034–1049, 2012. ISSN 0178-4617. doi: 10.1007/s00453-011-9497-2. (Cited on page 49.)

[36] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility Analysis in the Sporadic DAG Task Model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 225–233, July 2013. doi: 10.1109/ECRTS.2013.32. (Cited on pages 49, 87, 132, 133, and 179.)

[37] W. J. Bouknight, S. Denenberg, D. McIntyre, J. M. Randall, A. Sameh, and D. Slotnick. The Illiac IV system. *In Proceedings of the IEEE*, 60(4):369–388, April 1972. ISSN 0018-9219. doi: 10.1109/PROC.1972.8647. (Cited on page 29.)

[38] A. Burns and S. K. Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering (JCSE)*, 2(1):74–97, Mar. 2008. (Cited on page 105.)

[39] Y. Chandarli, F. Fauberteau, M. Damien, S. Midonnet, and M. Qamhieh. YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms. In *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2012. (Cited on page 145.)

[40] Y. Chandarli, M. Qamhieh, F. Fauberteau, and M. Damien. YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems. Technical report, UPE LIGM ESIEE, 2014. (Not cited.)

[41] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-time Tasks Under Precedence Constraints. *Real-Time Systems*, 2(3):181–194, Sept. 1990. ISSN 0922-6443. doi: 10.1007/BF00365326. (Cited on page 36.)

[42] M. Chetto, D. Masson, and S. Midonnet. Fixed Priority Scheduling Strategies for Ambient Energy-Harvesting Embedded Systems. In *Proceedings of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, pages 50–55. IEEE Computer Society / ACM Computer Press, Aug. 2011. (Cited on page 146.)

[43] H. Cho, B. Ravindran, and D. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 101–110. IEEE Computer Society, Dec. 2006. (Cited on page 30.)

[44] H. S. Chwa, J. Lee, K.-m. Phan, A. Easwaran, and I. Shin. Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms. In *Proceedings of the 25th euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34, 2013. (Cited on pages 45, 46, 87, and 179.)

[45] S. Collette, L. Cucu, and J. Goossens. Algorithm and Complexity for the Global Scheduling of Sporadic Tasks on Multiprocessors with Work-Limited Parallelism. In *Proceedings of the 15th International Conference on Real-Time and Network systems (RTNS)*, 2007. (Cited on page 38.)

[46] S. Collette, L. Cucu, and J. Goossens. Integrating Job Parallelism in Real-Time Scheduling Theory. *Information Processing Letters*, 106:180–187, 2008. (Cited on page 38.)

[47] P. Courbin and L. George. FORTAS : Framework fOr Real-Time Analysis and Simulation. In *Proceedings of 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, pages 21–26, July 2011. (Cited on page 144.)

[48] R. I. Davis and A. Burns. A Survey of Hard Real-time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems. *ACM Computing surveys*, pages 1 – 44, 2011. (Cited on page 31.)

[49] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511. (Cited on page 11.)

[50] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In J. L. Rosenfeld, editor, *Information Processing*, pages 807–813, Stockholm, Sweden, Aug. 1974. North-Holland, American Elsevier. (Cited on page 31.)

[51] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1): 127–140, 1978. (Cited on pages 15 and 29.)

[52] F. Fauberteau. RTMSIM. `http://rtmsim.triaxx.org/`. (Cited on page 146.)

[53] F. Fauberteau, S. Midonnet, and L. George. Laxity-Based Restricted-Migration Scheduling. In Z. Mammeri, editor, *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE Computer Society, Sept. 2011. (Cited on page 146.)

[54] F. Fauberteau, S. Midonnet, and M. Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. *SIGBED Rev.*, 8(3):28–31, Sept. 2011. (Cited on page 41.)

[55] D. G. Feitelson. *A Survey of Scheduling in Multiprogrammed Parallel Systems*. IBM TJ Watson Research Center, 1994. (Cited on page 37.)

[56] D. G. Feitelson. Packing Schemes for Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 89–110. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61864-5. doi: 10.1007/BFb0022289. (Not cited.)

[57] D. G. Feitelson and L. Rudolph. Parallel Job Scheduling: Issues and Approaches. In *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer, 1995. (Cited on page 37.)

[58] N. W. Fisher. *The multiprocessor real-time scheduling of general task systems*. PhD thesis, University of North Carolina, 2007. (Cited on page 16.)

[59] N. W. Fisher, S. K. Baruah, and T. P. Baker. The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities. In *Proceedings of the 18th Euromicro Conference on Real-time Systems (ECRTS)*, pages 118–127. IEEE Computer Society, July 2006. (Cited on page 41.)

[60] J. Goossens and V. Berten. Gang FTP Scheduling of Periodic and Parallel Rigid Real-time Tasks. In *Proceedings of the 18th Real-Time and Network Systems (RTNS)*, pages 189–196. IRIT Press, Nov. 2010. (Cited on pages 37 and 38.)

[61] J. Goossens and R. Devillers. The Non-Optimality of the Monotonic Priority Assignmentsfor Hard Real-Time Offset Free Systems. *Real-Time Systems*, 13(2):107–126, Sept. 1997. ISSN 0922-6443. doi: 10.1023/A:1007980022314. (Cited on page 149.)

[62] J. Goossens and C. Macq. Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation. In *Proceedings of the 9th International Conference on Real-Time Systems (RTS)*, pages 133–148, Mar. 2001. (Cited on pages 81 and 149.)

[63] J. Goossens, S. Funk, and S. K. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Journal of Real-Time Systems*, 25(2-3):187–205, Sept. 2003. ISSN 0922-6443. doi: 10.1023/A:1025120124771. (Cited on page 30.)

[64] V. R. Group. Embedded software and tools market intelligence service. In *Multicore Components and Tools*, volume 5, 2011. (Cited on page 13.)

[65] R. Ha. *Validating timing constraints in multiprocessor and distributed real-time systems*. PhD thesis, University of Illinois, Dept. of Computer Science, Urbana-Champaign, IL, USA, 1995. (Cited on page 107.)

[66] R. Ha and J. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *14th International Conference on Distributed Computing Systems*, pages 162–171. IEEE Comput. Soc. Press, 1994. (Cited on pages 25 and 107.)

[67] C.-C. Han and K.-J. Lin. Scheduling Parallelizable Jobs on Multiprocessors. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 59–67, 1989. (Cited on page 37.)

[68] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed Priority Scheduling Periodic Tasks with Varying Execution Priority. In *Proceedings of the 12th IEEE Real-Time Systems Symposium (RTSS)*, pages 116–128, 1991. (Cited on pages 32, 33, and 34.)

[69] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing Analysis for Fixed Priority Scheduling of Hard Real-Time Systems. In *IEEE Transactions on Software Engineering*, pages 13–28, 1994. (Cited on page 32.)

[70] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano. MAST: Modeling and Analysis Suite for Real-Time Applications. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, 2001. (Cited on page 144.)

[71] P. Jayachandran and T. Abdelzaher. A Delay Composition Theorem for Real-Time Pipelines. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 29–38, 2007. (Cited on page 36.)

[72] P. Jayachandran and T. Abdelzaher. Transforming Distributed Acyclic Systems into Equivalent Uniprocessors under Preemptive and Non-Preemptive Scheduling. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 233–242, 2008. (Cited on pages 35 and 36.)

[73] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance [scheduling problems]. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 214–221, 1995. (Cited on page 9.)

[74] S. Kato and Y. Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 459–468, Dec 2009. doi: 10.1109/RTSS.2009.42. (Cited on pages 37 and 38.)

[75] A. Khemka and R. K. Shyamasundar. An Optimal Multiprocessor Real-Time Scheduling Algorithm. *Journal of Parallel and Distributed Computing*, 43:37–45, 1997. (Cited on page 30.)

[76] G. Koren and D. Shasha. D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems. *Society for Industrial and Applied Mathematics Journal*, 24(2): 318–339, Apr. 1995. ISSN 0097-5397. (Cited on page 146.)

[77] K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*, pages 259–268. IEEE Computer Society, 2010. (Cited on pages 23, 39, 40, 41, 54, 86, and 176.)

[78] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, Dec 1990. doi: 10.1109/REAL.1990.128748. (Cited on pages 4 and 28.)

[79] J. Y.-T. Leung and M. Merrill. A Note on Preemptive Scheduling of Periodic, Real-time Tasks. *Information Processing Letters*, 11(3):115 – 118, 1980. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(80)90123-4. (Cited on page 149.)

[80] J. Y.-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, Dec. 1982. (Cited on page 28.)

[81] J. Li, K. Agrawal, C. Lu, and C. Gill. Analysis of Global EDF for Parallel Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, number 314, 2013. (Cited on pages 47, 48, 49, 87, 88, 90, 98, 105, 130, 131, 133, and 179.)

[82] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014. (Cited on pages 24, 87, 166, 177, and 179.)

[83] C. L. Liu. Scheduling Algorithms for Hard Real-Time Programming of a Single Processor. *JPL Space Programs Summary 37-60*, II:31–37, 1969. (Cited on page 31.)

[84] C. L. Liu. Scheduling Algorithms for Multiprocessors in a Hard Real-time Environment. *JPL Space Programs Summary 37-60*, II:28–31, 1969. (Cited on page 29.)

[85] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. (Cited on pages 3, 18, 27, 28, 31, and 148.)

[86] I. Lupu and J. Goossens. Scheduling of Hard Real-Time Multi-Thread Periodic Tasks. In *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS)*, pages 35–44, 2011. (Cited on page 46.)

[87] T. Machines. Inc. The Connection Machine CM-5 Technical Summary. *Thinking Machines Corporation*, 1991. (Cited on page 38.)

[88] D. Masson. RTSS v1 and v2. `https://svnigm.univ-mlv.fr/projects/rtsimulator/`. (Cited on pages 145 and 146.)

[89] D. Masson and S. Midonnet. Userland Approximate Slack Stealer with Low Time Complexity. In G. C. Buttazzo and P. Minet, editors, *Proceedings of the 16th Real-Time and Network Systems (RTNS)*, pages 29–38, Oct. 2008. (Cited on page 146.)

[90] D. Masson and S. Midonnet. Handling non-periodic events in real-time java systems. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 45–77. Springer US, 2012. ISBN 978-1-4419-8158-5. (Cited on page 146.)

[91] A. Mok and D. Chen. A multiframe model for real-time tasks. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 22–29, Dec 1996. doi: 10.1109/REAL.1996.563696. (Cited on page 18.)

[92] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8): 114, 1965. (Cited on page 11.)

[93] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques Optimizing the Number of Processors to Schedule Multi-threaded Tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 321–330, July 2012. doi: 10.1109/ECRTS.2012.37. (Cited on page 43.)

[94] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. U-EDF: An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–23, July 2012. doi: 10.1109/ECRTS.2012.36. (Cited on page 31.)

[95] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal Time-critical Scheduling via Resource Augmentation (Extended Abstract). In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC, pages 140–149, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258570. (Cited on pages 15 and 49.)

[96] M. Qamhieh and S. Midonnet. Schedulability Analysis for Directed Acyclic Graphs on Multiprocessor Systems at a Subtask Level. In *Proceedings of the 19th International Conference on Reliable Software Technologies*, Ada-Europe, 2014. (Not cited.)

[97] M. Qamhieh and S. Midonnet. Experimental Analysis of the Tardiness of Parallel Tasks in Soft Real-time Systems. In *18th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, JSSPP, 2014. (Cited on page 168.)

[98] M. Qamhieh and S. Midonnet. An Experimental Analysis of DAG Scheduling Methods in Hard Real-Time Multiprocessor Systems. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pages 284–290. ACM, 2014. (Not cited.)

[99] M. Qamhieh and S. Midonnet. Simulation-Based Evaluations of DAG Scheduling in Hard Real-time Multiprocessor Systems. *Applied Computing Review*, 14(4):27–39, 2014. (Not cited.)

[100] M. Qamhieh, F. Fauberteau, and S. Midonnet. Performance Analysis for Segment Stretch Transformation of Parallel Real-time Tasks. In *Proceedings of the 2th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, page 4pp, Nantes, France, Sept. 2011. (Not cited.)

[101] M. Qamhieh, S. Midonnet, and L. George. A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model. In *The 18th IEEE Real-Time and Embedded Technology and Applications Symposium Work-in-Progress (WiP) Proceedings*, pages 45–48, Beijing, Chine, Apr. 2012. (Cited on page 169.)

[102] M. Qamhieh, S. Midonnet, and L. George. Dynamic Scheduling Algorithm for Parallel Real-time Graph Tasks. *SIGBED Rev.*, 9(4):25–28, Nov. 2012. ISSN 1551-3688. (Cited on page 169.)

[103] M. Qamhieh, S. Midonnet, and L. George. Graph-to-Segment Transformation Technique minimizing the number of processors for Real-time Multiprocessor Systems. In *Workshop on Power, Energy, and Temperature Aware Real-time Systems (PETARS)*, page 6pp, San Juan, Peurto Rico, Dec. 2012. (Cited on page 44.)

[104] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 287–296. ACM, 2013. (Not cited.)

[105] M. Qamhieh, L. George, and S. Midonnet. A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 13:13–13:22. ACM, 2014. (Not cited.)

[106] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, Nov. 2011. (Cited on page 30.)

[107] M. Richard, P. Richard, E. Grolleau, and F. Cottet. Contraintes de Precedences et Ordonnancement Monoprocesseur. *The 10 RTS Embedded Systems*, pages 121–138, 2002. (Cited on pages 34 and 35.)

[108] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 217–226, 2011. (Cited on pages 42, 68, and 176.)

[109] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill. Parallel Real-Time Scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2014. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.2297919. (Cited on pages 42, 43, and 68.)

[110] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a Flexible Real-Time Scheduling Framework. *ACM SIGAda Ada Letters*, 24(4):1–8, Dec. 2004. (Cited on page 144.)

[111] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the Usability of Real-time Scheduling Theory with the Cheddar Project. *Real-Time Systems*, 43(3):259–295, 2009. (Cited on page 144.)

[112] A. Srinivasan and S. K. Baruah. Deadline-based Scheduling of Periodic Task Systems on Multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/S0020-0190(02)00231-4. (Cited on page 30.)

[113] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The Digraph Real-Time Task Model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 71–80, April 2011. doi: 10.1109/RTAS.2011.15. (Cited on pages 31 and 37.)

[114] M. Stigge, P. Ekberg, N. Guan, and W. Yi. On the Tractability of Digraph-Based Task Models. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 162–171, July 2011. doi: 10.1109/ECRTS.2011.23. (Cited on page 37.)

[115] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. (Cited on page 10.)

[116] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10 (3):384–393, June 1975. ISSN 0022-0000. doi: 10.1016/S0022-0000(75)80008-0. (Cited on page 37.)

[117] S. H. Unger. A Computer Oriented toward Spatial Problems. *In Proceedings of the IRE*, 46(10): 1744–1750, Oct 1958. ISSN 0096-8390. doi: 10.1109/JRPROC.1958.286755. (Cited on page 28.)

[118] R. Urunuela, A.-M. Déplanche, and Y. Trinquet. STORM: a Simulation Tool for Real-time Multiprocessor Scheduling Evaluation. *Workshop of GDR SOC SIP*, page 1, 2009. (Cited on page 144.)

[119] H. Zeng and M. Di Natale. Using Max-Plus Algebra to Improve the Analysis of Non-cyclic Task Models. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 205–214, July 2013. doi: 10.1109/ECRTS.2013.30. (Cited on pages 19 and 44.)

[120] H. X. Zhao, S. Midonnet, and L. George. Worst-Case Response Time Analysis of Sporadic Graph Tasks with Fixed Priority Scheduling on a Uniprocessor. In *Proceedings of Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2005. (Cited on page 35.)

[121] H.-X. Zhao, L. George, and S. Midonnet. Worst-Case Response Time Analysis of Sporadic Graph Tasks with EDF Scheduling on a Uniprocessor. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006.*, pages 271–278, 2006. (Cited on page 35.)

[122] D. Zhu, D. Mosse, and R. Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 142–151. IEEE Computer Society, Dec. 2003. (Cited on page 30.)

Scheduling of Parallel Real-time DAG Tasks on Multiprocessor Systems

By Manar Qamhieh

Abstract: We study the problem of real-time scheduling of parallel Directed Acyclic Graph (DAG) tasks on homogeneous multiprocessor systems. In this model, a DAG task consists of a set of subtasks that execute under precedence constraints. At all times, the real-time scheduler is responsible for determining how subtasks execute, either sequentially or in parallel, based on the available processors of the system.

We propose two DAG scheduling approaches to determine the execution form of DAG tasks. The first approach is the DAG Stretching algorithm, from the Model Transformation approach, which forces DAG tasks to execute as sequentially as possible.

The second approach is the Direct Scheduling, which aims at scheduling DAG tasks while respecting their internal dependencies and maintaining the parallel form of DAGs. We provide real-time schedulability analyses for Direct Scheduling at DAG-Level and at Subtask-Level.

Keywords: Hard Real-time Scheduling, Multiprocessor Systems, Parallel Applications, Inter-Subtask Parallelism, Global Preemptive Scheduling, Earliest Deadline First Scheduling Algorithm.