
# Multiprocessor Scheduling and Mapping Techniques for Real-Time Parallel Applications

**José Carlos Nunes da Fonseca**

Supervisor: Vincent Michel Philippe Nélis

Co-Supervisor: Luís Miguel Rosário da Silva Pinho

Programa Doutoral em Engenharia Electrotécnica e de Computadores

January 24, 2019

Faculdade de Engenharia da Universidade do Porto

# Multiprocessor Scheduling and Mapping Techniques for Real-Time Parallel Applications

**José Carlos Nunes da Fonseca**

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

**Doctor Philosophiae in Electronic & Computer Engineering**

Approved by:

President: José Alfredo Ribeiro da Silva Matos
External Referee: Robert Davis
External Referee: Eduardo Quiñones Moreno
FEUP Referee: Luís Miguel Pinho de Almeida
FEUP Referee: Pedro Alexandre Guimarães Lobo Ferreira Souto
Supervisor: Vincent Michel Philippe Nélis

January 24, 2019

*To my mother, Elvira. I owe you everything I am and achieve.*

# Abstract

The conceptual frontier that for many years separated the real-time embedded systems domain from the high-performance computing domain has been shattered by recent technological advances and market trends. Nowadays, many contemporary applications have started to cross the boundaries between the two domains: they are subject to stringent timing requirements and have huge computation demands, which cannot be successfully satisfied following the sequential execution paradigm. Intra-task parallelism enables an application to run simultaneously on different cores, thus allowing for increased performance and offering opportunities to make efficient use of the emergent many-core embedded architectures. However, parallelization adds another dimension to the already challenging problem of multiprocessor real-time scheduling.

In this dissertation, we are interested in studying the problem of scheduling a set of hard real-time parallel tasks atop multiprocessor systems, where each parallel task is represented as a Directed Acyclic Graph (DAG). The DAG task model reflects general features of parallelism characteristic of widely used parallel programming models (such as OpenMP). In this model, a task is defined as a set of concurrent subtasks whose execution has to obey to a set of precedence constraints. Subtasks that are independent of each other may execute either in parallel or sequentially, depending on the decisions of the real-time scheduler. We address both the global and partitioned scheduling paradigms, under traditional preemptive scheduling algorithms, while exploiting the internal structure of the DAGs. Furthermore, we also investigate the applicability of the DAG model to the schedulability analysis of parallel applications with conditional control-flow constructs.

First, we address the schedulability of a set of DAG tasks according to a global fixed-priority scheduling algorithm. To this end, we present a response time analysis based on the concept of problem window, a technique that has been extensively used to study the schedulability of sequential task in multiprocessor systems. Such problem window approach usually categorizes interfering jobs in three different groups: carry-in, carry-out and body jobs. By taking into account the precedence constraints between subtasks pertaining to the same DAG, we propose two novel techniques to derive more accurate upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks. Using this new characterization, we then derive a schedulability analysis to compute an improved upper-bound on the worst-case response time of each parallel task in both the constrained and arbitrary deadline case.

Second, we look at the problem of scheduling a set of partitioned DAG tasks with constrained deadlines from two different perspectives. Here, partitioning means statically assigning each subtask to a specific core, yet allowing multiple subtasks of the same DAG task to run on different cores. In the first approach, we develop a response time analysis for fixed-priority scheduling, which can be used under any given subtask-to-core mapping. To solve the issue that arises from the potential cross-core dependencies, we show that a DAG task can be modeled and analyzed as a

set of self-suspending tasks and present an algorithm to break the cyclic computations. In order to increase the effectiveness of this schedulability test, we also propose a response time analysis for sporadic self-suspending tasks running on a uniprocessor system. In the other approach, we propose a schedulability analysis based on a workload duplication technique. Specifically, we present a partitioning algorithm that maps similar paths of a DAG to the same cores, aiming to minimize the number of cores required for task feasibility and to eliminate cross-core dependencies. Thanks to the duplication of key subtasks, all resulting partitions become independent of each other. Thus, the problem of scheduling a set of partitioned DAGs is reduced to the problem of scheduling a set of sequential tasks on multiprocessors in a partitioned manner.

Third, we deal with the problem of modeling and scheduling a set of DAG tasks with conditional execution. Although it is expected that industrial applications feature conditional operations that depend on run-time data, the DAG model assumes that each and every instance of the DAG releases and executes all its subtasks. While conditional statements were not a major concern in the case of sequential tasks, we are the first to show that they are detrimental for the schedulability analysis of parallel tasks and thus should be explicitly modeled. Consequently, we generalize the DAG model by proposing a multi-DAG model where each conditional parallel task is characterized by a set of execution flows, each of which represented as a separate DAG. Due to conditional statements, only one of such execution flows is undertaken at each task activation. Then, we develop a two-step algorithm that constructs a single DAG of servers (servers are the scheduling entities) to handle the execution of a multi-DAG task, and prove that these servers are able to safely and fully execute any of its execution flows. As a result, each multi-DAG task can be modeled by its single DAG of servers, which facilitates in leveraging the existing single-DAG schedulability analysis techniques for analyzing the schedulability of conditional DAG tasks.

Experimental results validate the contributions proposed in this dissertation and show that they may lead to a substantial reduction in the pessimism of the schedulability analyses for DAG tasks comparatively to state-of-the-art methods. Such improvements are essential for exploiting multiprocessors in real-time embedded systems in a more efficient way.

# Resumo

A fronteira conceptual que, por muitos anos, separou o domínio dos sistemas embebidos e de tempo-real do domínio da computação de alto desempenho tem vindo a desmoronar com os recentes avanços tecnológicos e tendências de mercado. Nos dias que correm, muitas das aplicações contemporâneas já começam a evidenciar características associadas aos dois domínios: estas aplicações estão sujeitas a requisitos temporais muito rigorosos e exigem elevado desempenho computacional, condições que não podem ser satisfeitas recorrendo apenas ao paradigma de execução sequencial. O modelo de computação paralela habilita a execução simultânea de uma aplicação em vários processadores, o que possibilita um aumento considerável do seu desempenho e um uso mais eficiente do extremo poder computacional oferecido pelas novas arquitecturas *many-core*. No entanto, o paralelismo vem tornar o problema de escalonamento de tempo-real ainda mais desafiante.

Nesta dissertação estamos interessados em estudar o problema de escalonar um conjunto de tarefas paralelas e de tempo-real num sistema multi-processador, no qual cada tarefa paralela é representada por um grafo acíclico e direccionado (*DAG*). O modelo de tarefas DAG espelha tipos gerais de paralelismo que são característicos dos modelos de programação paralela mais conceituados (como é o caso do OpenMP). Neste modelo, uma tarefa é definida como um conjunto de sub-tarefas concorrentes cuja ordem de execução tem de respeitar um conjunto de precedências. Sub-tarefas que são independentes umas das outras podem executar tanto em paralelo como sequencialmente, tudo depende das decisões tomadas pelo escalonador de tempo-real. Neste sentido, consideramos os paradigmas de escalonamento global e particionado, recorrendo a algoritmos de escalonamento tradicionais, enquanto tiramos partido do conhecimento da estrutura interna dos DAGs. Além disso, também investigamos o grau de aplicabilidade do modelo DAG na análise de escalonabilidade de tarefas paralelas que contêm estruturas condicionais de controlo.

O primeiro problema considerado prende-se com o escalonamento de um conjunto de tarefas DAG de acordo com um algoritmo global de prioridades fixas. Para este efeito, é apresentada uma análise de tempo de resposta baseada no conceito conhecido como "*problem window*", uma técnica que tem sido bastante empregue para estudar a escalonabilidade de tarefas sequenciais em sistemas multi-processador. Tal abordagem normalmente classifica as diferentes instâncias das tarefas interferentes em três grupos distintos: *carry-in*, *carry-out* and *body*. Através da exploração das precedências entre sub-tarefas do mesmo DAG, são propostas duas técnicas inovadoras para estimar com mais precisão a carga máxima de trabalho (e consequentemente a interferência) gerada pelas instâncias *carry-in* and *carry-out* das tarefas com mais prioridade. Esta nova caracterização permite-nos refinar a análise de escalonabilidade tradicional, o que resulta numa redução do tempo de resposta máximo de cada tarefa paralela no pior cenário, tanto no caso em que os prazos não podem exceder os períodos de activação *constrained deadline*, como no caso geral de prazos arbitrários *arbitrary deadline*.

Em segundo lugar, investigamos o problema de escalonar um conjunto de tarefas DAG particionadas com *constrained deadline* através de duas abordagens bastante diferentes. Neste contexto, particionar significa atribuir estaticamente cada sub-tarefa a um processador específico, mas permitindo que várias sub-tarefas do mesmo DAG executem em processadores diferentes. Na primeira abordagem, é desenvolvida uma análise de tempo de resposta para escalonamento por prioridades fixas, que pode ser utilizada independentemente das particularidades do mapeamento um-para-um fornecido. De forma a capturar as dependências transversais a vários processadores devido ao particionamento, é demonstrado que uma tarefa DAG pode ser modelada e analisada como um conjunto de tarefas que se auto-suspendem (*self-suspending tasks*). Para este efeito, é proposto um algoritmo que identifica o pior cenário de escalonamento, estabelece uma ordem para que as computações sejam fidedignas, e compõe recursivamente as tarefas que se auto-suspendem. Uma vez que detectamos um erro crítico na literatura relacionada, e para aumentar a qualidade desta solução, também apresentamos uma análise de tempo de resposta para tarefas esporádicas que se auto-suspendem em sistemas uni-processador. Relativamente à segunda abordagem, é proposta uma análise de escalonabilidade baseada numa técnica de duplicação da carga de trabalho (neste caso os nós dos grafos) que tem usufruído de grande sucesso no domínio da computação de alto desempenho. Em particular, é apresentado um algoritmo de particionamento que mapeia caminhos semelhantes de um DAG para o mesmo processador, com o objectivo de minimizar o número de processadores necessários para garantir o cumprimento os requisitos temporais da tarefa e eliminar as dependências transversais a múltiplos processadores. Graças à duplicação de certas sub-tarefas, todas as partições resultantes tornam-se independentes umas das outras. Deste modo, o problema de escalonar um conjunto de tarefas DAG é reduzido ao problema de escalonar um conjunto de tarefas sequenciais num sistema multi-processador de acordo com o paradigma particionado.

Por último, enfrentamos o problema de modelar e escalonar um conjunto de tarefas DAG que exibem execução condicional. Embora seja expectável que as aplicações industriais apresentem operações condicionais que dependem do estado de certas variáveis em tempo de execução, o modelo DAG assume que cada uma das instâncias da tarefa paralela acciona e executa todas as suas sub-tarefas. Enquanto a questão das expressões condicionais nunca foi muito relevante no caso de tarefas sequenciais, esta dissertação expõe que as expressões condicionais são fundamentais para a análise de escalonabilidade de tarefas paralelas e, como tal, devem ser modeladas explicitamente. Consequentemente, generalizamos o modelo DAG ao propor um modelo multi-DAG onde cada tarefa paralela e condicional é caracterizada por uma colecção de fluxos de execução, cada um dos quais é representado por um DAG. Devido às estruturas de controlo, apenas um único fluxo de execução é prosseguido em cada activação da tarefa. Em seguida, é apresentado um algoritmo que constrói um DAG de servidores (servidores são as entidades de escalonamento) para manipular a execução de uma tarefa multi-DAG, juntamente com a demonstração que tal estrutura de servidores é capaz de garantir o correcto e completo processamento de qualquer fluxo de execução. Como resultado, cada tarefa multi-DAG pode ser representada pelo DAG de servidores correspondente, o que faz com que os métodos de análise de escalonabilidade existentes para DAGs singulares possam ser aproveitados para verificar a escalonabilidade de tarefas paralelas e condicionais sem incorrer num grau de pessimismo proibitivo.

Os resultados experimentais validam as contribuições propostas nesta dissertação, e mostram que as mesmas permitem reduzir substancialmente o pessimismo das análises de escalonabilidade para DAGs, em comparação com o estado da arte. Estas melhorias são fundamentais para aumentar os índices de eficiência no uso das arquitecturas multi-processador em sistemas embebidos e de tempo real.

# Acknowledgments

This PhD was far from smooth. In fact, it was a roller coaster of challenges and emotions. But isn't that what it takes to have a great ride? Now I realize. My adventure would not be truly life-changing if it wasn't for the wonderful folks that joined along. To all of you I express my wholehearted gratitude. In particular, I would like to dedicate the following few but sincere words to the people who I am indebted to for helping me in manifold large and small ways over the last six years.

First, I will always be immensely grateful to my supervisor Vincent Nélis, my co-supervisor Luís Miguel Pinho and my "senpai" Geoffrey Nelissen for taking me under their wing and not giving up on me when I have gone astray. Many thanks to Vincent for his dedication, expert advices, continuous encouragement, and for teaching me proper research methodology. Many thanks to Miguel for his precious guidance, unwavering support, valuable feedback, and for always finding time to meet me despite his busy schedule. Many thanks to Geoffrey for his sharp insights, motivation, interesting and detailed discussions, comprehensive assistance, incredible availability, and for pushing my scientific maturity to a new level — It is a pleasure to work with you. These achievements would have not been possible without your decisive contribution.

Second, my perpetual heartfelt gratitude and appreciation goes to the people that mean the world to me and ask nothing in return: my family. My parents, Elvira and Manuel, and my sister, Helena, have been a constant source of love, unquestionable devotion, selfless concern, motivation, patience, understanding and bliss. My dearest friends Marcos, Nuno, Marta, Pacheco, Leal and José Alberto bring an endless stream of joy and excitement to my life and know better how to reset my "malfunctioning chip". Yes, you guys are family too!

Third, my journey was way more genuine, amusing and enjoyable thanks to the friendship of Cláudio, David and André. The office is a comfort zone to me when you guys are around, together we rage and celebrate, and our countless coffee and smoke breaks always make my day. Among the many great colleagues I feel fortunate to have met at CISTER, I would like to extend a special thanks to Guru, Boris, Kos, João, Garibay and Konstantinos for our scientific endeavors, memorable trips and/or funny social sessions. Also, a huge and heartfelt thanks goes to Luís Nogueira for opening CISTER's doors to me in the first place and then for inviting me to pursue this PhD.

Finally, I would like to thank the body of directors of CISTER Research Center for creating an outstanding research environment and their financial support. A special thanks is also due to every staff member at CISTER, who has been extremely friendly and helpful all these years.

José Fonseca

# Publications

This PhD thesis has resulted in the following scientific publications:

José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi and Luís Miguel Pinho. *How to deal with control-flow information in parallel real-time applications?*. In the 5th Real-Time Scheduling Open Problems Seminar. Madrid, Spain, 2014.

Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, José Fonseca, Marko Bertogna, Eduardo Quiñones, Roberto Vargas and Andrea Marongiu. *The challenge of time-predictability in modern many-core architectures*. In the 14th International Workshop on Worst-Case Execution Time Analysis. Madrid, Spain, 2014.

José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi and Luís Miguel Pinho. *A Multi-DAG Model for Real-Time Parallel Application with Conditional Execution*. In the 30th ACM/SIGAPP Symposium On Applied Computing. Salamanca, Spain, 2015.

Geoffrey Nelissen, José Fonseca, Gurulingesh Raravi and Vincent Nélis. *Timing Analysis of Fixed-Priority Self-Suspending Sporadic Tasks*. In the 27th Euromicro Conference on Real-Time Systems. Lund, Sweden, 2015.

José Carlos Fonseca, Vincent Nélis, Geoffrey Nelissen and Luís Miguel Pinho. *Analysis of self-interference within DAG tasks*. In the 6th Real-Time Scheduling Open Problems Seminar. Lund, Sweden, 2015.

José Fonseca, Geoffrey Nelissen, Vincent Nélis and Luís Miguel Pinho. *Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling*. In the 11th IEEE International Symposium on Industrial Embedded Systems. Krakow, Poland, 2016.

José Fonseca, Geoffrey Nelissen and Vincent Nélis. *Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling*. In the 25th International Conference on Real-Time Networks and Systems. Grenoble, France, 2017. **Best paper award.**

José Fonseca, Geoffrey Nelissen and Vincent Nélis. *Schedulability Analysis of DAG Tasks with Arbitrary Deadlines under Global Fixed-Priority Scheduling*. Real-Time Systems, Springer. 2019. **Invited paper (to appear).**

# Contents

# List of Figures

# List of Abbreviations

| | |
|---|---|
| BCRT | Best-Case Response Time |
| DAG | Directed Acyclic Graph |
| DM | Deadline Monotonic |
| EDF | Earliest Deadline First |
| FP | Fixed-Priority |
| GDM | Global Deadline Monotonic |
| GEDF | Global Earliest Deadline First |
| GFP | Global Fixed-Priority |
| GSSG | Global Synchronous Server Graph |
| HPC | High-Performance Computing |
| JLDP | Job-level Dynamic-Priority |
| JLFP | Job-level Fixed-Priority |
| MILP | Mixed-Integer Linear Programming |
| RM | Rate Monotonic |
| PDM | Partitioned Deadline Monotonic |
| PEDF | Partitioned Earliest Deadline First |
| PFP | Partitioned Fixed-Priority |
| RTES | Real-Time Embedded Systems |
| RTA | Response Time Analysis |
| SSG | Synchronous Server Graph |
| WCET | Worst-Case Execution Time |
| WCRT | Worst-Case Response Time |

# Chapter 1

# Introduction

## 1.1 Real-time systems

A real-time system is any information processing system where the correctness of each computation depends not only on its logical result but also on the time instant at which such result is produced (Stankovic, 1988). Failure to respond within a specified time interval is considered as a faulty response. Thus, the key property of a real-time system is *time predictability*. Real-time applications are nowadays indispensable in our daily lives spanning areas such as transportation, telecommunications, healthcare, industrial automation, multimedia, security, energy and financial services.

While a multimedia application can tolerate some timing delays without major consequences, it is fundamental that a control application strictly respects its timing constraints. Based on the different levels of criticality, real-time systems are broadly classified into two groups: *hard* and *soft* real-time systems. In a hard real-time system, catastrophic events (e.g., loss of human life) may occur if the expected functionality is delivered after the pre-defined temporal requirement. On the other hand, for soft real-time systems, late results still retain utility despite leading to performance degradation, as long as the delay is within an acceptable range. In this research work, we focus exclusively on the former.

Real-time systems are reactive and recurrent in nature. For this reason, real-time applications are typically modeled as finite collections of processes called "tasks" that are repeated according to different rates dictated by the environment.

### 1.1.1 Real-time task model

In the traditional real-time task model (Liu and Layland, 1973), a real-time application is composed by a set of $n$ independent sequential tasks $\tau = \{\tau_1, \ldots, \tau_n\}$. Each task $\tau_i$ releases a sequence of (potentially) infinite identical instances called "jobs" to be executed on the target platform. Each task $\tau_i$ is characterized by a 3-tuple $(C_i, T_i, D_i)$ with the following interpretation:

- The **worst-case execution time (WCET)** $C_i$ denotes an upper-bound on the execution time required by any job of $\tau_i$ to complete its computation without interruption. Hence, the actual execution time of a job must not exceed its WCET under any scenario. The WCET estimation is critical for the reliability of real-time systems. Although the exact maximum value is in general impossible to derive, one must guarantee that the computed WCET is both *safe* (i.e., the upper-bound is not underestimated) and *tight* (i.e., marginally pessimistic).

- The **period or minimum inter-arrival time** $T_i$ represents the frequency at which new jobs of $\tau_i$ are released in the system. Jobs of a *periodic* task are released exactly every $T_i$ time units, whereas two consecutive jobs generated by a *sporadic* task must be separated by at least $T_i$ time units.

- The **relative deadline** $D_i$ determines that every job of $\tau_i$ has to complete its execution within $D_i$ time units from its release. While in a hard real-time system all jobs of each task must meet their deadlines, deadline misses are tolerated in a soft-real-time system if the execution tardiness of the jobs is not unbounded.

According to the relation between the deadline and the period of each task, a task set $\tau$ is said to have (i) *implicit deadlines* if $D_i = T_i$, (ii) *constrained deadlines* if $D_i \leq T_i$, or (iii) *arbitrary deadlines* if $D_i > T_i$ is allowed (i.e., there is no constraint between the relative deadline and the period), $\forall \tau_i \in \tau$. For both implicit and constrained deadline tasks, at most one job of each task is active at any time instant $t$. In contrast, multiple active jobs of an arbitrary deadline task may overlap in a given time interval. In this dissertation, we consider primarily sporadic tasks with constrained deadlines, although the general case of arbitrary deadlines is covered in Section 3.7.

The $k^{th}$ job of task $\tau_i$ denoted by $J_{i,k}$ becomes ready for execution at arrival time $a_{i,k}$, has an absolute deadline $d_{i,k} = a_{i,k} + D_i$, and remains active until finishing time $f_{i,k}$. The difference between the finishing time and the arrival time corresponds to the response time of the job. Formally, $r_{i,k} = f_{i,k} - a_{i,k}$. Additionally, the response time $R_i$ of $\tau_i$ is given by the maximum response time of all its jobs: $R_i = \max_{\forall J_{i,k} \in \tau_i} (r_{i,k})$.

The fraction of the processor's capacity required by any job of task $\tau_i$ is defined as its utilization $U_i = \frac{C_i}{T_i}$. Similarly, the utilization $U_\tau$ of a task set $\tau$ is given by the sum of the utilization of its tasks, i.e. $U_\tau = \sum_{i=1}^{n} U_i$. On a platform comprised of $m \geq 1$ processors, the following inequalities constitute necessary feasibility conditions:

$$U_i \leq 1, \forall \tau_i \in \tau \tag{1.1}$$

$$U_\tau \leq m \tag{1.2}$$

### 1.1.2 Schedulability analysis

Let $\mathcal{A}$ be a scheduling algorithm. A task set $\tau$ is $\mathcal{A}$-schedulable if all jobs of its tasks meet their deadlines when scheduled by $\mathcal{A}$. That is, $R_i \leq D_i$ for each $\tau_i \in \tau$. A task set $\tau$ is said to be feasible if it exists at least one scheduling algorithm under which $\tau$ is schedulable. Algorithm $\mathcal{A}$ is optimal if all feasible task sets are $\mathcal{A}$-schedulable.

The process to determine at design time whether a task set $\tau$ is schedulable under a scheduling algorithm $\mathcal{A}$ and on a given platform is called *schedulability test*. Schedulability tests can be sufficient, necessary or exact. A *sufficient test* guarantees that if the condition is true, then $\tau$ is $\mathcal{A}$-schedulable but the opposite cannot be concluded. On the other hand, a *necessary test* cannot deem $\tau$ $\mathcal{A}$-schedulable in any case, but if the test fails then $\tau$ is definitely not schedulable by $\mathcal{A}$. An *exact test* is both sufficient and necessary, since it always precisely classifies $\tau$ as either schedulable or not.

A counter-intuitive effect on the schedulability of a task set provoked by a positive change in the task parameters is referred to as a "scheduling anomaly". A scheduling algorithm or schedulability test is said to be *sustainable* if no scheduling anomalies are found. In other words, a scheduling algorithm $\mathcal{A}$ is sustainable if a task set $\tau$ deemed $\mathcal{A}$-schedulable does not become unschedulable under $\mathcal{A}$ when $\tau$ is relaxed by, for example, reducing a WCET $C_i$, enlarging a period $T_i$ or increasing a deadline $D_i$. (The interested reader is referred to (Cerqueira et al., 2018) for a formal classification). As in practice the execution time of jobs will vary significantly, this property is of paramount importance to assure the timeliness of the system.

### 1.1.3 Real-time schedulers

A scheduler is the entity responsible for defining the execution order of tasks on the processors. Whenever there are more pending jobs than the number of cores available $m$, the scheduler has to decide which $m$ pending jobs to allocate to the cores. The main goal of a real-time scheduler is to take scheduling decisions that allow every task in the system to respect their timing constraints. Real-time schedulers are commonly divided in two categories:

- **Static schedulers** take scheduling decisions based on a scheduling table pre-computed at design time and which stores the task set specific allocation scheme. Constructing a scheduling table requires full knowledge of the tasks activation pattern. For this reason, static schedulers tend to be too inflexible for sporadic tasks.

- **Dynamic schedulers** follow scheduling rules defined by a scheduling algorithm to make scheduling decisions during run-time according to the current system state (e.g., task parameters or remaining workload). Therefore, dynamic schedulers do not rely on pre-computed information. We restrict our attention to a particular class of these schedulers called *priority-driven*, where a scheduling algorithm defines the priority assignment of jobs and takes scheduling decisions that favor the higher priority workload.

Priority-driven scheduling algorithms can be further categorized according to how the job priorities may vary over time. Here we distinguish three major classes.

- **Fixed-priority (FP)**: each task is assigned with a constant priority and all of its jobs inherit such fixed-priority. Thus, the scheduling decisions are not affected by the elapsed time. Rate Monotonic (RM) and Deadline Monotonic (DM) are representative examples of these algorithms.

- **Job-level fixed-priority (JLFP)**: although priorities are allowed to change over time, each distinct job is assigned with a fixed-priority when it arrives in the system. That is, different jobs of a same task may have different priorities but once assigned the priority of a job never changes. Earliest Deadline First (EDF) is the most notable JLFP scheduling algorithm.

- **Job-level dynamic-priority (JLDP)**: the priority of a job may change during its execution as a function of time. An example of this most general class of the schedulers is the Least Laxity First algorithm.

A scheduling algorithm is said to be *work-conserving* if no processor stays idle as long as there are pending jobs waiting to be executed. Furthermore, scheduling algorithms can be differentiated based on their capability to alter the job-to-processor allocation when the processor is busy. In *preemptive scheduling*, the execution of a running job is interrupted (preempted) whenever higher priority jobs are released. Its execution is resumed only after all the active higher priority jobs terminate or voluntarily yield the processor. In contrast, in *non-preemptive scheduling* jobs once scheduled always run continuously to completion as they cannot be interrupted. Since scheduling decisions are taken after a job terminates, the execution of a higher priority job may be delayed by at most one lower priority job. This effect is known as *blocking*. In this thesis, we mainly consider fixed-priority preemptive scheduling but EDF preemptive scheduling is also addressed in Section 4.6.

### 1.1.4 Uniprocessor real-time theory

The seminal research into uniprocessor real-time scheduling dates back to the late 1960s and early 1970s, and it was primarily applied to schedule computer programs during the first manned space flight to the moon (Liu, 1969; Liu and Layland, 1973). Remarkably, Liu and Layland (Liu and Layland, 1973) introduced provably optimal (class wise) FP and JLFP algorithms and respective tests for the scheduling of periodic tasks. During the 1980s and 1990s, these policies were studied under more general models, including sporadic arrivals (Mok, 1983), synchronization (Sha et al., 1990) and overheads (Katcher et al., 1993). Today, this theory is considered mature and successfully put in practice for industrial purposes.

Under FP scheduling, tasks are commonly indexed in order of decreasing priority, i.e., if $i < j$ then the priority of $\tau_i$ is higher than that of $\tau_j$. Ties are broken arbitrarily. For example, the RM algorithm introduced in (Liu and Layland, 1973) prioritizes tasks with shorter periods such that if

$T_i < T_j$ then $i < j$, while DM (Leung and Whitehead, 1982) prioritizes tasks with shorter relative deadlines.

For implicit deadline periodic tasks, Liu and Layland showed that RM is optimal among the FP schedulers (Liu and Layland, 1973) and derived the following sufficient schedulability test which also applies to sporadic tasks. A task set $\tau$ comprised of $n$ implicit deadline sporadic task is schedulable under RM on a uniprocessor system if $U_\tau \leq n(2^{\frac{1}{n}} - 1)$. When $n \to \infty$, the utilization bound $n(2^{\frac{1}{n}} - 1)$ converges to $\ln(2) \approx 0.69$. This is the highest-achievable utilization bound for any FP algorithm, since there are implicit deadline task sets with an utilization slightly larger than the bound that no FP algorithm can schedule.

However, RM is not optimal for uniprocessor systems because every implicit deadline tasks set where $U_\tau \leq 1$ is feasible on a uniprocessor. Similarly, RM is not optimal when tasks may have deadlines different than their periods. Instead, DM priority assignment, where $i < j$ if $D_i < D_j$, is optimal for constrained deadline tasks (Leung and Whitehead, 1982). Nevertheless, the RM schedulability test also holds for DM by simply replacing the term $U_\tau$ with $\sigma_\tau = \sum_{\forall \tau_i \in \tau} \frac{C_i}{\min(D_i, T_i)}$, where $\sigma_\tau$ denotes the density of the task set.

Although the above schedulability test is very efficient, it fails to identify those implicit deadline task sets whose utilization exceeds the bound but are still RM-schedulable. To this end, Joseph and Pandya (Joseph and Pandya, 1986) proposed a response time analysis (RTA) to compute the exact worst-case response time (WCRT) of each sporadic task explicitly. The key concept behind the analysis is the existence of a *critical instant*, which defines a release pattern for all the tasks so that the response time of the first job released by the task under analysis is maximized. Their approach considers constrained deadline task sets under any FP priority assignment. Given such WCRTs, $R_i \leq D_i$ for each $\tau_i \in \tau$ provides a straightforward schedulability test. The response time of a task $\tau_i$ is given by the smallest value that satisfies the following recursion and $R_i \geq C_i$:

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j \tag{1.3}$$

Starting with $C_i$ as an initial value for $R_i$, $R_i$ is computed by iteratively evaluating the right-hand side until the equation converges. Convergence is guaranteed by the necessary feasibility condition $U_\tau \leq 1$. Response time analyses have been proven sustainable in (Baruah and Burns, 2006) and are therefore invaluable to determine schedulability in real-time systems. Response time analyses for arbitrary deadline task sets are also available in the literature (Tindell et al., 1994), but require a longer time interval to be checked as multiple jobs of a same task may be active simultaneously. Such interval is known as *busy-period*.

Despite FP scheduling being widely used in practical systems, the JLFP policy EDF theoretically dominates any other preemptive algorithm for uniprocessor platforms. EDF is very intuitive, since it schedules in order of urgency. That is, EDF prioritizes jobs according to their absolute deadlines so that the job with the nearest deadline is always the one executing.

A very simple example may clarify how EDF works. Let us consider the task set detailed in Table 1.1, which has four tasks and utilization: $U_\tau = \frac{2}{13} + \frac{3}{13} + \frac{2}{15} + \frac{3}{17} = 72.2\%$. Fig. 1.1 shows

the timeline execution for the first job of each task. Parameter $O_i$ denotes the offset of $\tau_i$, i.e., the release time of the first job of $\tau_i$ w.r.t. the reference time. The only task released at instant 0 is $\tau_4$, so it starts executing immediately. At instant 1, $\tau_3$ arrives with an earlier deadline. Since $\tau_4$ needs more 2 times units to finish its instance, it is preempted by $\tau_3$. It goes like this until instant 6, when $\tau_1$ finishes his job. Now that the remaining three tasks are ready, the earliest deadline task is selected for execution: $\tau_3$. The schedule goes on this descending way until instant 10 when the last first job terminates.

Table 1.1: A task set example for EDF schedule.

| **Task** | $C_i$ | $T_i$ | $D_i$ | $O_i$ |
|---|---|---|---|---|
| $\tau_1$ | 2 | 11 | 11 | 3 |
| $\tau_2$ | 3 | 13 | 13 | 2 |
| $\tau_3$ | 2 | 15 | 15 | 1 |
| $\tau_4$ | 3 | 17 | 17 | 0 |

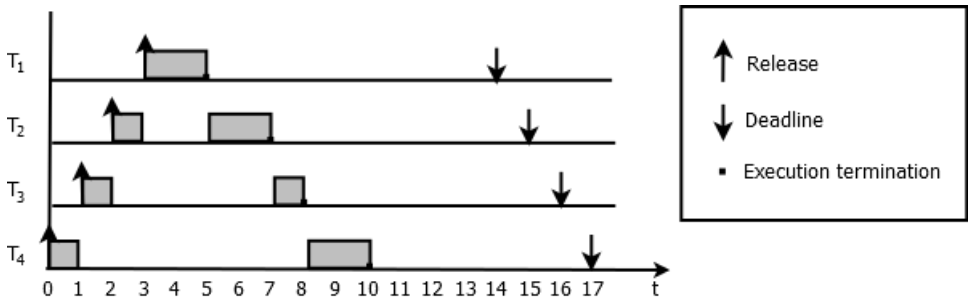



Figure 1.1: An EDF schedule example.

The theoretically superiority of EDF comes from the fact that if there exists a feasible schedule for a sporadic task set, then the task set is EDF-schedulable irrespective of the deadline constraints (Dertouzos, 1974). Thus, EDF is optimal among all preemptive scheduling algorithms on uniprocessors. In the case of implicit deadline tasks, the utilization bound $U_\tau \leq 1$ constitutes a simple exact schedulability condition (Liu and Layland, 1973). However, when generalized to arbitrary deadline tasks, the density bound $\sigma_\tau \leq 1$ is no longer an exact schedulability test. In fact, this test is considerably pessimistic if some of the tasks have constrained deadlines. Exact tests have been proposed to close such gap at the cost of increased computational complexity (more precisely, they run in pseudo-polynomial time) (Albers and Slomka, 2005). The interested reader is referred to (Sha et al., 2004) for an excellent survey in the field of uniprocessor scheduling.

### 1.1.5 Multiprocessor real-time theory

Unfortunately, multiprocessor real-time scheduling has not yet enjoyed the success as its uniprocessor counterpart. As early as in 1969, Liu (Liu, 1969) observed the intrinsic complexity of multiprocessor scheduling and how hardly uniprocessor algorithms could be extended to it:

> *"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."*

In a multiprocessor system, the scheduling problem consists not only on deciding which $m$ jobs to execute at any given point in time, but also on how to distribute (map) jobs among the processors. Hence, multiprocessor scheduling encompasses a priority assignment problem and an allocation problem. Regarding the allocation and consequent migration behavior of the jobs, multiprocessor scheduling can be distinguished in three major categories (Carpenter et al., 2004):

- In **global** scheduling, a job is allowed to execute on any processor at any point in time and therefore can migrate between processors during its activity. Broadly speaking, the system has one scheduler which dispatches jobs to the processors according to a single shared ready queue, where the priority ordering is maintained. While global scheduling shall overcome the algorithmic capacity loss intrinsic to partitioned approaches, it has practical downsides related to costly overheads due to preemption and migration operations.

- In **partitioned** scheduling, tasks are statically assigned to processors in a fixed manner so that all jobs of the same task must execute in the allocated processor. Thus, migrations are not supported. In this case, each processor has its own independent scheduler. Consequently, the multiprocessor scheduling problem is reduced to a set of uniprocessor scheduling problems. This problem reduction implies that the task set must first be partitioned such that every task is assigned to a processor and no processor is overloaded. To find valid task assignments, bin-packing heuristics (such as First-Fit, Best-Fit and Worst-Fit) are usually employed since the allocation problem is NP-hard.

- In **semi-partitioned** scheduling, some tasks are pinned to specific processors while other tasks are allowed to migrate across processors. Semi-partitioning can be seen as a compromise between partitioned and global scheduling that aims at combining the best of both approaches.

Table 1.2: A task set example causing the Dhall effect.

| **Task** | $C_i$ | $T_i$ | $U_i$ |
|---|---|---|---|
| $\tau_1$ | $2\varepsilon$ | 1 | $\to 0$ |
| $\tau_2$ | $2\varepsilon$ | 1 | $\to 0$ |
| $\tau_3$ | 1 | $1+\varepsilon$ | $\to 1$ |

Highlighting the complexity of multiprocessor scheduling, Dhall (Dhall, 1977) reported that when globally enforcing the typical scheduling policies (e.g., RM or EDF) on a multiprocessor host, some task sets may miss deadlines even though very low system utilization is requested. To

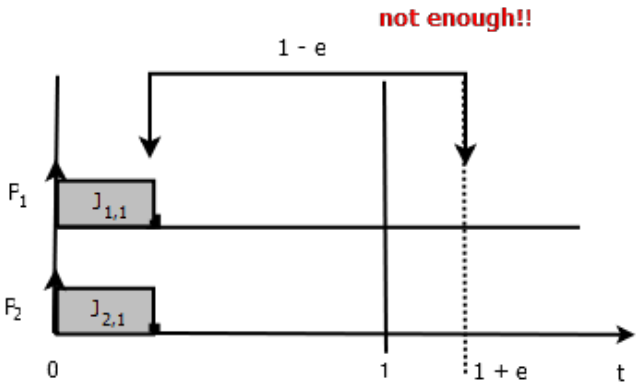



Figure 1.2: A Dhall effect schedule example.

provide an understanding of the so-called *Dhall effect*, let us consider an example. Consider a system with 2 processors ($m = 2$) and 3 implicit deadline tasks ($n = 3$), as specified by Table 1.2, to be scheduled according to the EDF policy. Since all tasks are released at $t = 0$, the first job of $\tau_1$ and $\tau_2$ with deadline equal to 1 will have higher priority over the first job of $\tau_3$, whose deadline is $1 + \varepsilon$. Consequently, processors $P_1$ and $P_2$ are assigned to $J_{1,1}$ and $J_{2,1}$ during the time interval $[0, 2\varepsilon]$, leaving a maximum of 1 - $\varepsilon$ time units for $J_{3,1}$ before its deadline, which is not enough for it to be completely executed (see Fig. 1.2). Hence, this task set is not schedulable under EDF on a 2-processor computing system although $\sum_{i=1}^{n} U_i \to 1$ as $\varepsilon \to 0$.

This finding discouraged the research community to study global scheduling algorithms for two decades, until Phillips et al. (Phillips et al., 1997) showed that the Dhall effect is mostly related to tasks with high utilization, and not an inherent global scheduling issue. Nevertheless, even in the case of implicit deadline tasks, the utilization bound of any global FP (GFP) scheduler is upper-bounded by $\frac{m+1}{2}$, which matches the best results on partitioned scheduling. For a given priority assignment and a specific task set, global worst-case response time bounds can be computed using multiprocessor response time analysis (see (Davis and Burns, 2011) for a comprehensive survey). In particular, Bertogna and Cirinei (Bertogna and Cirinei, 2007) derived a response time analysis for constrained deadline sporadic tasks under global EDF (GEDF) based on a *problem window* introduced by Baker (Baker, 2003) to estimate the maximum interfering workload that can be generated by each higher priority task.

By taking into consideration the maximum utilization $U_{max}$ among tasks within a task set, Goossens et al. (Goossens et al., 2003) showed that a set of independent periodic tasks with implicit deadlines can be successfully schedulable by GEDF on $m$ processors if

$$U_\tau \leq m - (m-1)U_{max}. \tag{1.4}$$

Later, it was proven that this bound also holds for sporadic tasks with arbitrary deadlines by factoring in density instead of utilization (Bertogna et al., 2005; Baker and Baruah, 2007). Although no optimal schedulers exist for both constrained and arbitrary task sets (Fisher et al., 2010), optimality as been achieved by more complex global JLDP algorithms (for example, PD$^2$ (Srini-

vasan and Anderson, 2006)) when all tasks have implicit deadlines.

The aforementioned results were derived for *homogeneous* (also known as identical) multiprocessors, where every processor in the system has the same computing capabilities and the execution rate of the tasks is equal on each one of them. The type of multiprocessor systems can be further classified as *uniform* or *heterogeneous*. Although in both types different processors may have different speed, such speed determines the execution rate of a task in the former, while in the latter the execution rate also depends on the characteristics of the task itself. In this research work, we restrict our attention to a multiprocessor platform composed of $m$ homogeneous cores. This abstraction comes from the fact that schedulability analyses are assumed to be independent from the low-level timing analyses which account for the architecture-specific details. Furthermore, we use both the term processor and core as a synonym for a single processing unit in the same host platform.

## 1.2 Research motivation

More than forty years ago, in 1973, Chang Liu and James Layland proposed in (Liu and Layland, 1973) a simplistic model to characterize the timing behavior of time-critical control and monitor functions that they termed "pure process control". Their model abstracts each of these pure process controls by two numbers: a worst-case execution requirement and an execution rate (called period). Since then the real-time community has developed an extensive set of task models, platform models and scheduling techniques by relaxing some of the assumptions originally made by Liu and Layland to incorporate the requirements of new systems, applications and architectures (Davis and Burns, 2011; Sha et al., 2004). Although researchers have built an impressive body of knowledge that gives a deep understanding of the broad variety of scheduling problems, today's applications in the real-time embedded systems (RTES) domain are neither designed nor implemented the way it used to be back in 1973, and most of the models available today are not expressive enough to correctly model the timing behavior of most of the contemporary embedded applications.

In fact, the conceptual frontier that segregated high-performance computing (HPC) and RTES domains for many years is becoming thinner every day. In a nutshell, HPC designers have been challenged to speed up applications with continuous input streams that involved massive computational and/or memory-intensive operations. On the other hand, RTES designers have been challenged to guarantee at design time that recurrent applications with specialized functionalities meet their timing requirements under worst-case scenarios. Nowadays, HPC systems face an increasing demand for energy-efficiency and bounded response times, whereas RTES strive for extra flexibility and computational performance. Unsurprisingly, many modern applications (e.g., intelligent transportation systems and autonomous driving) have started to cross the boundaries between the two domains: they have huge computational demands and are subject to strong timing constraints, which cannot be answered by the sequential execution paradigm.

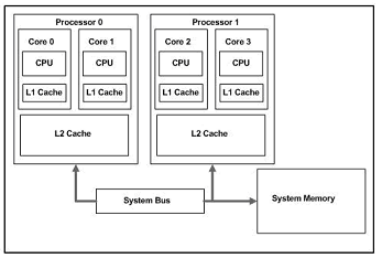



Figure 1.3: Typical multicore architecture.

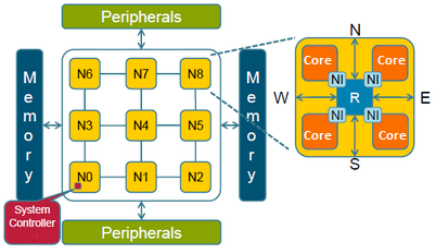



Figure 1.4: A many-core architecture.

In the last few years, multicore processors have erupted into both computing markets due to physical limitations facing scalable performance of classic uniprocessor architectures. By integrating multiple cores within a chip, different applications can be scheduled concurrently on the same platform. More importantly, a single application can be executed simultaneously on multiple cores through parallelization. These features allow for an efficient exploitation of the hardware computing capabilities, while also reducing size, weight, power and cost requirements. Hence, multicore architectures are increasingly considered as the solution to meet the ever-more demanding cost and performance requirements across the whole computing spectrum (Ungerer et al., 2010). However, traditional multicore architectures have not enjoyed an enduring all-around success in either computing domain mainly due to the shared resources and interconnect topology employed. In the RTES domain, we have witnessed the existence of timing anomalies which compromised a predictable execution pattern and worst-case estimates. In the HPC domain, increasing the number of cores stopped translating into proportional performance scalability due to the communication bottleneck observed in the bus-based interconnect, which is shared for inter-core, memory and I/O accesses. In order to overcome such limitations, main hardware manufactures have recently started shipping chips containing dozens or hundreds of simpler cores, interconnected by modular networks-on-chip. Kalray MPPA (de Dinechin et al., 2013) (shipping versions feature 256 cores) in the embedded domain and the Intel MIC (Intel, 2017) (Intel Xeon Phi features up to 72 cores) in the HPC domain are examples of these new powerful and scalable architectures known as "many-core". Figures 1.3 and 1.4 illustrate typical differences between multicore and many-core platforms.

Given the recent market trends for predictable performance and the advent of many-core systems, there is a strong push towards parallel computing. The parallelization is typically achieved by splitting applications into multiple smaller sequential computation units which may run simultaneously on different cores, thus benefiting from the cooperation between processing elements. This intra-task parallelism can be expressed by the programmers using parallel programming models such as Cilk (Frigo et al., 1998), Intel's Parallel Building Blocks (Intel, 2010), OpenMP (OpenMP Architecture Review Board, 2018) and Java Fork/Join (Lea, 2000). These high-level parallel frameworks seek to reduce the complexity of multicore programming by giving programmers abstract execution models, where programmers annotate their applications to suggest the parallel decomposition. In some cases, the annotations act simply as hints that can be

ignored and safely replaced with sequential counterparts. Both the parallel decomposition itself and mapping of computations to cores are the responsibility of the language implementation and, more specifically, of the run-time scheduler. While parallel programming models offer the degree of flexibility and optimization desired in HPC systems, they are not easily applicable to RTES since they compromise the ability of deriving reliable time bounds. Similarly conclusions can be drawn both ways. Therefore, it is of paramount importance to investigate new techniques that address the time predictability and parallel execution challenges now transverse to both computing domains.

In order to capture such parallelization opportunity and overcome the limitations of the *sequential* task models, the RTES community has been actively proposing new *parallel* task models and respective schedulability analysis to the new many-core embedded designs (Lakshmanan et al., 2010; Axer et al., 2013; Baruah et al., 2012; Li et al., 2013; Saifullah et al., 2011; Chwa et al., 2013; Andersson and de Niz, 2012). These important initial steps towards the integration of high-performance and real-time requirements have been strengthen by the outcome of two European projects. The parMERASA project (parMESARA Project, 2014) developed a customized multicore hardware design that is timing analyzable, and specialized software parallel patterns to parallelize industrial applications, together with WCET analysis and verification tools. Recently, the P-SOCRATES project (P-SOCRATES Project, 2016) devised a complete and coherent software system stack for the design, analysis and execution of real-time parallel applications onto COTS-based many-core embedded architectures such as Kalray MPPA and Texas Instruments Keystone II.

This dissertation takes further steps in addressing the time-criticality and parallelization challenges, with particular emphasis on proving guarantees on the timing behavior of the system in the presence of execution parallelism. That is, we focus on scheduling and analysis methodologies to determine if a set of real-time parallel applications is able to meet their stringent deadlines when executing jointly on a multi/many-core platform. Application's performance is boosted by the features of the targeted parallel programing model. To this end, we consider a task model that supports general parallel applications with functional and extra-functional dependencies, called the *DAG model*. It reflects sophisticated types of dynamic, fine-grained and irregular parallelism typical from HPC applications, while having strong similarities with the current OpenMP tasking model (the de facto standard for parallel programming on shared memory systems (OpenMP Architecture Review Board, 2018)). We believe that the outcome of these research studies will contribute to significantly improve the performance capabilities of future real-time embedded systems, while also broadening the spectrum of target applications.

## 1.3 The sporadic directed acyclic graph (DAG) model

In this section, we define the base parallel task model and notation used throughout this dissertation. Extensions or deviations will be handled in the corresponding chapters.

We consider a set of $n$ sporadic real-time tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ to be scheduled by a preemptive fixed-priority algorithm on a platform $\Pi$ composed of $m$ unit-speed processors. We assume that priorities are per-task and that task $\tau_i$ has higher-priority than $\tau_k$ if $i < k$. Each task $\tau_i$ is characterized by a 3-tuple $(G_i, D_i, T_i)$ with the following interpretation. Task $\tau_i$ is a recurrent process that releases a (potentially) infinite sequence of *jobs*, with the first job released at any time during the system execution and subsequent jobs released at least $T_i$ time units apart. Every job released by $\tau_i$ has to complete its execution within $D_i$ time units from its release. We consider that $\tau$ is comprised of constrained deadline tasks, i.e., $D_i \leq T_i, \forall i$.

Each job of $\tau_i$ is modeled by a DAG $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$ is a set of $n_i$ nodes and $E_i \subseteq (V_i \times V_i)$ is a set of directed edges connecting any two nodes. Each node $v_{i,j} \in V_i$ represents a computational unit (referred to as *subtask*) that must execute sequentially. A subtask $v_{i,j}$ has a worst-case execution time (WCET) denoted by $C_{i,j}$. Each directed edge $(v_{i,a}, v_{i,b}) \in E_i$ denotes a precedence constraint between the subtasks $v_{i,a}$ and $v_{i,b}$, meaning that subtask $v_{i,b}$ cannot execute before subtask $v_{i,a}$ has completed its execution. In this case, $v_{i,b}$ is called a *successor* of $v_{i,a}$, whereas $v_{i,a}$ is called a *predecessor* of $v_{i,b}$. A subtask is then said to be *ready* if and only if all of its predecessors have finished their execution. For simplicity, we will omit the subscript $i$ when referring to the subtasks of task $\tau_i$ if there is no possible confusion. A subtask with no incoming (resp., outgoing) edges is referred to as a *source* (resp., a *sink*) of the DAG. Without loss of generality, we assume that each DAG has a single source $v_1$ and a single sink $v_{n_i}$. Note that any DAG with multiple sinks/sources complies with this requirement, simply by adding a dummy source/sink with zero WCET to the DAG, with edges from/to all the previous sources/sinks.

For each subtask $v_j \in V_i$, its set of direct predecessors is given by $pred(v_j)$, while $succ(v_j)$ returns its set of direct successors. Formally, $pred(v_j) = \{v_k \in V_i \mid (v_k, v_j) \in E_i\}$ and $succ(v_j) = \{v_k \in V_i \mid (v_j, v_k) \in E_i\}$. Furthermore, $ances(v_j)$ denotes the set of ancestors of $v_j$, defined as the set of subtasks that are *either directly or transitively* predecessors of $v_j$. Analogously, we denote by $desce(v_j)$ the descendants of $v_j$. Formally, $ances(v_j) = \{v_k \in V_i \mid v_k \in pred(v_j) \vee (\exists v_\ell, v_\ell \in pred(v_j) \wedge v_k \in ances(v_\ell))\}$ and $desce(v_j) = \{v_k \in V_i \mid v_k \in succ(v_j) \vee (\exists v_\ell, v_\ell \in succ(v_j) \wedge v_k \in desce(v_\ell))\}$. Any two subtasks that are not ancestors/descendants of each other are said to be *independent*. Independent subtasks may execute in parallel.

**Definition 1** (Path)**.** *For a given task $\tau_i$ , a path $\lambda = (v_1, \ldots, v_{n_i})$ is a sequence of subtasks $v_j \in V_i$ such that $v_1$ is the source of $G_i$ , $v_{n_i}$ is the sink of $G_i$ , and $\forall v_j \in \lambda \setminus \{v_{n_i}\}$, $(v_j, v_{j+1}) \in E_i$.*

Informally, a path $\lambda$ is a sequence of subtasks from the source to the sink in which there is a precedence constraint between any two adjacent subtasks in $\lambda$. Thus, there is no concurrency between the subtasks that belong to a same path. The length of a path $\lambda$, denoted $len(\lambda)$, is the sum of the WCET of all its subtasks, i.e., $len(\lambda) = \sum_{\forall v_j \in \lambda} C_j$.

**Definition 2** (Length of a task)**.** *The length $L_i$ of a task $\tau_i$ is the length of its longest path.*

**Definition 3** (Critical path)**.** *A path of $\tau_i$ that has a length $L_i$ is a critical path of $\tau_i$.*

Note that when the number of cores $m$ is greater than the maximum possible parallelism of $\tau_i$ (for instance, $m \geq n_i$), the length $L_i$ represents the worst-case response time (WCRT) of $\tau_i$ in

isolation (also known as the *makespan* of the graph). Therefore, an obvious necessary condition for the feasibility of $\tau_i$ is $L_i \leq D_i$.

**Definition 4** (Workload)**.** *The workload $W_i$ of a task $\tau_i$ is the sum of the WCET of all its subtasks, i.e. $W_i = \sum_{j=1}^{n_i} C_j$.*

Finally, we prove the following property on $\tau_i$'s execution and its critical path.

**Lemma 1.** *At most $W_i - \max\{0,\ L_i - \ell\}$ units of workload can be executed by a job of $\tau_i$ in a window of length $\ell$.*

*Proof.* By Def. 1, all subtasks in a critical path have precedence constraints and must therefore execute sequentially. Since the length of every critical path is $L_i$, $\ell$ time units after its release, a job of $\tau_i$ must still execute during at least $\max\{0,\ L_i - \ell\}$ time units to complete. Hence, at most $W_i - \max\{0,\ L_i - \ell\}$ units executed in the interval of length $\ell$. □

Since the workload of a DAG task $\tau_i$ implies that each subtask of a critical path must execute to its WCET, which takes in total no less than $L_i$ time units irrespective of the schedule, the next corollary follows.

**Corollary 1.** *No schedule of $G_i$ whose length is shorter than $L_i$ can accommodate $W_i$ units of workload.*

## 1.4 Problem description and thesis statement

In this dissertation, we study the problem of scheduling and mapping real-time parallel applications modeled as DAG tasks on top of homogeneous multiprocessor systems. The prime goal is to address the time-criticality intrinsic to real-time systems, while tackling the challenges facing parallelization. Hence, we focus on design choices that favor or simplify the worst-case behavior of the overall system, and on static analysis to guarantee a priori that all these highly concurrent and work-intensive applications will fulfill their stringent timing requirements. Nevertheless, we acknowledge that the benefits of parallelization may be wasted if the functional properties of the DAGs are neglected.

We consider a priority assigment only at the task-level to further increase the synergy with current run-time environments for parallel workloads. Accordingly, we addressed a set of schedulability analysis problems for multiprocessor systems under classical preemptive scheduling algorithms by exploring the internal structure of the DAGs. Namely, both global and partitioned paradigms, as well as applications with conditional execution.

The central preposition of this dissertation is that:

**Schedulability analysis for real-time multiprocessor systems composed of DAG tasks can be significantly improved by exploiting the internal structure of such parallel tasks.**

## 1.5 Contributions

In order to evaluate the thesis, we derive a set of scheduling and mapping methods that leverage the internal parallel structures to tighten the schedulability of general DAG tasks running atop a multiprocessor systems. Our contributions include:

- C1 - A sufficient schedulability test for GFP scheduling of sporadic DAG tasks both with constrained and arbitrary deadlines (Fonseca et al., 2017).

  (a) A characterization of a DAG's execution pattern under a certain schedule, called *workload distribution*;

  (b) A new worst-case scenario and respective sliding window algorithm for maximizing the interfering workload generated by a higher priority task;

  (c) Two novel techniques to derive workload distributions and upper-bounds for the worst-case workload produced by both the carry-in and carry-out jobs of an interfering task;

  (d) A response time analysis for constrained deadline DAG tasks that dominates the response time analysis proposed by (Melani et al., 2017). The RTA is based on the concept of problem window, a technique that has been extensively used to study the schedulability of sequential tasks in multiprocessor systems;

  (e) An extension of the response time analysis to the general case of DAG tasks with arbitrary deadlines, which accounts for the interference exerted by multiple carry-in jobs.

- C2 - Two sufficient schedulability tests for the partitioned scheduling of sporadic DAG tasks with constrained deadlines (Fonseca et al., 2016).

  (a) A novel path-based response time analysis for fixed-priority DAG tasks with arbitrary given mappings;

  (b) A model transformation and respective analytical enhancements so that theory on self-suspending tasks can be used to analyze partitioned DAG tasks;

  (c) An algorithm to parse a path of a DAG, while characterizing its worst-case scheduling behavior and inter-core dependencies;

  (d) A DAG partitioning algorithm to minimize the number of cores required to guarantee the feasibility of a partitioned DAG task;

  (e) A duplication-based mapping strategy that allows a DAG task to be modeled as a set of independent sequential tasks, one sequential task for each partition derived. As a result, traditional multiprocessor partitioned schedulability tests and allocation methods can be safely applied.

- C3 - A task model for real-world applications with conditional execution (Fonseca et al., 2015).

(a) The identification of an open problem: control-flow information cannot be neglected in the analysis of conditional parallel tasks;

(b) A new parallel task model, called *multi-DAG*, which generalizes the sporadic DAG model by capturing the different flows of execution that stem from conditional statements;

(c) A two-step algorithm to construct a DAG of servers[1] that encapsulate the execution of a multi-DAG task. With the equivalence, existing schedulability analysis for non-conditional DAG tasks under a work-conserving scheduling algorithm can safely be applied to conditional DAG tasks;

(d) A mapping rule to arbitrate the assignment of subtasks to servers and guarantee that a multi-DAG task can be fully and correctly executed by the corresponding DAG of servers, irrespective of the conditional branches taken at run-time.

- C4 - A relevant sufficient schedulability test for uniprocessor FP scheduling of sporadic self-suspending tasks with constrained deadlines, supporting contribution C2, was also developed in a collaborative effort (Nelissen et al., 2015).

  (a) A counter-example showing that a well-established claim, regarding the critical instant for a sporadic self-suspending task scheduled together with higher priority sequential tasks, was incorrect.

  (b) A new set of conditions for such critical instant which highlights the complexity of the problem;

  (c) An algorithm to compute the exact WCRT of a self-suspending task with a single suspension region when every higher priority task is sequential;

  (d) A mixed-integer linear programming (MILP) formulation to estimate the WCRT of multiple self-suspending tasks with an arbitrary number of suspension regions.

It is our belief that, together, these four contributions (C1, C2 and C3 as main, and C4 as secondary) successfully achieve the main goal of this thesis, and play an important role in advancing the state-of-the-art on the modeling and analysis of real-time systems comprised by general parallel tasks.

## 1.6 Outline

The remainder of this manuscript is structured as follows.

Chapter 2 presents a concise description of the state-of-the-art on real-time parallel task models and respective schedulability results.

[1] In the context of this PhD, the term "server" is employed with the same meaning as in (Baruah et al., 2002), for instance. Servers are the entities to be scheduled on the cores. Each server has a pre-defined cpu-budget to be "consumed" through the execution of ready tasks, every time a server is granted a core. A task cannot execute within a server if its budget is exhausted. However, our DAG of servers only executes subtasks of the corresponding multi-DAG and each server cannot be released if its precedence constraints are not satisfied.

In Chapter 3 and related to contribution C1, we derive a schedulability analysis for sporadic DAG tasks scheduled by GFP. More specifically, throughout Sections 3.1 to 3.3 we motivate the need to take into account the precedence constraints within the DAGs, while providing the necessary background on the multiprocessor response time analysis for parallel tasks (Section 3.2) and introducing the proposed worst-case scenario for the interfering workload of the higher priority tasks (Section 3.3). In Sections 3.4 and 3.5, we show how to more accurately characterize and upper-bound the worst-case carry-in and carry-out workloads, respectively. We then leverage these new upper-bounds to derive improved response time analysis both for constrained (Section 3.6) and arbitrary deadline tasks (Section 3.7). Finally, Section 3.8 reports the experimental results, showing substantial schedulability improvements in comparison to state-of-the-art techniques.

In Chapter 4 and related to contribution C2, we propose two schedulability analysis for sporadic DAG tasks scheduled according to the partitioned paradigm. Section 4.3 introduces a novel response time analysis for DAG tasks under partitioned FP (PFP) scheduling, in which each subtask is statically assigned to a specific core. In Section 4.4, we (1) show that a partitioned DAG task can be modeled as a set of self-suspending tasks, (2) present an algorithm to traverse a DAG and characterize its worst-case scheduling behavior, and (3) discuss how to enhance existing response time analyses for sporadic self-suspending tasks in order to estimate the WCRT of a partitioned DAG task. Section 4.5 generalizes the analysis to the case of multiple partitioned DAG tasks. Throughout Section 4.6, we present a schedulability test under partitioned EDF (PEDF) scheduling and a subtask-to-core mapping based on a duplication technique. First, in Section 4.6.1, we discuss the complexity and pessimism in the previous analysis. Then, in Section 4.6.2, we propose a DAG partitioning algorithm to reduce the number of cores required for feasibility via the duplication of key subtasks. This algorithm also eliminates every single cross-core dependency such that all resulting partitions are independent of each other. Thus, the problem of scheduling a set of partitioned DAGs becomes equivalent to the problem of scheduling a set of sequential tasks on multiprocessors in a partitioned manner, as we show in Section 4.6.3. Finally, the performance of both schedulability tests is compared against other scheduling methods in Section 4.7.

Related to contribution C3, Chapter 5 exposes the problem introduced by conditional DAGs and presents a transformation technique so that existing schedulability tests for non-conditional parallel tasks can be applied to such general model. In Sections 5.1 and 5.2, we highlight the importance of explicitly modeling conditional parallel constructs by showing that is rather complex to determine the actual worst-case DAG structure for the overall system. Hence, in Section 5.3, we propose a multi-DAG model that captures control-flow information through the enumeration of all feasible execution flows. Sections 5.4 and 5.5 then present a two-step algorithm to construct a unique DAG of servers for each multi-DAG task, and a mapping rule to arbitrate the assignment of subtasks to servers, so that any execution flow can fully and correctly be executed by the corresponding DAG of servers. The transformation ensures that schedulability analysis can be performed over the derived DAGs, bridging the gap between previous parallel task models and the vicissitudes of conditional DAG tasks. Finally, we discuss the benefits and limitations of this original solution in Section 5.6.

Chapter 6 presents concluding remarks and future research directions.

In Appendix A and related to contribution C4, we present a uniprocessor response time analysis for sporadic self-suspending tasks. In particular, Section A.4 debunks some conceptions regarding the worst-case release pattern for this task model formalized in Section A.3. In Section A.5, we then develop an algorithm to compute the exact WCRT of a self-suspending task with a single suspension region when the higher priority tasks are sequential. Since the exact test is intractable in the general case, Section A.6 proposes a MILP formulation for computing an upper-bound on the WCRT of a self-suspending task with an arbitrary number of suspension regions. This formulation is extended in Section A.7 to consider the case where multiple self-suspending tasks interfere with each other. Finally, an experimental evaluation is provided in Section A.8.

Counter examples to previously published works are provided in Appendix B.

# Chapter 2

# Related Work

Most results in real-time scheduling for multiprocessor systems concentrate on sequential task models (see (Davis and Burns, 2011) for a comprehensive survey). While these works enable several tasks to execute concurrently on the same multiprocessor host and meet their deadlines, they do not allow individual tasks to take advantage of a multiprocessor platform by exploiting their parallelization opportunities.

The parallelization is typically achieved by splitting applications into multiple smaller sequential computation units (the subtasks) which may run simultaneously on different cores. Such intra-task parallelism grants a more efficient use of the new powerful architectures and boosts applications performance. Software parallelism can be expressed by the programmers using frameworks such as Cilk (Frigo et al., 1998), Intel's Parallel Building Blocks (Intel, 2010), OpenMP (OpenMP Architecture Review Board, 2018) and Java Fork/Join (Lea, 2000). For real-time systems, this means that a task $\tau_i$ with utilization higher than one ($U_i > 1$), or execution requirements exceeding the deadline ($C_i > D_i$), now has the chance to fulfill its stringent timing constraints.

The problem of scheduling parallel tasks has been extensively studied in the general purpose and high-performance computing domains (Kwok and Ahmad, 1999; Drozdowski, 2009). Algorithms such as work-stealing (Blelloch et al., 1999), list scheduling (Kwok and Ahmad, 1996), clustered-based scheduling (Yang and Gerasoulis, 1994) and duplication-based scheduling (Ahmad and Kwok, 1998) provide invaluable solutions to many graph-theoretical computing problems — this list is by no means exhaustive. However, all such work focuses on non-recurrent tasks and concerns mostly with load balancing, energy-consumption, minimization of schedule length (makespan) or admission control. Since real-time constraints and worst-case scenarios are typically not contemplated, these works are not fully applicable to real-time embedded systems.

For the real-time community, a shift from sequential to parallel programming paradigms entails the adoption of more expressive task models to characterize (i) the distinct subtasks that compose a parallel application, and (ii) all the *precedence constraints* that define the relation between the subtasks, and (iii) internal timing properties. Several works have already been proposed to tackle the schedulability of real-time parallel tasks running atop a multicore platform, as we shall revise next. Related work on self-suspending tasks is deferred to Section A.2.

## 2.1 Early work

First results applicable to real-time parallel tasks date from 1989 when Han and Lin (Han and Lin, 1989) have shown the NP-hardness of preemptive scheduling parallel jobs, and the intractability of many parallel scheduling problems. The non-preemptive case was later studied by Wang and Cheng (Wang and Cheng, 1992) which also proposed a heuristic based on the makespan metric.

From an optimization point of view, some research has studied cache-aware schedulers for multi-threaded tasks (Anderson and Calandrino, 2006; Calandrino and Anderson, 2009). The former considers Pfair algorithm and encourage tasks of the same weight to be co-scheduled in order to minimize cache misses, whereas the latter shows a significant performance improvement, with a slight overhead trade-off, when their cache-aware scheduler performs an accurate profiling.

According to the way parallel tasks may exploit the available computing resources, Goossens and Berten (Goossens and Berten, 2010) classified a parallel task as: *rigid*, if the number of processor assigned to the task is specified at design time; *moldable*, if the number of processors assigned to the task is determined at run-time but cannot change throughout its execution; or *malleable*, if the number of processors assigned to the task can change during its execution. The work in (Goossens and Berten, 2010) also presented a gang-based scheduling algorithm for rigid tasks. Regarding moldable tasks, Manimaran et al. (Manimaran et al., 1998) considered non-preemptive EDF, whereas Kato and Ishikawa (Kato and Ishikawa, 2009) proposed the Gang EDF algorithm. For malleable tasks, Lee and Lee (Lee and Lee, 2006) proposed an algorithm that derives a feasible schedule using as few processors as possible, while Collete et al. (Collette et al., 2008) study the feasibility problem under global scheduling.

In the context of distributed systems, Tindell and Clark (Tindell and Clark, 1994) introduced an end-to-end schedulability analysis for a sequence of events called transactions, which was later refined by Palencia et al. (Palencia et al., 1997). Parallelism is expressed through these transactions. Enhancements to this analysis were then proposed in (Palencia and Gonzalez Harbour, 1998) by considering offsets and in (Palencia and Harbour, 1999) by considering precedence relations. EDF systems have also been addressed (Palencia and Harbour, 2003).

Regarding task models whose jobs are not parallel themselves but that allow partial parallelism within a task due to the high expressiveness of the model, Stigge et al. introduced in (Stigge et al., 2011) the digraph real-time task model. In their model, arbitrary directed graphs express the release pattern of different types of jobs in terms of timing and order. Nevertheless, only uniprocessor systems are considered.

## 2.2 Fork-join model

Owing to the intra-task parallelism generated by the prominent parallel programming models (e.g., OpenMP (OpenMP Architecture Review Board, 2018) and Cilk (Frigo et al., 1998)), one the first more realistic parallel task models proposed in the literature was the fork-join model. In this model, a task is represented as an interleaved sequence of sequential and parallel segments, always

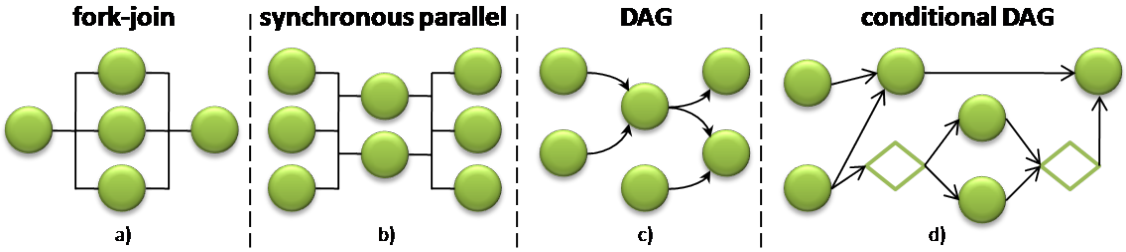



Figure 2.1: Different models of parallel real-time tasks.

starting with a sequential segment. Thus, nested parallelism is not allowed. Parallel segments comprise multiple subtasks. Any subtask within a segment becomes eligible for execution only when the preceding segment is completed. Usually, the number of subtasks spawned in each parallel segment is fixed and may not exceed the number of cores in the system. Conceptually, the fork-join model characterizes a master thread of execution that at certain points is divided in multiple smaller threads that may execute in parallel. As soon as all such threads complete their execution, the master thread is resumed. The fork-join model is depicted in Fig. 2.1 inset a).

In (Lakshmanan et al., 2010) the authors studied the scheduling of periodic implicit deadline fork-join tasks through a decomposition algorithm, and proved a *capacity augmentation bound*[1] of 3.42 for partitioned DM (PDM). The "decomposition" process consists in assigning independent release offsets and virtual deadlines to each subtask in a DAG, in order to bypass the internal execution semantics. Different subtasks may then be scheduled and analyzed as independent sequential tasks even if they belong to the same DAG.

The work by Lakshmanan et al. was extended in (Kim et al., 2013a) to global DM (GDM) and an arbitrary number of subtasks in the parallel segments. GDM scheduling was shown to have a capacity augmentation bound of 3.73. Maia et al. (Maia et al., 2017) considered a semi-partitioned approach using work-stealing. Axer et al. (Axer et al., 2013) proposed a response time analysis for tasks with arbitrary deadlines under partitioned fixed-priority scheduling. Unfortunately their analysis is flawed as we report in the Appendix B.

## 2.3 Synchronous parallel model

The synchronous parallel model extends the fork-join model by allowing successive parallel segments and arbitrary number of subtasks in each segment. Still, this model imposes synchronization points at every segment's boundary, meaning that all the subtasks in a segment may begin execution only when all the subtasks of the previous segment have finished their execution. The synchronous model is depicted in Fig. 2.1 inset b).

[1] A scheduling algorithm $\mathcal{A}$ provides a capacity augmentation bound of $b$ if it can schedule any task set $\tau$ satisfying the following two conditions: (i) the total utilization of $\tau$ is at most $m/b$, and (ii) the critical path length of each task $\tau_i$ is at most $1/b$ of its relative deadline (Li et al., 2013). Note that a capacity augmentation bound directly leads to a schedulability test, similarly to a utilization bound.

Capacity augmentation bounds of 4 and 5 were derived in (Saifullah et al., 2011) for GEDF and PDM, respectively, after decomposing each periodic implicit deadline parallel task into a set of constrained deadline sequential tasks. Nelissen et al. (Nelissen et al., 2012) proposed decomposition techniques to optimize the number of cores needed to schedule sporadic synchronous parallel tasks with constrained deadlines, and proved a *resource augmentation bound*[2] of 2 for a class of optimal scheduling algorithms (e.g., PD$^2$ and U-EDF). For GEDF scheduling without decomposition, the authors in (Andersson and de Niz, 2012) proved a resource augmentation bound of 2, while Chwa et al. (Chwa et al., 2013) presented a response time analysis which extends the traditional concept of interference to cope with the parallel behavior of the tasks.

Later, Maia et al. (Maia et al., 2014) borrowed the concept of critical interference introduced in (Chwa et al., 2013) and derived an improved response time analysis for GFP scheduling. Their work considers the worst-case workload that can be generated by both carry-in and carry-out jobs, while taking into account the precedence constraints between the different segments. In the next chapter, we apply this idea to the general case of DAG tasks and borrow from (Maia et al., 2014) the concept of "sliding window".

## 2.4 DAG model

A more flexible and general parallel structure is captured by the DAG model, as considered in this dissertation and formalized in Section 1.3, where a task is instead represented by a directed acyclic graph. Nodes represent subtasks to be sequentially executed and edges define precedence constraints between nodes. According to this model, a subtask becomes ready for execution as soon as all its precedence constraints are satisfied, and independent subtasks may execute in parallel. The DAG model is depicted in Fig. 2.1 inset c).

A task set comprised of a single arbitrary deadline DAG was studied in (Baruah et al., 2012) and a resource augmentation bound of 2 was derived for GEDF. Considering multiple DAGs, Bonifaci et al. (Bonifaci et al., 2013) proved a resource augmentation bound of $2-1/m$ and $3-1/m$ for GEDF and GDM, respectively. Efficient schedulability tests were also proposed in both works. Later, Baruah (Baruah, 2014) generalized the analytical techniques of (Bonifaci et al., 2013) to improve the effectiveness of the schedulability tests in the case of constrained deadlines. Saifullah et al. (Saifullah et al., 2014) extended their result in (Saifullah et al., 2011) to DAG tasks and non-preemptive GEDF. Li et al. (Li et al., 2013) proved a capacity augmentation bound of $4-2/m$ for sporadic DAGs with implicit deadlines under GEDF scheduling, which Qamhieh et al. (Qamhieh et al., 2013) also analyzed considering explicitly the precedence constraints between subtasks. Decomposition algorithms have been proposed in (Qamhieh et al., 2014) and (Jiang et al., 2016) also for GEDF. Serrano et al. (Serrano et al., 2017) proposed a response time analysis for both

[2] A scheduling algorithm $\mathcal{A}$ provides a resource augmentation bound (also called *speed-up factor*) of $b$ as long as the following condition holds: if an optimal scheduling algorithm can schedule any task set $\tau$ on $m$ unit-speed cores, then $\mathcal{A}$ can schedule $\tau$ on $m$ cores of speed $b$ (Kalyanasundaram and Pruhs, 2000). Note that a resource augmentation bound may not provide a schedulability test, since there may be no way to tell whether the optimal scheduling algorithm can schedule a given task set on $m$ unit-speed cores.

the eager and the lazy preemption approaches, under GFP scheduling and a limited preemption model.

While most of these schedulability tests are computationally very efficient, they either (i) do not benefit from the specific timing and functional properties of the task set under analysis (they are derived based on worst-case configurations), or (ii) they rely on decomposition, which incurs severe system overheads due to the magnitude of subtasks and introduces pessimism during the transformation from parallel to sequential workloads. In contrast, we are interested in analyzing the response time of each DAG task without assigning intermediate parameters, according to general fixed-priority preemptive scheduling (which so far has not received much attention) under both global and partitioned paradigms.

In the case of global scheduling, only the works proposed in (Melani et al., 2015) (described briefly in the next subsection and in detail in Section 3.2) and (Parri et al., 2015) are strictly related to our contribution C1, presented in Chapter 3. Parri et al. (Parri et al., 2015) proposed a response time analysis for GEDF and GDM that accounts for the interference suffered and exerted by each subtask instead of each task. According to the authors, their analysis is essentially tailored for arbitrary deadline tasks, and thus it often leads to prohibitive pessimism in the case of constrained deadlines where the self-interference should not take into account multiple job releases. For this reason, experimental results including these two schedulability analyses are reported independently in Section 3.8. In the case of partitioned scheduling, to our knowledge, apart from the results obtained for distributed systems, no work is directly comparable to our contribution C2, presented in Chapter 4. Nonetheless, one may find some similarities in regard to the *federated scheduling* paradigm (Li et al., 2014).

Federated scheduling can be seen as an extreme type of partitioned scheduling where each heavy task (i.e., a task with a density greater than one) is assigned to a set of dedicated processors, and light tasks (i.e., those whose density is smaller than or equal to one) are partitioned on the remaining processors such that all their subtasks execute in a sequential manner. Note that the light tasks can also be globally scheduled, despite the sequential behavior enforced. Li et al. (Li et al., 2014) proposed a federated scheduling algorithm for implicit deadline DAGs which has a capacity augmentation bound of 2 by dedicating $\gamma_i = \lceil (W_i - L_i)/(D_i - L_i) \rceil$ cores to each heavy task $\tau_i$, but there have also been generalizations to constrained deadlines (Baruah, 2015a), arbitrary deadlines (Baruah, 2015b) and mixed-criticality systems (Li et al., 2016a). Jiang et al. (Jiang et al., 2017) proposed a semi-federated approach, where the fractional processing capacity required by a heavy task is scheduled sequentially together with the light tasks instead of dedicating an extra processor, i.e., $\gamma_i = \lfloor (W_i - L_i)/(D_i - L_i) \rfloor$. In order to control the interference suffered by the fractional part, an heavy task is executed through a set of container tasks with a fixed load bound and a dynamic deadline, such that the container task with largest load bound is always occupied and container tasks become empty when time reaches their absolute deadline (causing the execution of some subtasks to be split). However, this solution to the resource waste problem implies frequent migrations (which are non-existent in partitioned scheduling) and sophisticated run-time dispatchers. More details regarding the federated scheduling algorithms are given in

Section 4.7, where we perform an experimental evaluation.

On a side note, work-stealing strategies addressing soft real-time constraints have also been studied in (Nogueira and Pinho, 2012) and (Li et al., 2016b), as well as heterogeneous multiprocessor platforms (Yang et al., 2016).

## 2.5 Conditional DAG model

Consequently to our contribution C3 and research developments reported in Chapter 5, researchers have recently started addressing conditional parallel tasks, by taking into consideration the different flows of execution that a parallel task may experience due to control structures (for example, *if-then-else*) within their code. The conditional DAG model extends the DAG model by defining a new type of node called conditional node. Conditional nodes come in start/end pairs. A conditional start node denotes that only one of its direct successors non-conditional nodes can be selected for execution during one instance of the parallel task. Hence, only one branch between the start and end of the conditional nodes can be undertaken. The model also specifies a set of rules to avoid violating the correctness of the conditional constructs and the semantics of the DAG. Namely, there cannot be any incoming or outgoing edge connecting a node inside a conditional branch to nodes outside that conditional statement or within other conditional branchs. The conditional DAG model is depicted in Fig. 2.1 inset d).

Melani et al. (Melani et al., 2015, 2017) identified two meaningful parameters to characterize the worst-case behavior of a conditional DAG task and presented a response time analysis based on such parameters for GFP and GEDF. The schedulability test proposed in (Melani et al., 2015, 2017) is effective for both conditional and non-conditional DAG tasks as shown by their experimental evaluation. The analysis is based on the concept of "problem window" (Baker, 2003) and assumes that every interfering task $\tau_i$ executes as a uniform block that occupies all $m$ cores for $W_i/m$ time units. In this context, the parameter $W_i$ denotes the worst-case workload among every possible execution flow. Although such abstraction is powerful for conditional DAGs due to the complexity of the problem, it is overkill for regular DAG tasks, where a precise and sound workload characterization can be derived efficiently. Based on this observation, next chapter builds on top of the results of (Melani et al., 2015) considering the case of non-conditional DAG tasks schedule by GFP.

Baruah et al. (Baruah et al., 2015) proved that the speedup factor of $2 - 1/m$ derived for the GEDF scheduling of non-conditional DAG tasks also holds for systems of conditional DAGs, and proposed a sufficient GEDF schedulability test that has pseudo-polynomial run-time. Later, the federated scheduling approach has been applied to conditional DAG tasks (Baruah, 2015c).

# Chapter 3

# Schedulability Analysis for Global Fixed-Priority Scheduling

One of the major sources of pessimism in the RTA of globally scheduled real-time tasks resides on the computation of the interference that tasks impose on each other. This problem is further exacerbated when intra-task parallelism is permitted, because of the complex internal structure of parallel tasks. This chapter considers the GFP scheduling upon $m$ processors of a set of sporadic real-time tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, where each task $\tau_i$ is modeled by a DAG of concurrent subtasks as formalized in Section 1.3. Tasks in $\mathcal{T}$ are sorted by decreasing priorities. In Section 3.7, we address the general case of arbitrary deadline tasks.

Next, we present a RTA based on the concept of problem window, a technique that has been extensively used to study the schedulability of sequential tasks in multiprocessor systems. The problem window approach of RTA usually categorizes interfering jobs in three different groups: carry-in, carry-out and body jobs. We propose two novel techniques to derive more accurate upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks. Those new bounds take into account the precedence constraints between subtasks pertaining to the same DAG. We show that with this new characterization of the carry-in and carry-out workload, the proposed schedulability test offers significant improvements on the schedulability of general DAG tasks for randomly generated task sets in comparison to state-of-the-art techniques. In fact, we show that, while the state-of-art analysis does not scale with an increasing number of processors when tasks have constrained deadlines, the results of our analysis are barely impacted by the processor count in both the constrained and the arbitrary deadline case.

## 3.1 Motivation

A key challenge in the RTA of globally scheduled multiprocessor task systems is to compute an upper-bound on the interference that a task under analysis suffers from the higher priority tasks. The complexity of computing such inter-task interference bound is exacerbated for parallel tasks, DAGs in particular, due to their rich and irregular internal structure. Furthermore, a DAG

Table 3.1: Performance of the schedulability test proposed in (Melani et al., 2015).

| Schedulability ratio (%) | 94 | 63 | 49 | 32 | 24 | 16 | 14 | 10 |
|---|---|---|---|---|---|---|---|---|
| Number of cores | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |

task may also be subject to intra-task interference because its own parallel computations will eventually contend with each other for processing time. Addressing these new challenges without overlooking the precedence constraints between subtasks is fundamental to derive scalable and less pessimistic schedulability analysis.

To the best of our knowledge, the work proposed by Melani et al. (Melani et al., 2015) represents the first attempt at analyzing the schedulability of a set of sporadic DAG tasks with a general GFP scheduling policy. Their RTA is based on the concept of problem window developed originally by Baker (Baker, 2003). This technique consists in estimating the maximum interfering workload produced by a higher priority task in a time interval of arbitrary length by breaking down the interference into different types of jobs. While the work in (Melani et al., 2015) indeed succeeded in upper-bounding the interfering workload generated by DAG tasks, it does so by considering that every higher priority job in the problem window is a compact block of execution which uniformly occupies all the available processors until its completion.

Since most DAGs exhibit different degrees of parallelism throughout their execution and do not necessarily require to constantly access all processors, such abstraction leads to a significant overestimation of the inter-task interference. This extra level of pessimism in the schedulability analysis is evident in the experimental results reported in Table 3.1 (more details about the system configuration are deferred to Section 3.8). Table 3.1 shows the percentage of task sets that are deemed schedulable by the schedulability test proposed in (Melani et al., 2015) when increasing the number of available cores but keeping the platform utilization fixed at 70% and the number of tasks proportional to the number of cores. The steady schedulability performance deterioration visible in Table 3.1 for the aforementioned test is counter-intuitive, as one would expect at least a constant schedulability ratio when the parallelism of the platform is increased and the average task utilization remains unchanged. Motivated by these observations, we propose techniques to derive improved bounds on the inter-task interference by exploiting the knowledge of the precedence constraints in the internal structure of the DAGs.

## 3.2 Interference among DAG tasks

In this section, we introduce the concept of interference for DAG tasks. We also summarize the RTA introduced by Melani et al. (Melani et al., 2015) as it sets the foundations for the schedulability analysis proposed in the upcoming sections. Although their work targets a more general task model, known as "conditional DAG model", empirical evaluation in (Melani et al., 2015) shows that it is also state-of-the-art for the non-conditional DAG tasks considered in this chapter.

A key challenge in the RTA of globally scheduled multiprocessor systems is the computation of the *interference* among tasks. For sequential tasks, the interference exerted on a task $\tau_k$ is defined as the cumulative length of all the time intervals in which $\tau_k$ is ready but cannot be scheduled on any processor due to the concurrent execution of $m$ higher priority tasks. In order to adapt this definition to the parallel structure of DAG tasks, we introduce the notion of critical chain.

**Definition 5** (Critical chain)**.** *The critical chain $\lambda_k$ of a DAG task $\tau_k$ is the path of $\tau_k$ that leads to its worst-case response time $R_k$, with ties broken arbitrarily.*

To determine the worst-case response time of $\tau_k$, we then need to identify such critical chain and compute the maximum possible interference exerted on it. We start by characterizing the interference on a DAG task $\tau_k$.

**Definition 6** (Interference)**.** *The interference $I_k$ on a DAG task $\tau_k$ is the cumulative length of all the time intervals in which at least one subtask that belongs to $\tau_k$'s critical chain is ready but cannot be scheduled on any processor because all m cores are busy.*

Alternatively, the total interference can be expressed as a function of the worst-case interfering workload generated by each task in the system.

**Definition 7** (Interfering workload)**.** *The interfering workload $W_k^i$ imposed by a DAG task $\tau_i$ on a DAG task $\tau_k$ represents the total workload executed by subtasks of $\tau_i$, while at least one subtask that belongs to $\tau_k$'s critical chain is ready but cannot be scheduled on any processor.*

Defs. 6 and 7 also allow us to formulate a bound on the worst-case response time of $\tau_k$ under any work-conserving scheduling algorithm:

$$R_k \leq len(\lambda_k) + I_k = len(\lambda_k) + \frac{1}{m} \sum_{\forall \tau_i \in \tau} W_k^i \tag{3.1}$$

Furthermore, under fixed-priority scheduling, a task $\tau_k$ cannot suffer interference from lower priority tasks. That is, $W_k^i = 0$, $\forall i > k$. However, when $i = k$, we have $W_k^i \geq 0$. That is because other subtasks of $\tau_k$ that *do not* belong to its critical chain may also delay the completion of $\tau_k$ itself. This phenomenon peculiar to parallel tasks is called *self-interference*.

Unfortunately, deriving concrete values for either the overall term $I_k$ or the individual terms $W_k^i$ is computational intractable for non-trivial task sets, otherwise a schedulability test would easily follow from Eq. 3.1. For this reason, an established workaround is to bound the total worst-case interfering workload by analyzing the maximum possible workload that can be produced by each interfering task during the worst-case instance of $\tau_k$. In the sequel, we present the upper-bounds derived in (Melani et al., 2015) for both the self-interference (i=k) and inter-task interference (i<k) components in the context of GFP scheduling, as well as the resulting response time equation.

Regarding the self-interference, in a constrained deadline setting two jobs of a same task $\tau_k$ cannot interfere with each other. That is because one job must finish before the next one is released, otherwise $\tau_k$ would fail to meet its deadline and the system would immediately be deemed

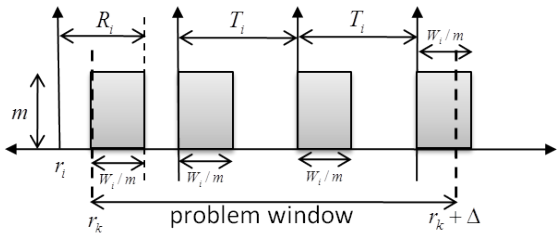



Figure 3.1: Worst-case interfering workload produced by a higher priority task $\tau_i$ in a window of length $\Delta$, as considered in (Melani et al., 2015).

unschedulable. Therefore, the self-interfering workload is independent of the response time of $\tau_k$. Furthermore, due to the absence of priorities at the subtask-level, every subtask *that is not part of $\tau_k$'s critical chain* may potentially contribute to the overall response time of $\tau_k$ and thus to its self-interfering workload $W_k^k$.

Let $M_k$ denote the contribution of DAG task $\tau_k$ to its own response time, i.e., $M_k \stackrel{\text{def}}{=} len(\lambda_k) + W_k^k/m$. It was proven in (Melani et al., 2015) that, for *constrained deadline* tasks, an upper-bound on $M_k$ is given by

$$M_k \leq L_k + \frac{1}{m}(W_k - L_k) \tag{3.2}$$

That is, the self-interfering workload is upper-bounded by $W_k^k \leq W_k - L_k$ (i.e., the remaining workload of $\tau_k$ after excluding the length of its critical path). Importantly, Eq. 3.2 not only provides a bound on the maximum makespan of $\tau_k$ (i.e., its WCRT in isolation) but also ensures that the critical chain $\lambda_k$ can be safely replaced by a critical path of $\tau_k$ in the response time analysis, as long as such critical path is subject to at least the same amount of inter-task interference. Hence, we hereinafter restrict our attentions to a single critical path of $\tau_k$, fixed arbitrarily.

Contrary to the self-interference, the amount of inter-task interfering workload depends on the length of the time interval that we consider. The longer the time interval, the more workload can be generated by the higher priority tasks and thus the larger is the inter-task interference on the analyzed task $\tau_k$. For a time window of length $\Delta$ starting at $\tau_k$'s release, the contribution of a higher priority task $\tau_i$ to the inter-task interfering workload $W_k^i$ is divided in three portions (see Fig. 3.1):

1. **Carry-in**: it accounts for the contribution of jobs of $\tau_i$ with release times before the beginning of the problem window (i.e., before $\tau_k$'s release at time $r_k$) and a deadline after the beginning of the problem window, i.e., after $r_k$. The *carry-in jobs* workload corresponds to the portion of those jobs execution that could not finish prior to $r_k$. Note that for constrained deadline systems, if $\tau_i$ is schedulable, then $\tau_i$ has at most one carry-in job.

2. **Body**: it takes into account the contribution of all subsequent job releases of $\tau_i$ that are fully contained in the window. The contribution of each of the *body jobs* to the interfering workload is upper-bounded by its total execution time $W_i$ (i.e., $\tau_i$'s workload).

3. **Carry-out**: in the related literature, it usually accounts for the contribution of a job of $\tau_i$ with release time within the problem window and deadline after the end of the window (i.e., after $r_k + \Delta$). Yet, in this work we will slightly bend the definition and instead consider that a *carry-out job* is a job that is released within the problem window less than $T_i$ time units before its end (i.e., the carry-out job of $\tau_i$ is released at time $t$ such that $(r_k + \Delta - T_i) < t < (r_k + \Delta)$). Note that our definition is compliant with the state-of-the-art definition when tasks have implicit deadlines (i.e., $D_i = T_i$). The interfering workload of the *carry-out job* corresponds to the portion of its execution that actually overlaps with the time interval $[r_k, r_k + \Delta)$.

In (Melani et al., 2015), the authors formulated a generic bound on the worst-case workload generated by an interfering task $\tau_i$ with constrained deadline within such window of length $\Delta$. This upper-bound, which we state below, relates to the maximum interfering workload imposed by $\tau_i$ on task $\tau_k$ under analysis by fixing $\Delta = R_k$. Hence, $W_k^i \leq \mathcal{W}_i(R_k)$, where $\mathcal{W}_i(\Delta)$ is defined as follows:

$$\mathcal{W}_i(\Delta) \stackrel{\text{def}}{=} \left\lfloor \frac{\Delta + R_i - W_i/m}{T_i} \right\rfloor W_i + \min\left(W_i, m((\Delta + R_i - W_i/m) \bmod T_i)\right) \tag{3.3}$$

Notice that Eq. 3.3 ignores completely the structure of the DAG $G_i$ of $\tau_i$ and corresponds to the scenario depicted in Fig. 3.1. The first term includes both the contributions from the carry-in and body jobs, whereas the second term represents the carry-out component. The interference imposed by $\tau_i$ on $\tau_k$ within the problem window is maximized when: (1) the carry-in job starts executing at the start of the time window and finishes by its WCRT $R_i$, (2) all subsequent jobs are released and executed as soon as possible and (3) every job of $\tau_i$ is assumed to execute on all the $m$ cores during $W_i/m$ time units.

Putting all the pieces together, for a given DAG task $\tau_k$, the schedulability condition $R_k \leq D_k$ relies on a classic iterative RTA. Starting with $R_k = L_k$, an upper-bound on the response time of task $\tau_k$ under GFP scheduling can be derived by a fixed-point iteration on the following expression:

$$R_k = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m}\sum_{\forall i<k} \mathcal{W}_i(R_k) \tag{3.4}$$

## 3.3 Proposed worst-case scenario

Looking at the RTA described in the previous section, it is obvious that one of the major sources of pessimism in the computation of the WCRT is the estimation of the inter-task interference within the problem window. This is clear by examining the execution pattern assumed for every job of the tasks $\tau_i$ that interferes with the analyzed task $\tau_k$ (see Fig. 3.1). All these jobs are assumed to execute as a big compact block which uniformly occupies the $m$ cores during $W_i/m$ time units. Although this assumption provides a safe upper-bound on the interference that they cause, the upper-bound may be greatly improved by not overlooking the rich internal structure of their DAG. Both the precedence constraints and the number of subtasks in the DAG define the possible shapes

that the execution of $\tau_i$ entails. In general, wider and uneven shapes limit the amount of workload that effectively enters the problem window. In fact, most DAGs do not exhibit a constant degree of parallelism equal to $m$ throughout their entire execution (as it is assumed in the state-of-the-art analysis). Instead, the maximum workload they may execute in a given time interval is limited by their internal structure. Such situation is illustrated in Fig. 3.2, where we present the proposed worst-case scenario to maximize the workload of an interfering task $\tau_i$. This observation is further emphasized in the example below.

**Example 1.** *Consider the execution of the task of Fig. 3.3a on $m = 5$ cores. The maximum parallelism attained by the DAG $G_i$ is equal to 5, when subtasks $\{v2, v3, v4, v5, v6\}$ execute simultaneously. Such concurrent execution can only last for 4 time units. After that, the degree of parallelism drops to 2 as $v_7$ becomes ready but $v_2$ has not finished yet. We point out that different execution patterns are possible between the subtasks mentioned so far if we include, for example, interference from higher priority tasks. However, they cannot increase the amount of time during which $G_i$ requires all the available cores. Moreover, both the source $v_1$ and the sink $v_8$ cannot execute concurrently with any other subtask of $G_i$. Therefore, the maximum workload that can be generated by $G_i$ in a window of length 5 is at most 22. Yet, the state-of-the-art analysis presented in Section 3.2 assumes that 25 time units of interfering workload have been generated in a window of length 5.*

Accordingly, we use the internal structure of each DAG to derive more accurate upper-bounds on their contributions to the carry-in and carry-out interfering workload. Notice that, according to this analysis method, the DAG's internal structure does not affect the contribution of the body jobs to the interfering workload since they are fully contained in the problem window. Thus, their exact execution pattern is irrelevant.

Similar to the work in (Melani et al., 2015), our analysis of the inter-task interference is based on the notion of a problem window of length $\Delta$. However, as depicted in Fig. 3.2, we model more accurately the worst-case scenario by taking into account different execution patterns for the carry-in and carry-out jobs. Therefore, the workload produced by task $\tau_i$ is maximized in the problem window $[r_k, r_k + \Delta)$ of $\tau_k$ when: (i) every subtask of the body jobs of $\tau_i$ executes for its WCET; (ii) the carry-in job released at a time $r_i < r_k$ finishes its execution at time $r_i + R_i$ and executes as much workload as possible as late as possible (to maximize its workload contribution to the problem window); (iii) all subsequent jobs are released $T_i$ time units apart; and (iv) the carry-out job starts its execution as soon as it is released and executes as much workload as possible as early as possible (hence maximizing its workload in the problem window).

Our main problem to solve is the lack of a relative reference point between the release time of the carry-in job of $\tau_i$ and the window $[r_k, r_k + \Delta)$. More specifically, the value $r_k - r_i$ is unknown a priori because, as it will become clear in forthcoming sections, the worst-case schedules of the carry-in and carry-out jobs are incomparable. Let $\Delta_i^{CI}$ and $\Delta_i^{CO}$ denote the length of the carry-in portion and the length of the carry-out portion of $\tau_i$'s schedule, respectively. Formally, we have

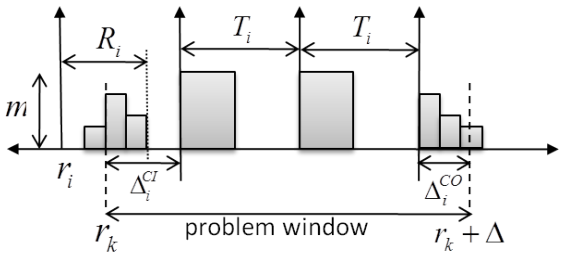



Figure 3.2: New worst-case scenario for the computation of $\tau_i$'s interfering workload.

that[1] (see Fig. 3.2 for visual reference)

$$\Delta_i^{CI} \stackrel{\text{def}}{=} r_i + T_i - r_k \tag{3.5}$$

$$\Delta_i^{CO} \stackrel{\text{def}}{=} \max\{0,\ (r_k + \Delta) - (r_k + \Delta_i^{CI} + \lfloor \frac{\Delta - \Delta_i^{CI}}{T_i} \rfloor_0 \times T_i)\} \tag{3.6}$$

We seek to derive (i) an upper-bound on the interfering workload executed by $\tau_i$'s carry-in job as a function of $\Delta_i^{CI}$, (ii) an upper-bound on the interfering workload executed by $\tau_i$'s carry-out job as a function of $\Delta_i^{CO}$, and (iii) determine concrete values for $\Delta_i^{CI}$ and $\Delta_i^{CO}$ such that the interfering workload of $\tau_i$ on task $\tau_k$ cannot be larger under any possible execution scenario.

To characterize the execution pattern of a carry-in and a carry-out job of $\tau_i$, we introduce the notion of *workload distribution*.

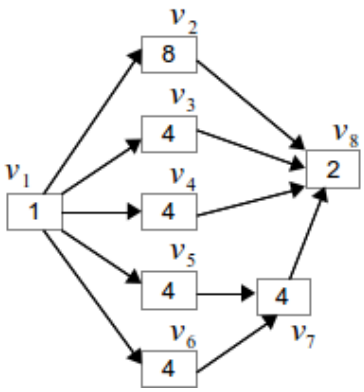
**Definition 8** (Workload distribution)**.** *For a given task $\tau_i$ and a given schedule $S$ of $\tau_i$'s subtasks, the workload distribution $\mathcal{WD}_i^S = [B_1, \ldots, B_\ell]$ describes $S$ as a sequence of consecutive blocks. Each block $B_b \in \mathcal{WD}_i^S$ is a tuple $(w_b, h_b)$ with the interpretation that there are $h_b$ subtasks (height) of $G_i$ executing during $w_b$ time units (width) in $S$, immediately after the completion of the subtasks that execute in the $(b-1)^{th}$ block.*

Note that $\mathcal{WD}_i^S$ does not provide any information about the precedence constraints in the DAG $G_i$, neither it is required for $S$ to be a valid schedule of $G_i$. Also, according to Def. 8, every interfering job of a task $\tau_i$ is modeled in (Melani et al., 2015) with a workload distribution $\mathcal{WD}_i^S$ that comprises only one block $B_1 = (\frac{W_i}{m}, m)$. In the next two sections, we will derive more accurate workload distributions in order to model the schedules of $\tau_i$'s carry-in and carry-out jobs that maximize their contribution to the interference suffered by a lower priority task $\tau_k$.

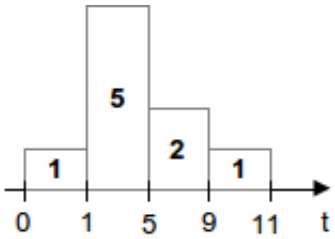
## 3.4 Carry-in workload

This section presents the analysis to compute the carry-in workload of a higher priority task $\tau_i$ in the problem window $[r_k, r_k + \Delta)$ of $\tau_k$. Recall that a carry-in job is a job of $\tau_i$ such that its release time $r_i$ is earlier than $r_k$ and its deadline falls after $r_k$. Therefore, to upper-bound the interfering workload generated by the carry-in job, we need to determine which subtasks of $\tau_i$ may execute

[1] The operator $\lfloor x \rfloor_0 \stackrel{\text{def}}{=} \max\{0,\ \lfloor x \rfloor\}$.




(a) Example of a DAG task.




(b) Workload distribution $\mathcal{WD}_i^{UCI}$. Numbers in blocks represent their height.

Figure 3.3: Example for the carry-in workload.

within the carry-in window $[r_k, r_k + \Delta_i^{CI})$, either fully or partially. Intuitively, to maximize the interfering workload the carry-in job should execute as much workload as possible as late as possible.

For ease of understanding, we will use Fig. 3.3a as an example task throughout our discussion on the carry-in job.

### 3.4.1 Workload distribution of the carry-in job

When the degree of parallelism of the DAG $G_i$ is not constrained by the number of cores (assuming $m = \infty$ for instance), the schedule of $G_i$ that yields the maximum makespan is simply that in which every subtask executes for its WCET. Note that because there are always available cores, each subtask is scheduled as soon as it becomes ready. We call this particular schedule "unrestricted carry-in" ($UCI$). If $f_j$ denotes the relative completion time of each subtask $v_j \in V_i$ in $UCI$, then it holds that:

$$f_j = \begin{cases} C_j & \text{if } v_j \text{ is the source} \\ C_j + \max\limits_{v_h \in pred(v_j)} (f_h) & \text{otherwise} \end{cases} \tag{3.7}$$

Note that the length (makespan) of $UCI$ is given by the completion time $f_{n_i}$ of the sink of $G_i$ and according to Eq. (3.7), $f_{n_i}$ is equal to the critical path length $L_i$.

Assuming that the source of $\tau_i$ starts executing at a relative time 0, the number of subtasks in $UCI$ that execute at any time $t \in [0, L_i)$ can be computed by the function $AS(t)$ defined as

$$AS(t) = \sum_{v_j \in V_i} actv(v_j, t) \tag{3.8}$$

where $actv(v_j, t)$ is equal to 1 if $v_j$ is executing at time $t$ and 0 otherwise. That is,

$$actv(v_j, t) = \begin{cases} 1 & \text{if } t \in [f_j - C_j, f_j) \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

Let $F_i$ be the set of finishing times of the subtasks $v_j \in V_i$ (without duplicates) sorted in non-decreasing order. We build a workload distribution $\mathcal{WD}_i^{UCI}$ modeling the schedule $UCI$ as follows:

- $\mathcal{WD}_i^{UCI}$ has as many blocks as there are elements in $F_i$;
- The $b^{th}$ block of $\mathcal{WD}_i^{UCI}$ is represented by the tuple $(t_{b+1} - t_b, AS(t_b))$ such that $t_b$ is the $b^{th}$ time instant in the ordered set $\{0\} \cup F_i$.

Built that way, $\mathcal{WD}_i^{UCI}$ models the maximum parallelism of $\tau_i$ at any time $t$ assuming that all subtasks execute for their WCET and are scheduled when they are ready. An example of such workload distribution is depicted in Fig. 3.3b for the DAG presented in Fig. 3.3a.

### 3.4.2 Upper-bounding the carry-in workload

Based on both the workload distribution $\mathcal{WD}_i^{UCI}$ and the WCRT $R_i$ estimated by Eq. (3.4), we compute an upper-bound on the interfering workload produced by one carry-in job of $\tau_i$ within its carry-in window $[r_k, r_k + \Delta_i^{CI})$. To do so, we push the workload distribution $\mathcal{WD}_i^{UCI}$ as much as possible "to the right". We first align the end of $\mathcal{WD}_i^{UCI}$ with the worst-case completion time of the carry-in job of $\tau_i$. That is, we align the end of $\mathcal{WD}_i^{UCI}$ with the time-instant $r_k + \Delta_i^{CI} - (T_i - R_i)$ (see Fig. 3.2). This assumes that the carry-in job of $\tau_i$ is released at $r_k + \Delta_i^{CI} - T_i$ and completes at most at $r_k + \Delta_i^{CI} - T_i + R_i$.

Since the problem window starts at $r_k$ and the carry-in job must complete by $r_k + \Delta_i^{CI} - (T_i - R_i)$, the part of the carry-in job that effectively interferes with $\tau_k$ is given by the subtasks of that job executed in the last $\Delta_i^{CI} - (T_i - R_i)$ time units of its schedule. Therefore, under the schedule $UCI$, the maximum interfering workload released by $\tau_i$'s carry-in job is upper-bounded by the function[2]:

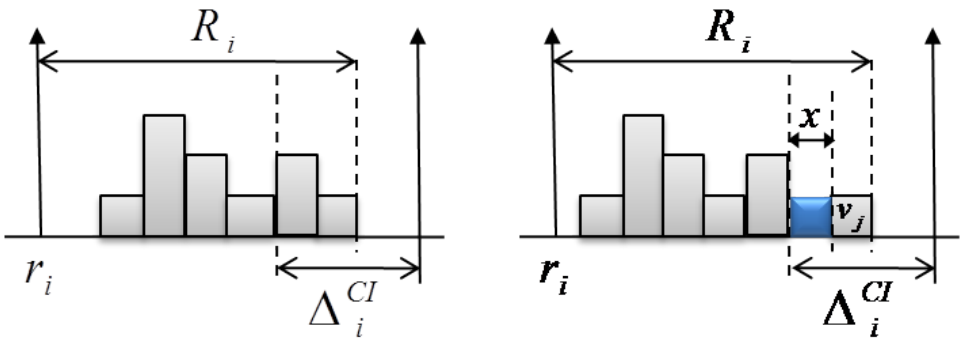
$$CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[ r_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right]_0^{w_b} \tag{3.10}$$

where $r_i \stackrel{\text{def}}{=} \Delta_i^{CI} - T_i$ is the latest time at which $\tau_i$'s carry-in job may be released (assuming that $r_k$ happens at time 0).

Eq. (3.10) returns 0 if $\Delta_i^{CI}$ is less than or equal to $(T_i - R_i)$ (i.e., if the carry-in job of $\tau_i$ completes before the beginning of the problem window). Otherwise, it sums the height $h_b$ of the workload distribution $\mathcal{WD}_i^{UCI}$ in its last $\Delta_i^{CI} - T_i + R_i$ time units.

**Example 2.** *If $\Delta_i^{CI} = 9$, $T_i = 20$, $R_i = 15$ and $\mathcal{WD}_i^{UCI}$ is given by the workload distribution presented in Fig. 3.3b, then Eq. (3.10) sums the height of the blocks in the last $\Delta_i^{CI} - T_i + R_i = 4$ time units of $\mathcal{WD}_i^{UCI}$. Hence, it gives us $CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = 2 \times 2 + 1 \times 2 = 6$. If $\Delta_i^{CI}$ was equal to 4, then Eq. (3.10) would return 0 since $\Delta_i^{CI} - T_i + R_i$ is then smaller than 0.*

[2] $[x]_z^y = max\{min\{x, y\}, z\}$, that is, $y$ and $z$ are an upper-bound and a lower-bound on the value of $x$, respectively.




(a) Before delaying the critical path. (b) After delaying the critical path.

Figure 3.4: Interference (blue block) on $\mathcal{WD}_i^{UCI}$ critical path.

We now prove that the interfering workload imposed by the carry-in job of $\tau_i$ is upper-bounded by the workload distribution $\mathcal{WD}_i^{UCI}$, when the end of $\mathcal{WD}_i^{UCI}$ is aligned with the time-instant $(r_k + \Delta_i^{CI} - T_i + R_i)$. Recall that $R_i$ is computed by Eq. (3.4).

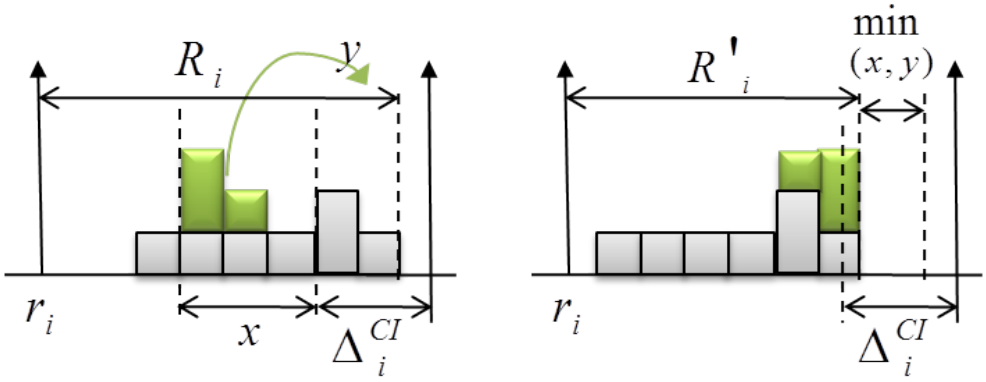
The carry-in workload computed by Eq. (3.10) assumes that (i) all subtasks of $\tau_i$ execute for their WCET, (ii) the number of cores does not limit $\tau_i$'s parallelism and (iii) the carry-in job of $\tau_i$ executes following the workload distribution $\mathcal{WD}_i^{UCI}$ just before its completion time at $r_k + \Delta_i^{CI} - T_i + R_i$. We prove in Lemmas 2 to 4 that those three assumptions maximize the interfering workload of $\tau_i$ in the carry-in window.

**Lemma 2.** *The interfering workload generated by the carry-in job of a higher priority task $\tau_i$ is maximized when all its subtasks execute for their WCET.*

*Proof.* If a subtask $v_j \in V_i$ executes for less than its WCET $C_j$, then either $v_j$ contributes less to the interfering workload (assuming that $v_j$ is executed within the carry-in window), or it may allow its successors (and subsequently its descendants) to be released earlier (note that the release time of subtasks that are not descendant of $v_j$ is not impacted). In the latter case, it may cause those descendants to start executing before (instead of within) the carry-in window and thus reduce the total interfering workload. Similarly, descendants of $v_j$ that were already starting before the beginning of the carry-in window, may complete before the start of the carry-in window, or earlier within the carry-in window. In both cases, the interfering workload in the carry-in window is reduced. □

**Lemma 3.** *Let $R_i$ be an upper-bound on the worst-case response time of $\tau_i$ and let $\mathcal{WD}_i$ be any workload distribution **of length** $L_i$ representing any possible schedule of $\tau_i$. Assume that $\mathcal{WD}_i$ is aligned to the right with the time-instant $r_k + \Delta_i^{CI} - T_i + R_i$. The workload that can be generated by $\mathcal{WD}_i$ in the carry-in window cannot be increased by delaying subtasks in $\tau_i$'s critical path.*

*Proof.* Remember that the length of the workload distribution $\mathcal{WD}_i$ is $L_i$, i.e., the length of $\mathcal{WD}_i$ is equal to the length of the critical path of $\tau_i$. Therefore, there must be a subtask of each $\tau_i$'s




(a) Before delaying subtasks not in the critical path.

(b) After delaying subtasks not in the critical path.

Figure 3.5: $y$ units of workload (green blocks) of $\mathcal{WD}_i^{UCI}$ are moved in the carry-in window.

critical path executing at any time instant between $(r_k + \Delta_i^{CI} - T_i + R_i - L_i)$ and $(r_k + \Delta_i^{CI} - T_i + R_i)$ (because $\mathcal{WD}_i$ is aligned to the right with $r_k + \Delta_i^{CI} - T_i + R_i$). This case is illustrated in Fig. 3.4a.

Now consider the case where $\mathcal{WD}_i$ is subject to self- and/or higher priority interference such that the execution of at least one subtask $v_j$ of a critical path of $\tau_i$ is delayed by $x$ time units.

Postponing the execution of $v_j$ by $x$ time units leads to move both the workload of $v_j$ and its descendants $x$ units "to the right". Because $v_j$ belongs to a critical path of $\tau_i$, the length of $\tau_i$'s carry-in job schedule is increased by $x$ (see Fig. 3.4b). However, because $R_i$ is assumed to be an upper-bound on $\tau_i$'s worst-case response time, $\tau_i$'s carry-in job cannot complete later than $r_k + \Delta_i^{CI} - T_i + R_i$. Therefore, as visualized in Fig. 3.4b, it is not the subtask $v_j$ or its descendants that are moved by $x$ time units "to the right", but instead it is all the workload executed by predecessors of $v_j$ that is pushed by $x$ time units to the left. Hence, the workload executed by $\tau_i$ in the carry-in window $[r_k, r_k + \Delta_i^{CI})$ can only decrease. □

**Lemma 4.** *Let $R_i$ be the upper-bound on the worst-case response time of $\tau_i$* ***computed by Eq.*** *(3.4). Aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $r_k + \Delta_i^{CI} - T_i + R_i$ gives an upper-bound on the maximum interfering workload that can be generated by $\tau_i$ in the carry-in window, independently of the interference imposed on $\tau_i$.*

*Proof.* Remember that the length of $\mathcal{WD}_i^{UCI}$ is $L_i$. Hence, Lemma 3 proved that the workload generated in the carry-in window cannot increase by interfering with the critical paths of $\tau_i$. Therefore, this proof needs to show that the claim is still true even when the interference exerted on $\tau_i$ does not interfere with its critical paths but may delay the execution of other subtasks of $\tau_i$.

The proof is by contradiction. Assume that there is a schedule of $\tau_i$ such that, by delaying subtasks of $\tau_i$, $y$ extra units of workload of $\tau_i$ enter the carry-in window $[r_k, r_k + \Delta_i^{CI})$ comparatively to the workload generated by $\mathcal{WD}_i^{UCI}$ (see Fig. 3.5a for an illustration of $y$ extra units of workload, colored in green, moved in the carry-in window). By Lemma 3, the delayed subtasks do not belong

to any critical path of $\tau_i$ and the length of $\tau_i$'s schedule is therefore not affected, i.e., it remains equal to $L_i$.

Let $v_j$ be any of the delayed subtasks and let $\delta_j$ be the minimum time for which its execution has to be delayed, in comparison to the schedule based on $\mathcal{WD}_i^{UCI}$, so that $v_j$ enters the carry-in window. Let $x$ be the maximum $\delta_j$ over all the delayed subtasks, i.e., $x \stackrel{\text{def}}{=} \max_j\{\delta_j\}$ (see Fig. 3.5a for an illustration of $x$). That is, at least one subtask has been delayed by at least $x$ time units to enter the carry-in window.

Since $m$ subtasks are allowed to execute in parallel on $m$ cores and the critical path of $\tau_i$ is not delayed, postponing a subtasks by $x$ time units implies that at least $(m-1)\times x$ interfering workload executes in parallel with the critical path to prevent the delayed subtask to execute on any of the $m$ cores. Additionally, note that the $y$ units of shifted workload do not interfere with the critical path either, and hence execute in parallel with the critical path, since by assumption the schedule length is not increased. Therefore, we have at least

$$(m-1)\times x + y$$

units of workload that do not interfere with the critical path but execute in parallel with it instead.

Let $R_i'$ be an upper-bound on *the actual* response time of $\tau_i$'s carry-in job under this modified schedule. Since $R_i$ is computed with Eq. (3.4), and Eq. (3.4) assumes that all higher priority jobs and all subtasks that do not belong to the critical path of $\tau_i$ interfere with it, $R_i'$ must be smaller than $R_i$ and we have

$$\begin{aligned} R_i' &\leq R_i - \left(\frac{(m-1)\times x + y}{m}\right) \\ &\leq R_i - \left(\frac{m\times y}{m} + \frac{(m-1)\times(x-y)}{m}\right) \\ &\leq R_i - y - \frac{(m-1)\times(x-y)}{m} \end{aligned} \tag{3.11}$$

We analyse two cases:

- If $y \leq x$, then the last term in (3.11) is positive and we have $R_i' \leq R_i - y$. Hence, the response time of $\tau_i$ and thus the length of $\tau_i$'s schedule in the carry-in window has been reduced by at least $y$ time units (see Fig. 3.5b). Since at least one subtask of each critical path of $\tau_i$ must execute at each of those time units (because the length of the schedule is $L_i$), the workload in the carry-in window has decreased by at least $y$ time units. This is in contradiction with the assumption that the workload increased in the carry-in window.

- If $y > x$, then the last term of (3.11) is negative and we have $R_i' \leq R_i - y - (x-y) = R_i - x$. Hence, $\tau_i$'s response time has reduced by at least $x$ time units. Therefore, the subtasks that were delayed by $x$ time units could not enter the carry-in workload since the whole schedule of $\tau_i$ is pushed to the left by $x$ time units too (see Fig. 3.5b). Therefore, it contradicts the assumption that extra workload of $\tau_i$ entered the carry-in window by delaying subtasks by $x$ time units.

The two cases above prove the claim. □

We are now ready to state the main result regarding the worst-case carry-in workload of $\tau_i$.

**Theorem 1.** *The interfering workload $W_i^{CI}$ generated by the carry-in job of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CI}$ is upper-bounded by $CI_i(WD_i^{UCI}, \Delta_i^{CI})$.*

*Proof.* The proof follows directly from Lemmas 2 to 4. □

### 3.4.3 Improved carry-in workload

The bound on the worst-case carry-in workload of $\tau_i$ as computed in Equation 3.10 may in some cases be pessimistic since the number of subtasks executing simultaneously in the workload distribution $\mathcal{WD}_i^{UCI}$ (i.e., the height of the blocks) could be greater than the number of cores in the platform. As we know for a fact that no more than $m$ subtasks can run in parallel on $m$ cores, this leads to another upper-bound.

**Lemma 5.** *An upper-bound on the maximum workload that can be generated by a task $\tau_i$ in a carry-in window of length $\Delta_i^{CI}$ is given by* $\max\{0,\ \Delta_i^{CI} - T_i + R_i\} \times m$.

*Proof.* Since $\tau_i$ cannot complete later than $R_i$, we know that $\tau_i$ does not execute during the last $(T_i - R_i)$ time units of the carry-in window (see Fig. 3.2). Therefore, $\tau_i$ executes during at most $\max\{0,\ \Delta_i^{CI} - (T_i - R_i)\}$ time units on $m$ processors within the carry-in window of length $\Delta_i^{CI}$, hence the claim. □

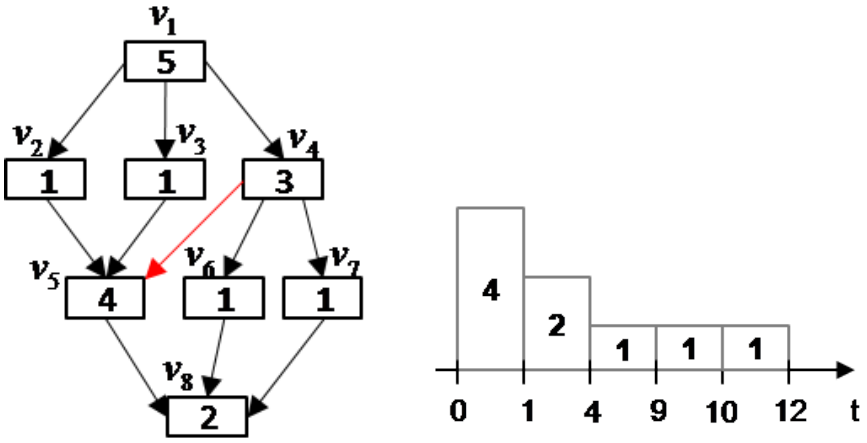
Since this new upper-bound cannot be compared with that given by Equation 3.10, Theorem 2 below shall present an improved upper-bound on $W_i^{CI}$ that is simply the minimum between that given by Equation 3.10 and that presented in Lemma 5.

**Theorem 2.** *The interfering workload $W_i^{CI}$ generated by the carry-in job of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CI}$ is upper-bounded by* $\min\{CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}),\ \max\{0,\ \Delta_i^{CI} - T_i + R_i\} \times m\}$.

*Proof.* Follows from Theorem 1 and Lemma 5. □

## 3.5 Carry-out workload

This section presents the analysis for computing an upper-bound on the carry-out part of the interfering workload of a higher priority task $\tau_i$ in the problem window $[r_k, r_k + \Delta)$ of a task $\tau_k$. The carry-out job is the last job of $\tau_i$ released in the problem window, i.e., its release time is within the open interval $(r_k + \Delta - T_i,\ r_k + \Delta)$. Contrary to the carry-in job, the maximum interference generated by the carry-out job of $\tau_i$ is found when it starts executing as soon as it is released and at its highest possible concurrency level. That is, we are interested in pushing the workload of that job as much as possible "to the left" of the schedule.




(a) An example DAG task. (b) Workload distribution $\mathcal{WD}_i^{UCO}$.

Figure 3.6: Example for the carry-out workload.

Also, contrary to the carry-in and the body jobs, finding an upper-bound on the interfering workload generated by the carry-out job does not necessarily imply that its subtasks execute for their WCET. Indeed, unless the entire carry-out job can contribute to the interference generated by $\tau_i$, one must consider that any of its subtask may instead be instantly processed (i.e., its execution time is $\varepsilon \to 0$). With this assumption, some dependencies may be immediately resolved and the degree of parallelism in the DAG is potentially increased, leading to more workload at the beginning of the carry-out job and potentially within the carry-out window. This remark is emphasized in the example below.

**Example 3.** *Consider the DAG in Fig. 3.6a (including the edge $(v_4, v_5)$ colored red). If every subtask executes for its WCET, then initially only one subtask is active ($v_1$) for 5 time units. On the other hand, if the subtasks $v_1$ and $v_4$ both execute for $\varepsilon$ time units, then the subtasks $v_2, v_3, v_6$ and $v_7$ are instantly ready and there are four subtasks active during the first time unit. Thus, if the carry-out window is only one time unit long, the latter case generates more workload.*

Hence, we seek to derive a schedule that maximizes the cumulative parallelism throughout the execution of the carry-out job. We call this schedule "unrestricted carry-out" (UCO).

### 3.5.1 DAG's maximum parallelism

In order to maximize the workload produced by the carry-out job of $\tau_i$ within the problem window, we need to find an execution pattern such that the overall parallelism cannot be further increased. If the carry-out window is sufficiently short, then the maximum degree of parallelism of $G_i$ maximizes the carry-out workload, as described in Example 3. Ideally, we would like to take the maximum parallelism of the DAG at each time instant as a solution to the problem of maximizing its cumulative parallelism within a time interval of arbitrary length. Unfortunately, this methodology cannot be applied to DAGs, since the scenario that maximizes the parallelism at a certain step may compromise the concurrency among subtasks later on. In fact, as shown in the example

below, whether or not the DAG's maximum parallelism must be considered depends on the length of the carry-out window.

**Example 4.** *Consider the DAG in Fig. 3.6a. The maximum parallelism is four, given by the subtasks* $v_2, v_3, v_6$ *and* $v_7$ *that can execute in parallel for at most* 1 *time unit. Note, however, that every schedule which maximizes the DAG's parallelism does not allow any of the remaining subtasks to execute in parallel — subtasks* $v_1, v_4, v_5$ *and* $v_8$ *have to execute sequentially due to their precedence constraints. Hence, if the maximum parallelism is reached, then the carry-out job cannot produce more than* 5 *units of workload within a window of length equal to* 2*. On the other hand, if subtask* $v_4$ *executes for* 1 *time unit, we can have three subtasks executing in parallel for* 2 *time units: first, subtasks* $v_2, v_3$ *and* $v_4$ *execute in parallel for* 1 *time unit, and then subtasks* $v_5, v_6$ *and* $v_7$ *also execute in parallel for* 1 *time unit. As a result, the latter schedule generates more interfering workload if the carry-out window is* 2 *time units long, but it produces at most* 3 *units of workload when the length of the window is reduced to* 1.

The issue highlighted in Example 4 comes from the potentially very complex connection structures between subgraphs composing the DAG task. Maximizing the parallelism in one subgraph may constrain and hence reduce the achievable parallelism in another subgraph. We simplify the problem at hand by transforming the initial DAG that describes the task in a well-structured, less general, type of DAG, which we call "nested fork-join DAG" (NFJ-DAG) (see below for an explanation on how the transformation is performed and why the transformation is correct). We define a NFJ-DAG[3] recursively as follows.

**Definition 9** (Nested fork-join DAG)**.** *A DAG comprised of two nodes connected by a single edge is NFJ. If* $G_1$ *and* $G_2$ *are two independent NFJ-DAGs, then the DAG obtained through either of the following operations is also NFJ:*
***a) Series composition****: merge the sink of* $G_1$ *with the source of* $G_2$*.*
***b) Parallel composition****: merge the source of* $G_1$ *with the source of* $G_2$ *and the sink of* $G_1$ *with the sink of* $G_2$*.*

The series composition links two NFJ-DAGs one after another, whereas the parallel composition juxtaposes two NFJ-DAGs by merging their sources and sinks. For example, the DAG of Fig. 3.6a is not a NFJ-DAG because it cannot be constructed without violating the rules in Def. 9. However, if the edge colored red $(v_4, v_5)$ is removed, then the DAG becomes NFJ. It is clear from the definition of a NFJ-DAG that maximizing the parallelism of any of its subgraphs cannot limit the maximum parallelism achievable by other subgraphs composing the NFJ-DAG.

#### 3.5.1.1 Transforming a DAG into a NFJ-DAG

Many efficient algorithms exist in the literature to identify if a DAG is NFJ (Valdes et al., 1979; He and Yesha, 1987). However, it is out of the scope of this dissertation to describe how those algorithms work. We assume here that one of those tests is performed on the graph $G_i$ describing $\tau_i$'s

[3] In graph theory, it is known as *two terminal series parallel digraph* (He and Yesha, 1987).

structure. If it turns out that the original DAG $G_i$ is not NFJ, a transformation is required. Traditionally, in graph theory, the transformation is performed by adding new edges between conflicting subtasks, so that the original precedences are preserved (González-Escribano et al., 2002). However, we are interested in removing edges so as to reduce the number of precedence constraints. This way, the set of all the valid schedules of $\tau_i$ (those that satisfy the precedence constraints of its original DAG $G_i$) is a subset of all the valid schedules of the resulting NFJ-DAG. That is because any schedule derived according to the DAG $G_i$ will always respect all the precedence constraints of the NFJ-DAG. As a result, the maximum carry-out workload that can be generated by the NFJ-DAG is at least as large as the maximum carry-out workload that can be generated by the initial DAG $G_i$.

Let us refer to a subtask $v_j$ as a join-node if its "in-degree" is larger than one, i.e. $|pred(v_j)| > 1$. Similarly, we refer to a subtask $v_j$ as a fork-node if its out-degree is larger than one, i.e. $|succ(v_j)| > 1$. According to Def. 9, a DAG (as defined in Section 1.3) is NFJ if and only if it respects the following property.

**Property 1.** *Let $\mathcal{J}_i$ be the set of join-nodes in $V_i$ and let $\mathcal{F}_i$ be the set of fork-nodes in $V_i$. DAG $G_i$ is a NFJ-DAG iff $\forall v_j \in \mathcal{J}_i$, there exists a subgraph $G'$ of $G_i$ such that $v_j$ is the sink of $G'$, the source of $G'$ is a fork-node $v_f \in \mathcal{F}_i$ and*
$\forall v_a \in G' \setminus \{v_f, v_j\}, \forall v_b \in \{succ(v_a) \cup pred(v_a)\}, v_b \in desce(v_f) \cup v_f \wedge v_b \in ances(v_j) \cup v_j$.

*Proof.* The property directly follows from Def. 9, which enforces that any join-node is the result of a *parallel composition*. Hence, for every join-node $v_j$ there must exist a fork-node $v_f$ such that the subgraph $G'$ that has $v_f$ as a source and $v_j$ as a sink is NFJ. Moreover, according to the construction rule defined in Def. 9, there cannot be any edge between a node $v_a \in G'$ and a node $v_b \notin G'$. Therefore, $\forall v_a \in G', \forall v_b \in \{succ(v_a) \cup pred(v_a)\}, v_b \in G'$, implying that $v_b \in desce(v_f) \cup v_f \wedge v_b \in ances(v_j) \cup v_j$. □

Using Property 1, a high-level algorithm for transforming a DAG $G_i$ into a NFJ-DAG $G_i^{NFJ}$ can be defined as follows.

1. Select the unvisited join-node $v_j \in \mathcal{J}_i$ that is the closest to the source of $G_i$.
2. Find all the edges $(v_c, v_j)$ in $E_i$ for which there is no fork-node $v_f \in \mathcal{F}_i$ such that Prop. 1 is true. Call this set the set of conflicting edges $E^C$.
3. Remove as many edges in $E^C$ as needed for join-node $v_j$ to respect Prop. 1 or its in-degree become equal to 1.
4. For each edge $(v_c, v_j) \in E^C$ that was removed, if $succ(v_c) = \emptyset$, add an edge $(v_c, v_{n_i})$ from node $v_c$ to the sink of $G_i$.
5. Mark $v_j$ as visited. Repeat until all join-nodes have been visited.

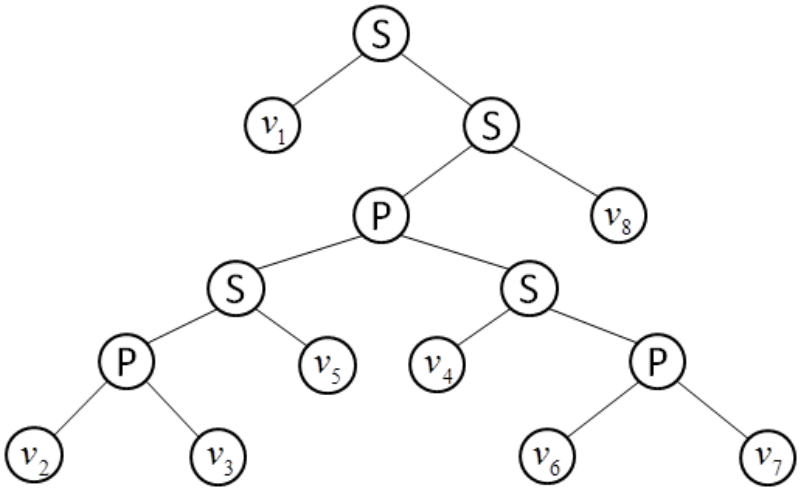



Figure 3.7: Decomposition tree of the NFJ-DAG in Fig. 3.6a.

**Example 5.** *The DAG of Fig. 3.6a has two join-nodes* $\{v_5, v_8\}$*. The above algorithm starts by analyzing join-node* $v_5$*. Since its ancestor* $v_4$ *has two direct successors* $\{v_6, v_7\}$ *which are not ancestors of* $v_5$*,* $(v_4, v_5)$ *is a conflicting edge. Because there is no other conflicting edge with respect to join-node* $v_5$*, our only choice is to remove the edge* $(v_4, v_5)$ *from the DAG. In the next iteration, the DAG is already NFJ as join-node* $v_8$ *does not violate Property 1.*

#### 3.5.1.2 Maximum parallelism in a NFJ-DAG

By Def. 9, a NFJ-DAG can be reduced to a collection of basic DAGs by successively applying series and parallel binary decomposition rules. Therefore, a NFJ-DAG $G_i^{NFJ}$ can be represented by a binary tree $T_i$, called *decomposition tree* (see Fig. 3.7 for an example). Each external node (leaf) of the decomposition tree corresponds to a subtask $v_j \in V_i$, whereas each internal node represents the composition type (series or parallel) applied to its subtrees. That is, the children of a internal node are either smaller NFJ-DAGs or subtasks. A node depicting a parallel or series composition is labeled *P* or *S*, respectively. The algorithm proposed by Valdes et al. (Valdes et al., 1979) can be used to efficiently build the decomposition tree of any NFJ-DAG. Fig. 3.7 shows the decomposition tree of the NFJ-DAG depicted in Fig. 3.6a (without the red edge).

The structure of the decomposition tree allows us to compute the sets of subtasks yielding the maximum parallelism of a NFJ-DAG $G_i^{NFJ}$ in an efficient manner. The recursive function $par(T_i^U)$ defined below returns a set of subtasks in a decomposition tree $T_i^U$ such that all subtasks in $par(T_i^U)$ can execute in parallel and the size of $par(T_i^U)$ is maximum. Note that, in Eq. (3.12) below, $T_i^L$ and $T_i^R$ denote the left and right subtrees of the binary tree $T_i^U$ rooted in node $U$,

respectively.

$$par(T_i^U) = \begin{cases} par(T_i^L) \cup par(T_i^R) & \text{if } U \text{ is a P-node} \\ par(T_i^L) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^L)| \geq |par(T_i^R)| \\ par(T_i^R) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^R)| > |par(T_i^L)| \\ \{U\} & \text{otherwise} \end{cases} \tag{3.12}$$

Eq. (3.12) works as follows. When node $U$ denotes a parallel composition, the maximum parallelism corresponds to the sum of the maximum parallelism of its children. On the other hand, the maximum parallelism in a series composition is given by the maximum parallelism among its children. The recursion of Eq. (3.12) stops when $U$ is a leaf of the decomposition tree and hence corresponds to a subtask in the associated NFJ graph. The set of subtasks in $G_i^{NFJ}$ with maximum parallelism is obtained by calling $par(.)$ for $G_i^{NFJ}$'s decomposition tree.

### 3.5.2 Workload distribution of the carry-out job

As discussed earlier in this section, the carry-out job of an interfering task $\tau_i$ generates the maximum interfering workload when it starts executing as soon as it is released and at its highest possible concurrency level. Therefore, we use the $par(.)$ function defined above to build the workload distribution $\mathcal{WD}_i^{UCO}$ that characterizes the $UCO$ schedule for the carry-out job of $\tau_i$.

The workload distribution $\mathcal{WD}_i^{UCO}$ is constructed using Algorithm 1. In short, the algorithm identifies the maximum number of subtasks that can run in parallel at any point during the execution of the carry-out job as follows. It finds the largest list of subtasks which may execute in parallel according to the decomposition tree of $G_i^{NFJ}$ (line 3). Then, it adds a new block (line 5) to the workload distribution $\mathcal{WD}_i^{UCO}$ with a width equal to the minimum WCET among those subtasks (line 4) and a height equal to the number of elements in the list. Finally, it proceeds by updating the subtasks' execution times in the reduction tree, i.e., decreasing their execution time by the amount of time they executed in parallel (line 6). When a subtask reaches an execution

**Algorithm 1:** Constructing $\mathcal{WD}_i^{UCO}$.

**Input** : $G_i^{NFJ}$, $T_i^{NFJ}$ - A NFJ-DAG and its decomposition tree.
**Output**: $\mathcal{WD}_i^{UCO}$ - Workload distribution of the schedule $UCO$.

1 $\mathcal{WD}_i^{UCO} \leftarrow \emptyset$;
2 **while** $T_i^{NFJ} \neq \emptyset$ **do**
3 $\quad P \leftarrow par(T_i^{NFJ})$;
4 $\quad width \leftarrow min\{C_p \mid v_p \in P\}$;
5 $\quad \mathcal{WD}_i^{UCO} \leftarrow [\mathcal{WD}_i^{UCO}, (width, |P|)]$;
6 $\quad \forall v_p \in P : C_p \leftarrow C_p - width$;
7 $\quad \forall v_j \in T_i^{NFJ}$ such that $C_j = 0$ : remove $v_j$ from $T_i^{NFJ}$;
8 **end**
9 **return** $\mathcal{WD}_i^{UCO}$;

time equal to 0 (it finishes), its corresponding leaf is removed from the decomposition tree (line 7). Whenever a node of the decomposition tree has no children anymore, it is also removed from the tree. Algorithm 1 is called iteratively until all leaves have been removed.

**Example 6.** *The workload distribution $\mathcal{WD}_i^{UCO}$ for the DAG of Fig. 3.6a is presented in Fig. 3.6b. It tells us that the NFJ-DAG in Fig. 3.6a (i.e., the resulting DAG after removing the red edge) can execute with a parallelism of* 4 *during* 1 *time unit. It can execute with a parallelism of* 2 *during* 3 *more time units and then it can finally execute with a parallelism of* 1 *during* 8 *additional time units. Therefore, $\mathcal{WD}_i^{UCO}$ upper-bounds the cumulative parallelism of the original DAG.*

### 3.5.3 Upper-bounding the carry-out workload

Similarly to what was presented for the carry-in workload, an upper-bound on the carry-out interfering workload generated by $\tau_i$ is calculated using the workload distribution $\mathcal{WD}_i^{UCO}$. Recall that $\Delta_i^{CO}$ denotes the length of the carry-out window of $\tau_i$ (see Eq. (3.6)).

The maximum workload executed by $\tau_i$ in any window of length $\Delta_i^{CO}$ is upper-bounded by the cumulative workload found in the first $\Delta_i^{CO}$ time units of the workload distribution $\mathcal{WD}_i^{UCO}$. Such cumulative workload is denoted by $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ and can be computed by the function:

$$CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCO}|} h_b \times \left[\Delta_i^{CO} - \sum_{p=1}^{b-1} w_p\right]_0^{w_b} \tag{3.13}$$

**Example 7.** *If $\Delta_i^{CO} = 3$ and $\mathcal{WD}_i^{UCO}$ is given by the workload distribution presented in Fig. 3.6b, then Eq. (3.13) sums the height of the blocks in $\mathcal{WD}_i^{UCO}$ up to* 3 *time units. As a result, we get $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}) = 4 \times 1 + 2 \times 2 = 8$. If $\Delta_i^{CO}$ was equal to* 10*, then $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ would be equal to* 16*.*

We now prove that $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ is indeed an upper-bound on $W_i^{CO}$.

**Theorem 3.** *The interfering workload $W_i^{CO}$ generated by the carry-out job of a higher priority task $\tau_i$ in a carry-out window of length $\Delta_i^{CO}$ is upper-bounded by $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$.*

*Proof.* We recall that $\tau_i$'s carry-out job generates the maximum interfering workload when it starts executing as soon as it is released and at its highest possible concurrency level.

First, we note that the NFJ-DAG $G_i^{NFJ}$, built from $G_i$ by removing some of $G_i$'s edges, has a concurrency level at least as high as $G_i$. Hence, the workload distribution $\mathcal{WD}_i^{UCO}$ constructed based on $G_i^{NFJ}$ has at least as much workload than $G_i$ in the carry-out window.

Since $\mathcal{WD}_i^{UCO}$ is constructed with Algorithm 1, and because Algorithm 1 computes the maximum parallelism of $G_i^{NFJ}$ at each time $t$, the height of $\mathcal{WD}_i^{UCO}$ on its first $\Delta_i^{CO}$ time units maximizes the workload that $\tau_i$ can generate in the carry-out window.

Finally, because $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ provides the cumulative workload in $\mathcal{WD}_i^{UCO}$ over its first $\Delta_i^{CO}$ time units, $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ upper-bounds the interfering workload that can be generated by $\tau_i$'s carry-out job. □

### 3.5.4 Improved carry-out workload

Note that because the workload distribution $\mathcal{WD}_i^{UCO}$ is built based on the NFJ-DAG of $\tau_i$ and not on its DAG, the length of the schedule $UCO$ may become shorter than $L_i$ when any of the removed edges belongs to the critical path of $G_i$. In fact, the length of $\mathcal{WD}_i^{UCO}$ matches the critical path length of $G_i^{NFJ}$, which may be shorter than the critical path of the initial DAG $G_i$ (since precedence constraints might have been removed).

**Example 8.** *The workload distribution $\mathcal{WD}_i^{UCO}$ presented on Fig. 3.6b has a length of* 12*, while the original DAG (*with *the red edge) in Fig. 3.6a has a critical path composed of $v_1$, $v_4$, $v_5$ and $v_8$ of length $L_i = 14$.*

As stated by Corollary 1, task $\tau_i$ cannot execute $W_i$ time units in less than $L_i$ time units. This allow us to derive a new bound on the worst-case interfering workload of $\tau_i$'s carry-out job.

**Lemma 6.** *The interfering workload $W_i^{CO}$ generated by the carry-out job of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CO}$ is upper-bounded by $W_i - \max\{0, L_i - \Delta_i^{CO}\}$.*

*Proof.* Directly follows from Lemma 1. □

Combining Theorem 3 with Lemma 6, we get an improved upper-bound on $W_i^{CO}$.

**Theorem 4.** *The interfering workload $W_i^{CO}$ generated by the carry-out job of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CO}$ is upper-bounded by* $\min\left\{CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}),\ \Delta_i^{CO} \times m,\ W_i - \max\{0, L_i - \Delta_i^{CO}\}\right\}$.

*Proof.* Because at most $m$ subtasks can execute simultaneously on $m$ cores, $\Delta_i^{CO} \times m$ is an upper-bound on the workload that can execute in a window of length $\Delta_i^{CO}$. Since $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ (Theorem 3) and $W_i - \max\{0, L_i - \Delta_i^{CO}\}$ (Lemma 6) are also upper-bounds on $W_i^{CO}$, so is the minimum between the three values. □

## 3.6 Schedulability analysis for constrained deadline tasks

In Sections 3.4 and 3.5 we have derived upper-bounds on the workload produced by the carry-in and carry-out jobs of $\tau_i$ as a function of $\Delta_i^{CI}$ and $\Delta_i^{CO}$, respectively. Now we show how to balance $\Delta_i^{CI}$ and $\Delta_i^{CO}$ such that the interfering workload in the problem window of length $\Delta$ is maximized. In this section, we assume that all tasks have constrained deadlines (i.e., $D_i \leq T_i$). The case of arbitrary deadlines is considered in Section 3.7. If tasks have constrained deadlines, then at most one job of each higher priority task $\tau_i$ can be a carry-in job, i.e., at most one job of $\tau_i$ can be released before $r_k$ and have a deadline after $r_k$. Similarly, at most one job of $\tau_i$ may be a carry-out job, i.e., there is at most one job of $\tau_i$ that can be the last job of $\tau_i$ released in the problem window.

The difficulty in computing the values $\Delta_i^{CI}$ and $\Delta_i^{CO}$ comes from the fact that the worst-case scenario for $\tau_k$ does not necessarily happen when the problem window is aligned with the start of the carry-in job or the end of the carry-out job (see Fig. 3.2). Furthermore, the positioning of

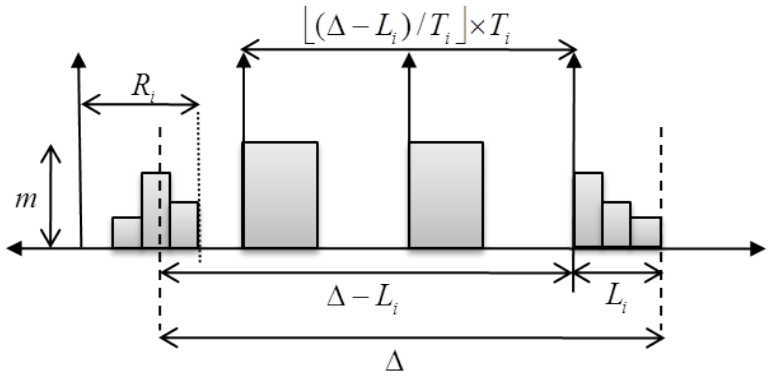



Figure 3.8: Scenario that maximizes the number of body jobs released by $\tau_i$ over $\Delta$.

the problem window of $\tau_k$ relatively to the release pattern of $\tau_i$ may have to vary according to the value of $\Delta$ in order to guarantee that the workload imposed by $\tau_i$ on $\tau_k$ is maximized.

Let $\Delta_i^C$ be the sum of the carry-in and the carry-out windows lengths, i.e, $\Delta_i^C = \Delta_i^{CI} + \Delta_i^{CO}$, and let $\mathcal{W}_i^C(\Delta_i^C)$ be the maximum workload produced by the carry-in and carry-out jobs of $\tau_i$ over $\Delta_i^C$. An upper-bound on the total interfering workload generated by $\tau_i$ in a time interval of length $\Delta$ is therefore given by

$$\mathcal{W}_i(\Delta) = \mathcal{W}_i^C(\Delta_i^C) + \max\left\{0, \left\lfloor \frac{\Delta - \Delta_i^C}{T_i} \right\rfloor\right\} \times W_i \tag{3.14}$$

where the first term is the maximum workload produced by both the carry-in job and the carry-out job of $\tau_i$ and the second term is the maximum number of body jobs that can be released by $\tau_i$ within $(\Delta - \Delta_i^C)$, multiplied by their maximum workload. To use Eq. (3.14), we need to compute $\Delta_i^C$ and $\mathcal{W}_i^C(\Delta_i^C)$. The value of $\Delta_i^C$ can be computed as follows.

$$\Delta_i^C = \Delta - \max\left\{0, \left\lfloor \frac{\Delta - B_i}{T_i} \right\rfloor\right\} \times T_i \tag{3.15}$$

where $B_i$ is the best-case response time (BCRT) of $\tau_i$ when it executes for its worst-case workload. It is given by

$$B_i = \max\{L_i, \frac{W_i}{m}\} \tag{3.16}$$

which was derived using Corollary 1 (i.e., the BCRT of $\tau_i$ cannot be smaller than $L_i$) and the fact that $\tau_i$ cannot execute on more than $m$ processors at a time, hence $B_i$ is lower-bounded by $\frac{W_i}{m}$.

The length $\Delta_i^C$ is thus obtained by aligning the problem window with the earliest completion time of the carry-out job of $\tau_i$ (which takes no less than $B_i$ time units to execute) and removing all the body jobs of $\tau_i$ from the problem window of length $\Delta$ (see Fig. 3.8). This way, the number of full jobs of $\tau_i$ in the problem window is maximized, and so is its interference. Note that the fact that $\Delta_i^C$ is computed by aligning the problem window with the end of $\tau_i$'s carry-out job does not mean that $\tau_i$'s interference is maximized when $\Delta_i^{CO}$ contains the full carry-out job of $\tau_i$. Instead, the window may be shifted left (yet without changing the number of body jobs) to include a larger portion of $\tau_i$'s carry-in job if it increases the total interfering workload generated by $\tau_i$.

**Lemma 7.** *The interfering workload $\mathcal{W}_i(\Delta)$ generated by a higher priority task $\tau_i$ in a window of length $\Delta$ is maximized when $\Delta_i^C$ is computed by Eq. (3.15).*

*Proof.* In this proof, we assume that $\Delta > B_i$ since otherwise $\Delta \leq T_i$ (i.e., assuming that $\tau_i$ is schedulable, its BCRT must be no larger than $D_i \leq T_i$) and there cannot be any body job released by $\tau_i$. This would imply that $\Delta_i^C$ is by default equal to $\Delta$, thereby proving the claim for that case.

Thus, if $\Delta > B_i$, we note that $B_i \leq \Delta_i^C < B_i + T_i$ when computed with Eq. (3.15). Two cases must be considered.

**Case 1.** If $\Delta_i^C$ is shortened then at most one more body job can be added to the problem window $\Delta$ (remember that $\Delta_i^C < B_i + T_i$ and $B_i \leq T_i$ and each body job executes in a window of length $T_i$). Therefore, the interfering workload generated by $\tau_i$'s body jobs increases by at most $W_i$ (i.e., the workload of exactly one job). Moreover, because $\Delta_i^C$ is now $T_i$ time units shorter, one less job can execute in $\Delta_i^C$ and the interfering workload $\mathcal{W}_i^C(\Delta_i^C)$ generated by $\tau_i$'s carry-in and carry-out jobs must decrease by at least $W_i$ time units too. Hence, in total, the interfering workload $\mathcal{W}_i(\Delta)$ does not increase.

**Case 2.** The length of $\Delta_i^C$ is increased. Using Eq. (3.15), the computed value of $\Delta_i^C$ is $\Delta$ minus an integer multiple of $T_i$ and thus, when injecting Eq. (3.15) into Eq. (3.14), we get that $\left\lfloor \frac{\Delta - \Delta_i^C}{T_i} \right\rfloor = \frac{\Delta - \Delta_i^C}{T_i}$. By increasing $\Delta_i^C$ by a positive value $\varepsilon$, it thus holds that $\left\lfloor \frac{\Delta - (\Delta_i^C + \varepsilon)}{T_i} \right\rfloor < \left\lfloor \frac{\Delta - \Delta_i^C}{T_i} \right\rfloor$ for $\varepsilon > 0$. Therefore, at least one less body job can execute in the time window of length $\Delta$ and the interfering workload generated by $\tau_i$'s body jobs is decreased by at least $W_i$. Furthermore, since the carry-out job is already completely included in $\Delta_i^C$ (i.e., $\Delta_i^C \geq B_i$), in the best case increasing the length of $\Delta_i^C$ will allow us to fully integrate $\tau_i$'s carry-in job in $\mathcal{W}_i^C(\Delta_i^C)$. Hence, $\mathcal{W}_i^C(\Delta_i^C)$ may be increased by at most $W_i$ time units (the workload of $\tau_i$'s carry-in job). Summing all the contributions to the interfering workload $\mathcal{W}_i(\Delta)$, we have that $\mathcal{W}_i(\Delta)$ does not increase. □

The problem of computing $\mathcal{W}_i^C(\Delta_i^C)$ can be formulated as the maximization of $CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$ subject to $\Delta_i^C = x1 + x2$. The optimal solution of this optimization problem is an upper-bound on $\mathcal{W}_i^C(\Delta_i^C)$, whereas the final values of the decisions variables $x1$ and $x2$ correspond to $\Delta_i^{CI}$ and $\Delta_i^{CO}$, respectively. We solve this problem by using Algorithm 2 that is based on a technique named "sliding window" introduced in (Maia et al., 2014). It computes the maximum solution to the optimization problem defined above in linear time by checking all possible scenarios in which the problem window is aligned with any block of $\mathcal{WD}_i^{UCI}$ or $\mathcal{WD}_i^{UCO}$. Specifically, the scenarios tested can be divided into two groups: (i) the beginning of the problem window coincides with the start of a block in $\mathcal{WD}_i^{UCI}$ (lines 7 to 14); or (ii) the problem window ends at the completion of a block in $\mathcal{WD}_i^{UCO}$ (lines 15 to 22). Algorithm 2 also tries the configuration where the carry-out workload in the problem window is maximized (lines 1 to 3) and where the carry-in workload in is maximized (lines 4 to 6). It was proven in (Maia et al., 2014), that the maximum interfering workload is obtained in one of those scenarios.

By replacing the terms $\mathcal{W}_i(R_k)$ $(1 \leq i < k)$ with Eq. (3.14) in Eq. (3.4), a schedulability condition for task $\tau_k$ is stated in the next theorem.

**Algorithm 2:** Computing $\mathcal{W}_i^C$ for constrained deadline tasks.

**Input** : $\Delta_i^C$, $\mathcal{WD}_i^{UCI}$, $\mathcal{WD}_i^{UCO}$.
**Output**: $\mathcal{W}_i^C$ - Upper-bound on the workload of both the carry-in and carry-out jobs.

```
/* We maximize the carry-out workload inside the problem window */
```
$x2 \leftarrow \min\{\Delta_i^C,\ B_i\}$;
$x1 \leftarrow \Delta_i^C - x2$;
$\mathcal{W}_i^C \leftarrow CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$;

```
/* We maximize the carry-in workload inside the problem window */
```
$x1 \leftarrow \min\{\Delta_i^C,\ B_i + (T_i - R_i)\}$;
$x2 \leftarrow \Delta_i^C - x1$;
$\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;

```
/* We align the start of the problem window with the boundaries of every block
   in WD_i^UCI */
```
$x1 \leftarrow T_i - R_i$;
**foreach** $(w_b, h_b) \in \mathcal{WD}_i^{UCI}$ *in reverse order* **do**
    $x1 \leftarrow x1 + w_b$;
    $x2 \leftarrow \Delta_i^C - x1$;
    **if** $x2 \geq 0$ **then**
        $\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;
    **end**
**end**

```
/* We align the end of the problem window with the boundaries of every block
   in WD_i^UCO */
```
$x2 \leftarrow 0$;
**foreach** $(w_b, h_b) \in \mathcal{WD}_i^{RCO}$ *in order of appearance* **do**
    $x2 \leftarrow x2 + w_b$;
    $x1 \leftarrow \Delta_i^C - x2$;
    **if** $x1 \geq 0$ **then**
        $\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;
    **end**
**end**
**return** $\mathcal{W}_i^C$;

**Theorem 5.** *A task $\tau_k$ is schedulable under GFP iff $R_k \leq D_k$, where $R_k$ is the smallest $\Delta > 0$ to satisfy $\Delta = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m}\sum_{\forall i<k} \mathcal{W}_i(\Delta)$.*

The task set is declared schedulable if all tasks are schedulable. This can be checked by applying Theorem 5 to each task $\tau_i \in \tau$, starting from the highest priority task (i.e., $\tau_1$) and proceeding in decreasing order of priority.

## 3.7 Schedulability analysis for arbitrary deadline tasks

In the previous section, we presented a RTA for the special case where all tasks have constrained deadlines. In this section, we treat the general case where tasks may have arbitrary deadlines.

The difficulty with arbitrary deadline tasks is twofold:

1. Let $J_k$ be the job of $\tau_k$ for which we compute the WCRT and assume that $J_k$ is released at time $r_k$. Since it may be that $D_k > T_k$, more than one job of $\tau_k$ may execute in the problem window $[r_k,\ r_k + \Delta)$. That is, jobs of $\tau_k$ released before $r_k$ (i.e., at time $t \leq r_k - T_k$) may not have completed their execution at $r_k$ and yet $\tau_k$ may still be schedulable (i.e., it completes all jobs before their deadlines). Therefore, Eq. (3.4) that computes the WCRT of $\tau_k$ must be updated to integrate the residual workload of jobs of $\tau_k$ released before $r_k$ but interfering with $J_k$'s execution.

2. The second difficulty is that higher priority tasks may have more than one carry-in job. Specifically, if $D_i > T_i$, more than one job of $\tau_i$ may be released before $r_k$ and have a deadline after $r_k$. This property, which is formally proven in Lemma 8 in Section 3.7.3, requires to derive a new bound on the carry-in wokload released by each higher priority task interfering with $\tau_k$.

We address the first issue in Section 3.7.1 and the second in Section 3.7.3.

### 3.7.1 Response time analysis

In this subsection, we update Eq. (3.4) and derive a new bound on the WCRT of a task $\tau_k$. We integrate the fact that, for arbitrary deadline tasks, a job $J_{k,l}$ of task $\tau_k$ may be released before the completion of its preceding job $J_{k,l-1}$. Indeed, let us assume that $J_{k,l-1}$ and $J_{k,l}$ were released at time $r_{k,l-1}$ and $r_{k,l}$, respectively. In the worst-case scenario we have that $r_{k,l} = r_{k,l-1} + T_k$ and $J_{k,l-1}$ may complete its execution at any time smaller than or equal to $(r_{k,l-1} + D_k)$. Therefore, if $D_k > T_k$, job $J_{k,l-1}$ may not have completed its execution when $J_{k,l}$ is released. In such situation, we assume that $J_{k,l}$ does not start executing before the completion of $J_{k,l-1}$[4]. Hence the earliest instant at which $J_{k,l}$ may start executing is not its release time $r_{k,l}$ anymore, but the maximum between its release time and the completion time of $J_{k,l-1}$.

We now consider the two cases mentioned above:

1. if job $J_{k,l}$ can start executing as soon as it is released (i.e., at $r_{k,l}$), then the previous job $J_{k,l-1}$ of $\tau_k$ has already completed by time $r_{k,l}$. In such case, the situation is identical, with respect to $J_{k,l}$, to the worst-case scenario considered for constrained deadline tasks. That is, there is no additional interference by previous jobs of $\tau_k$ and the WCRT of $J_{k,l}$ is therefore obtained using Eq. (3.4) and maximizing the higher priority task interference. This scenario is encountered for the first job released by $\tau_k$. Let $X_{k,1}$ be the completion time of that job. Without any loss of generality we can assume that that job was released at time 0. Hence we have $r_{k,1} = 0$ and, using Eq. (3.4), $X_{k,1}$ is upper-bounded by the smallest positive solution to

$$X_{k,1} = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m}\sum_{\forall i<k} \mathcal{W}_i(X_{k,1}) \tag{3.17}$$

[4]We enforce this execution behavior to avoid data inconsistencies between successive jobs of a same task. Indeed, a job may require the computation results of its preceding job to be able to proceed correctly.

2. if job $J_{k,l-1}$ is not yet completed when $J_{k,l}$ is released, then $J_{k,l}$ cannot start executing before the completion of $J_{k,l-1}$. Therefore, the worst-case scenario for $J_{k,l}$ happens when the overlap between the execution window of $J_{k,l-1}$ and the active window of $J_{k,l}$ is maximized. This happens when $J_{k,l-1}$ completes as late as possible and $J_{k,l}$ is released as early as possible. Assume that $X_{k,l-1}$ and $X_{k,l}$ are the worst-case completion times of $J_{k,l-1}$ and $J_{k,l}$, respectively. The WCRT of $J_{k,l}$ is then given by

$$\begin{aligned} R_{k,l} &= X_{k,l} - r_{k,l} \\ &= X_{k,l} - (l-1) \times T_k \end{aligned} \tag{3.18}$$

where

$$X_{k,l} = X_{k,l-1} + L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m} \sum_{\forall i<k} \Big( \mathcal{W}_i(X_{k,l}) - \mathcal{W}_i(X_{k,l-1}) \Big) \tag{3.19}$$

Eq. (3.19) is composed of four terms detailed hereafter.

- $X_{k,l-1}$ is the worst-case completion time of the preceding job $J_{k,l-1}$, i.e., the earliest time at which $J_{k,l}$ may start executing;
- $L_k$ is the minimum amount of time required by $J_{k,l}$ to complete its execution when it executes for its WCET and does not suffer any interference;
- $\frac{1}{m}(W_k - L_k)$ is an upper-bound on $J_{k,l}$'s self-interference (as proven in (Melani et al., 2017));
- $\frac{1}{m}\sum_{\forall i<k}(\mathcal{W}_i(X_{k,l}) - \mathcal{W}_i(X_{k,l-1}))$ is the maximum interfering workload that can be released by higher priority tasks in the problem window of length $X_{k,l}$ that has not yet been accounted for in the term $X_{k,l-1}$, i.e., the worst-case completion time of $J_{k,l-1}$.

The WCRT of a task $\tau_k$ is thus given by its job with the largest response time. Formally,

$$R_k = \max_{l>0} \{X_{k,l} - (l-1) \times T_k\} \tag{3.20}$$

where $X_{k,l}$ is the worst-case completion time of the $l^{\text{th}}$ job released by $\tau_k$ in the problem window. Combining Eq. (3.17) and (3.19) we get that

$$X_{k,l} = l \times (L_k + \frac{1}{m}(W_k - L_k)) + \frac{1}{m} \sum_{\forall i<k} \mathcal{W}_i(X_{k,l}) \tag{3.21}$$

Note that we can stop iterating over $l$ when

- we reach the first $l > 0$ such that $X_{k,l} \leq (l \times T_k)$, i.e., the first job of $\tau_k$ released in the problem window that completes before the release of the next job of $\tau_k$;
- we reach the first $l > 0$ such that $X_{k,l} > (l-1) \times T_k + D_k$, i.e., at the first job of $\tau_k$ released in the problem window that has a response time larger than its deadline.

In the first case the task $\tau_k$ is schedulable while in the second it is not. One of these two termination conditions holds eventually in most cases. However, it cannot be guaranteed that Eq. (3.20) always terminates in the general case, as it has already been shown for sequential tasks (Guan et al., 2009). Such rather special corners cases have not been detected at all during our experimental evaluation. Nonetheless, one can simply define a threshold for the values of $l$. Whenever the threshold is reached, the procedure terminates and the task $\tau_k$ is declared unschedulable. Note that this may decrease the effectiveness of the response time analysis.

The term $\mathcal{W}_i(X_{k,l})$ in Eq. (3.21) is computed using Eq. (3.14). Eq. (3.14) uses an upper-bound $\mathcal{W}_i^C$ on the carry-in and carry-out workload that can be released by higher priority task $\tau_i$. As discussed at the beginning of this section, each higher priority task may execute more than one carry-in job in the problem window and a new bound on $\mathcal{W}_i^C$ must be derived. We present this bound in the next subsections.

### 3.7.2 Carry-out workload

As defined in Section 3.2, a carry-out job is a job that is released in the problem window less than $T_i$ time units before the end of that window. Hence the carry-out job of $\tau_i$ is the last job that can be released by $\tau_i$ in the problem window (remember that job releases are at least $T_i$ time units apart). Therefore, each higher priority task $\tau_i$ can release at most one carry-out job, even when $\tau_i$ has an arbitrary deadline. It results that the upper-bound on the carry-out workload proven in Theorem 4 is still valid for arbitrary deadline tasks.

### 3.7.3 Carry-in workload

As mentioned in Section 3.2, a carry-in job is defined as a job released before the start of the problem window and with a deadline after the problem window start. When a higher priority task $\tau_i$ has a deadline smaller than or equal to its minimum inter-arrival time (i.e., $D_i \leq T_i$), at most one such carry-in job may exist. However, this result does not hold for tasks with arbitrary deadlines. Indeed, it may happen that $D_i > T_i$, in which case a job of $\tau_i$ may have its deadline after the release of one (or several) other job(s) of $\tau_i$. Yet, the number of carry-in jobs may still be upper-bounded as proven in Lemma 8.

**Lemma 8.** *Each higher priority task $\tau_i$ with an arbitrary deadline has at most $\lceil \frac{D_i}{T_i} \rceil$ carry-in jobs.*

*Proof.* Let $J_i$ be the earliest carry-in job released by $\tau_i$. Let $r_i$ be its release time and $d_i$ its absolute deadline. By definition of $J_i$, all jobs released before $r_i$ are not carry-in jobs. Let $c = \lceil \frac{D_i}{T_i} \rceil$. Let $J_{i+c}$ be any job of $\tau_i$ released at or later than $(r_i + c \times T_i)$. Then, $J_{i+c}$ is released at or after $d_i$ (because $d_i = r_i + D_i \leq r_i + \lceil \frac{D_i}{T_i} \rceil \times T_i = r_i + c \times T_i$). Since $J_i$ is a carry-in job, $d_i$ is necessarily after the problem window start. Hence any job $J_{i+c}$ is released after the problem window start and is not a carry-in job. Since at most $c-1$ jobs of $\tau_i$ can be released between $r_i$ and $r_i + c \times T_i$, we conclude that there are at most $c-1$ other jobs than $J_i$ that may be carry-in jobs. This proves the claim. □

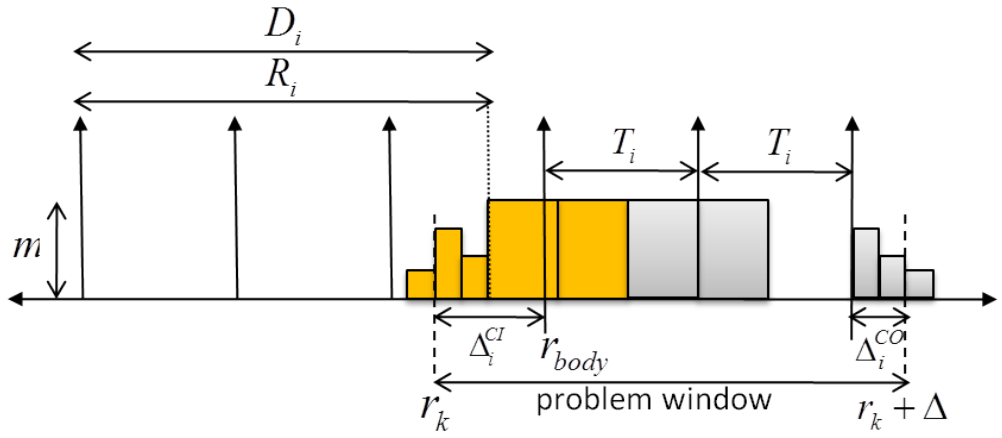



Figure 3.9: Worst-case interfering workload released by $\tau_i$ in $\tau_k$'s problem window when $D_i > T_i$. Yellow jobs are carry-in jobs.

Note that Lemma 8 covers the case of constrained deadline tasks too since $\lceil \frac{D_i}{T_i} \rceil = 1$ in that particular case.

**Example 9.** *Consider the worst-case interfering scenario of task $\tau_i$ depicted in Fig. 3.9. We have that $D_i = 2.6 \times T_i$. Hence three jobs may be released by $\tau_i$ before $r_k$ and have their deadline after $r_k$. Further, because in this example $R_i = D_i$, the three carry-in jobs (in yellow in the picture) execute at least partially in the problem window starting at time $r_k$.*

Since there might be more than one carry-in job released by $\tau_i$, we must update the definition of $\Delta_i^{CI}$ (Eq. (3.5)) and the upper-bound on the worst-case carry-in interfering workload (Eq. (3.10)).

As depicted in Fig. 3.2 for constrained deadline tasks and in Fig. 3.9 for arbitrary deadline tasks, we define the carry-in window of $\tau_i$ as the interval starting at the beginning of the problem window (i.e., at time $r_k$) and ending at the earliest release of a *body job* of $\tau_i$. Therefore, if $r_{body}$ is the release time of that job, we have that

$$\Delta_i^{CI} \stackrel{\text{def}}{=} r_{body} - r_k \tag{3.22}$$

By Lemma 8, we know that there are at most $\lceil \frac{D_i}{T_i} \rceil$ carry-in jobs released before $r_{body}$. Therefore, the $j^{\text{th}}$ carry-in job of $\tau_i$ (with $1 \leq j \leq \lceil \frac{D_i}{T_i} \rceil$) cannot be released later than time

$$r_{i,j} \stackrel{\text{def}}{=} r_{body} - j \times T_i \tag{3.23}$$

$$= r_k + \Delta_i^{CI} - j \times T_i \tag{3.24}$$

Similar to the constrained deadline case, the carry-in workload generated by $\tau_i$ would be maximized if each carry-in job of $\tau_i$ is released as late as possible and executes as much workload as possible in the problem window. Now, let $R_i$ be the upper-bound on the worst-case response time of $\tau_i$ computed with Eq. (3.21). Lemma 9 (see below) proves that aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $(r_{i,j} + R_i)$ and calculating the part of $\mathcal{WD}_i^{UCI}$'s workload released after

$r_k$ (using Eq. (3.10)) provides an upper-bound on the maximum interfering workload that can be generated by the $j^{\text{th}}$ carry-in job of $\tau_i$. Formally, we have that the interfering workload executed by the $j^{\text{th}}$ carry-in job of $\tau_i$ in the problem window is upper-bounded by

$$CI_{i,j}(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[\Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p\right]_0^{w_b} \tag{3.25}$$

This is stated in Lemma 9 below.

**Lemma 9.** *Let $R_i$ be the upper-bound on the worst-case response time of $\tau_i$ **computed by Eq.** (3.21). Aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $(r_{i,j} + R_i)$ gives an upper-bound on the maximum interfering workload that can be generated by $\tau_i$'s carry-in job released at $r_{i,j}$ in the carry-in window, independently of the interference imposed on $\tau_i$.*

*Proof.* Since Eq. (3.4) and Eq. (3.21) both compute the WCRT of a task based on the following algorithm (i) summing all the self interfering workload and all the workload released by higher priority tasks in the problem window, (ii) dividing it by the number of cores $m$, and (iii) adding the result to $\tau_k$'s critical path length, the proof of this lemma is identical in every word to the proof of Lemma 4, replacing Eq. (3.4) with Eq. (3.21). □

Since there are up to $\lceil \frac{D_i}{T_i} \rceil$ carry-in job, we have that the maximum interfering carry-in workload generated by $\tau_i$ is given by the sum of the interfering workload generated by each of its carry-in jobs. That is,

$$CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{j=1}^{\lceil \frac{D_i}{T_i} \rceil} \left( \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[\Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p\right]_0^{w_b} \right) \tag{3.26}$$

Note that the actual implementation of Eq. (3.26) can be drastically simplified using two simple mathematical facts on Eq. (3.26):

1. for each carry-in job $j$ such that $(\Delta_i^{CI} - j \times T_i + R_i - L_i) \geq 0$, the contribution of the inner-sum to the carry-in workload will always be $W_i$;

2. for each carry-in job such that $(\Delta_i^{CI} - j \times T_i + R_i) \leq 0$, the contribution of the inner-sum to the carry-in workload will always be 0.

This means that there is at most one carry-in job and therefore only one $j$ for which the summation on $b$ needs to be done. For all the other $\lceil \frac{D_i}{T_i} \rceil - 1$ carry-in jobs, the interfering workload can readily be considered to be equal to $W_i$ or 0 depending on whether $(\Delta_i^{CI} - j \times T_i + R_i - L_i) \geq 0$ or $(\Delta_i^{CI} - j \times T_i + R_i) \leq 0$, respectively.

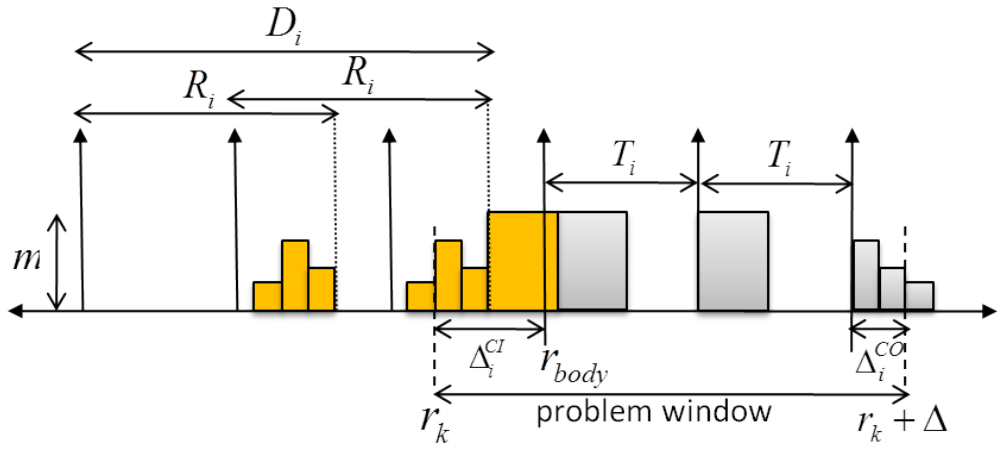



Figure 3.10: Worst-case interfering workload released by $\tau_i$ in $\tau_k$'s problem window when $D_i > T_i$ but $R_i < D_i$. Yellow jobs are carry-in jobs.

**Example 10.** *Consider the example in Fig. 3.10 where $D_i = 2.6 \times T_i$. As in Example 9, task $\tau_i$ releases three carry-in jobs. However, because the WCRT $R_i$ of $\tau_i$ is smaller than $D_i$, the carry-in job released at $(r_{body} - 3 \times T_i)$ completes no later than time $(r_{body} - 3 \times T_i + R_i)$ which is before the start of the problem window (i.e., time $r_k$). Therefore, we have that $(\Delta_i^{CI} - 3 \times T_i + R_i) < 0$ and the contribution of that carry-in job to the interfering workload is 0. On the other hand, the carry-in job released at time $(r_{body} - T_i)$ respects the inequality $(\Delta_i^{CI} - T_i + R_i - L_i) \geq 0$ since it starts and completes after the beginning of the problem window. Therefore, its contribution to the interfering workload is equal to its total workload $W_i$. For the carry-in job released at time $(r_{body} - 2 \times T_i)$, none of the two conditions holds. Hence its execution overlaps with the beginning of the problem window and its contribution to the interfering workload is a portion of its workload distribution $\mathcal{WD}_i^{UCI}$.*

**Theorem 6.** *The interfering workload $W_i^{CI}$ generated by the carry-in jobs of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CI}$ is upper-bounded by Eq. (3.26).*

*Proof.* It directly follows from the combination of Lemmas 8 and 9. □

Similar to the constrained deadline case covered in Section 3.4.3, an improve bound on the carry-in workload can be derived using Lemma 10 proven below.

**Lemma 10.** *An upper-bound on the maximum interfering workload that can be generated by a carry-in job of task $\tau_i$ released at time $r_{i,j}$ in a carry-in window of length $\Delta_i^{CI}$ is given by* $\max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\} \times m$.

*Proof.* Since no job of $\tau_i$ can complete later than $R_i$ time units after its release, we know that the carry-in job released at $r_{i,j}$ completes no later than $r_{i,j} + R_i = r_k + \Delta_i^{CI} - j \times T_i + R_i$ (using Eq. (3.24)). Therefore, the carry-in job executes during at most $\max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\}$ time units on $m$ processors within the carry-in window $[r_k,\ r_k + \Delta_i^{CI})$, hence the claim. □

Combining Theorem 6 with Lemma 10, we derive an improved bound on the carry-in workload of an interfering task $\tau_i$ with arbitrary deadline.

**Theorem 7.** *The interfering workload $W_i^{CI}$ generated by the carry-in jobs of a higher priority task $\tau_i$ in a window of length $\Delta_i^{CI}$ is upper-bounded by*

$$\sum_{j=1}^{\lceil \frac{D_i}{T_i} \rceil} \min \left\{ \max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\} \times m, \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[ \Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right]_0^{w_b} \right\} \tag{3.27}$$

*Proof.* Follows from Theorem 6 and Lemma 10. □

### 3.7.4 Upper-bounding the carry-in and carry-out interference

In the previous subsections, we have upper-bounded the carry-in and carry-out interference that a higher priority task $\tau_i$ can generate in windows of length $\Delta_i^{CI}$ and $\Delta_i^{CO}$, respectively. However, as already discussed in Section 3.6 for the constrained deadline case, the difficulty is to identify the lengths of $\Delta_i^{CI}$ and $\Delta_i^{CO}$ that maximize the total interference generated by $\tau_i$. For constrained deadline tasks, this optimization problem was solved using Algorithm 2. In this section, we adapt Algorithm 2 to support systems composed of arbitrary deadline tasks. The result is presented in Algorithm 3.

Like for the constrained deadline case, Algorithm 3 uses the sliding window technique to maximize the interfering workload released by a task $\tau_i$ in the problem window. First, the distance $\Delta_i^C$, which by definition is equal to $\Delta_i^{CI} + \Delta_i^{CO}$, is computed using Eq. (3.15) (note that the proof of Lemma 7 is still valid for arbitrary deadline tasks). Then, Algorithm 3 is called.

Algorithm 3 is identical to Algorithm 2 for lines 1 to 3 and lines 17 to 24, which are related to the carry-out workload. However, as it was to be expected, Algorithm 3 differs from Algorithm 2 for parts that are related to the carry-in workload (lines 4 to 16).

Algorithm 3 first tries to maximize the carry-out workload released by $\tau_i$ in the problem window (lines 1 to 3). To this end, it aligns the end of the problem window with the earliest time at which $\tau_i$'s carry-out job may complete (i.e., setting $\Delta_i^{CO}$ to $B_i$), or by setting $\Delta_i^{CO}$ to $\Delta_i^C$ if $\Delta_i^C$ is smaller than the BCRT of $\tau_i$. Then, Algorithm 3 similarly tries to maximize the carry-in workload released by $\tau_i$ in the problem window (lines 4 to 6). This is achieved by aligning the beginning of the problem window with the latest time at which the earliest carry-in job of $\tau_i$ may start executing. Hence we set $\Delta_i^{CI}$ to $(B_i + \lceil \frac{D_i}{T_i} \rceil \times T_i - R_i)$, where $(\lceil \frac{D_i}{T_i} \rceil \times T_i - R_i)$ is the smallest possible distance between the completion of the earliest carry-in job of $\tau_i$ and the release of its first body job at $r_{body}$. The length $(B_i + \lceil \frac{D_i}{T_i} \rceil \times T_i - R_i)$ is thus the smallest possible distance between the time at which the earliest carry-in job of $\tau_i$ starts executing and $r_{body}$. Line 4 also ensures that $\Delta_i^{CI}$ cannot be larger than $\Delta_i^C$.

**Algorithm 3:** Computing $\mathcal{W}_i^C$ for arbitrary deadline tasks.

**Input** : $\Delta_i^C$, $\mathcal{WD}_i^{UCI}$, $\mathcal{WD}_i^{UCO}$.
**Output**: $\mathcal{W}_i^C$ - Upper-bound on the workload of both the carry-in and carry-out jobs.

```
   /* We maximize the carry-out workload inside the problem window            */
1  x2 ← min{Δ_i^C, B_i};
2  x1 ← Δ_i^C − x2;
3  W_i^C ← CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2);

   /* We maximize the carry-in workload inside the problem window             */
4  x1 ← min{Δ_i^C, B_i + ⌈D_i/T_i⌉ × T_i − R_i};
5  x2 ← Δ_i^C − x1;
6  W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};

   /* We align the start of the problem window with the boundaries of every block
      in WD_i^UCI for every carry-in job of τ_i                                */
7  forall the j = 1 to ⌈D_i/T_i⌉ do
8      x1 ← j × T_i − R_i;
9      foreach (w_b, h_b) ∈ WD_i^UCI in reverse order do
10         x1 ← x1 + w_b;
11         x2 ← Δ_i^C − x1;
12         if x1 ≥ 0 and x2 ≥ 0 then
13             W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};
14         end
15     end
16 end

   /* We align the end of the problem window with the boundaries of every block
      in WD_i^UCO                                                             */
17 x2 ← 0;
18 foreach (w_b, h_b) ∈ WD_i^RCO in order of appearance do
19     x2 ← x2 + w_b;
20     x1 ← Δ_i^C − x2;
21     if x1 ≥ 0 then
22         W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};
23     end
24 end
25 return W_i^C;
```

Lines 6 to 16 iterate over the $\lceil \frac{D_i}{T_i} \rceil$ carry-in jobs released by $\tau_i$. For each carry-in job it computes the latest time at which that job may complete (line 8) and then aligns the beginning of the problem window with the start of every block in the workload distribution $\mathcal{WD}_i^{UCI}$ of that job (lines 10 to 14).

Lines 17 to 24 are identical to Algorithm 2 and align the end of the problem window with the end of every block in the workload distribution $\mathcal{WD}_i^{UCO}$ of $\tau_i$'s carry-out job.

The maximum interfering workload released by carry-in and carry-out jobs of $\tau_i$ is the maximum over the interfering workload computed for each of the scenarios described above (as already discussed in (Maia et al., 2014)).

## 3.8 Experimental evaluation

The analysis presented in this chapter has been implemented within the MATLAB framework released by the authors of (Melani et al., 2015). We follow the same technique as in (He and Yesha, 1987) and (Melani et al., 2015) to generate random task sets composed of DAG tasks.

Each DAG in the task set is initially a composition of two NFJ-DAGs connected in series. The NFJ-DAGs are constructed by recursively expanding their nodes. Each node has a probability $p_{par}$ to fork and a probability $p_{term}$ to join, where $p_{term} + p_{par} = 1$. Each parallel branch has a maximum *depth* that limits the number of nested forks. Additionally, the number of parallel branches leaving from a fork node is randomly chosen within a uniform distribution bounded by $[2, n_{par}]$. Finally, a general DAG is obtained by randomly adding directed edges between pairs of nodes, granted that such randomly-placed precedence constraints do not violate the "acyclic" semantics of the DAG. The probability of adding an edge between two nodes is given by $p_{add}$, with the restriction that any two nodes with a common fork-node as direct predecessor cannot be connected. This last restriction avoids generating degenerated DAGs that behave as sequential tasks.

Once the DAG $G_i$ of a task $\tau_i$ is constructed, the task parameters are assigned as follows. The WCET $C_j$ of a subtask $v_j \in V_i$ is uniformly chosen in the interval $[1, 100]$. The task length $L_i$, the workload $W_i$ and the maximum makespan $M_i$ (see Eq. 3.2) of $\tau_i$ are computed based on the internal structure of the DAG and the WCET of its nodes. The minimum inter-arrival time $T_i$ is uniformly chosen in the interval $[M_i, W_i/\beta]$, where the parameter $\beta$ is used to define the minimum utilization of all the tasks. Therefore, the task utilization becomes uniformly distributed over $[\beta, W_i/M_i]$. For all experiments that have a varying total utilization $U_{tot}$ (i.e., Fig. 3.11 and 3.16), we keep generating and adding new tasks to the task set until the target total utilization $U_{tot}$ is met. $U_{tot}$ is achieved exactly by adjusting the period of the last task added to the system. Otherwise, for all other experiments, we use UUnifast (Bini and Buttazzo, 2005) to derive individual task utilizations (and consequently their period) for a fixed value of $n$. Priorities are assigned following the DM policy.

For each tested system configuration, we generated and assessed the schedulability of 500 task sets. Unless stated otherwise, in all experiments reported herein, we have set $p_{par} = 0.8$, $p_{term} = 0.2$, $depth = 2$, $n_{par} = 5$, $p_{add} = 0.2$, $\beta = 0.035 \times m$, $U_{tot} = 0.7m$, $n = 1.5m$ and $m = 8$. These settings lead to a rich variety of internal DAG structures, some of which resemble real-world applications as noted in (Melani et al., 2017): we observed both heavy and unbalanced workloads with different degrees of parallelism and sequential segments in each task set. The maximum parallelism of a DAG (i.e., the number of subtasks that can execute in parallel) with such configuration is 25.

### 3.8.1 Evaluation for constrained deadlines

We compare our response time analysis for DAG tasks with constrained deadlines (referred to as IRTA-FP) to the schedulability analysis described in (Melani et al., 2015) (referred to as Mel-

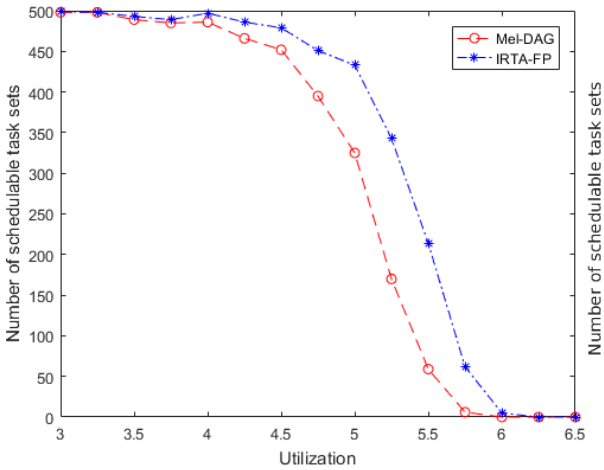



Figure 3.11: IRTA-FP varying $U_{tot}$.

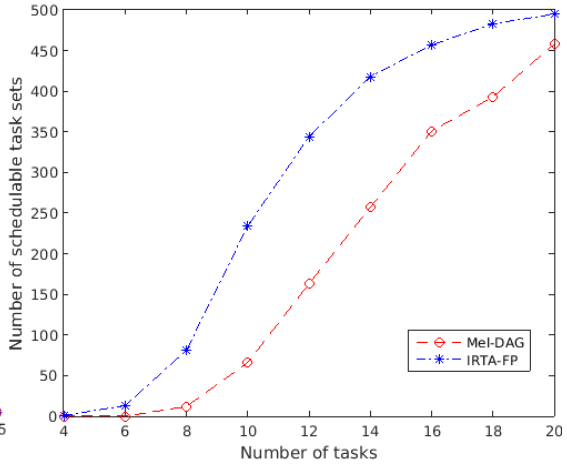



Figure 3.12: IRTA-FP varying $n$.

DAG) for GFP scheduling. In an attempt to maximize the schedulability ratios of these tests, we restrict our attentions to the case where the relative deadline $D_i$ is set equal to the period $T_i$.

In the first set of experiments, the system utilization $U_{tot}$ was varied in $(0,m]$ by steps of 0.25. Fig. 3.11 shows the number of schedulable task sets when $m = 8$. For both low and very high utilization (i.e., when all or none of the task sets are schedulable), IRTA-FP and Mel-DAG are indistinguishable. However, for $U_{tot} \in [4,6]$, IRTA-FP performs substantially better. In particular, when $U_{tot} = 5.25$, IRTA-FP schedules 341 task sets against 156 for Mel-DAG. Instead, Fig. 3.12 reports the schedulability as a function of the number of tasks $n$, with $n$ ranging from 4 to 20. The values of $U_{tot}$ and $m$ were kept constant and equal to $0.7m$ and 8, respectively. IRTA-FP outperforms Mel-DAG for any value of $n$ with an average gain of approximately 20%, although both tests converge to full schedulability for larger $n$. Intuitively, it is easier to schedule many light tasks than few heavy tasks.

We then study the impact of the DAG structures on the outcome of the two schedulability tests.

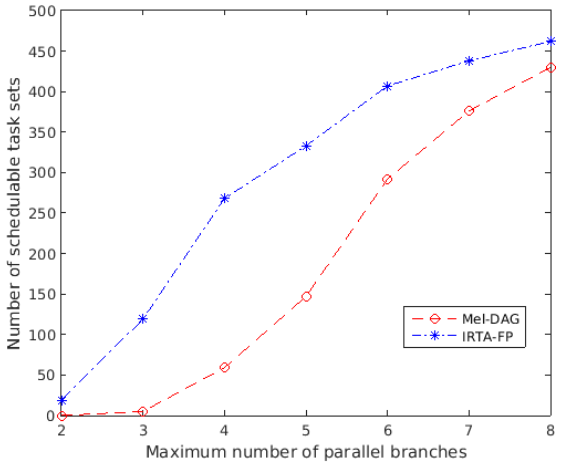



Figure 3.13: IRTA-FP varying $n_{par}$.

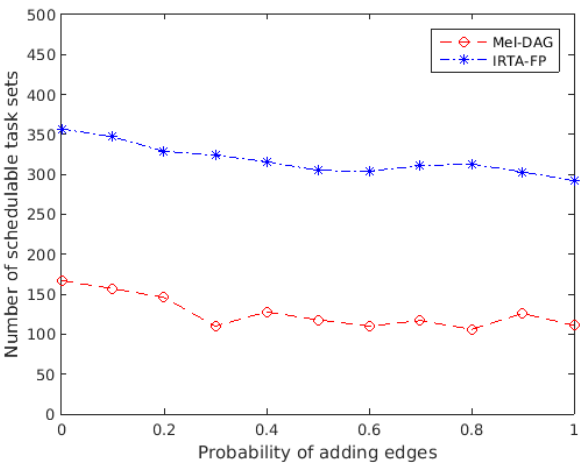



Figure 3.14: IRTA-FP varying $p_{add}$.

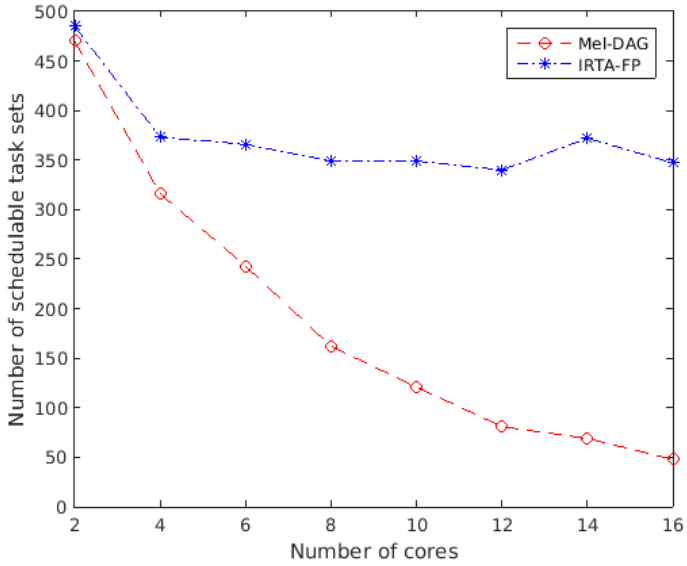



Figure 3.15: IRTA-FP varying $m$.

A trend similarly to that of Fig. 3.12 can be observed in Fig. 3.13, where we varied the maximum number of parallel branches $n_{par}$ in the interval $[2,8]$. Mel-DAG has clear limitations when the average parallelism of the DAGs is up to half of the platform's parallelism (i.e., $n_{par} \leq 4$) and only admits a large share of tasks sets for $n_{par} \geq 6$. On the other hand, IRTA-FP accepts at least 50% of the task sets for $n_{par} \geq 4$ even though the schedulability ratio reduces when the tasks become nearly sequential (i.e., $n_{par}$ becomes close to 2). As expected, both approaches are comparable when the task parallelism is consistently greater than $m$. Fig. 3.14 reports the results obtained for different types of DAGs, as the probability of adding edges $p_{add}$ between two nodes is increased from 0 to 1 by steps of 0.1. To clarify, $p_{add} = 0$ corresponds to generating NFJ-DAGs, while $p_{add} = 1$ leads to synchronous parallel tasks. In between we have arbitrary DAGs. IRTA-FP attains a solid 40% schedulability improvement over Mel-DAG for any value of $p_{add}$. Interestingly, such gain is not maximized when IRTA-FP benefits from a more accurate characterization of the carry-out workload (i.e., in the case of NFJ-DAGs). This stresses the importance of exploring the precedence constraints within a DAG when deriving bounds on the interfering workload. Furthermore, we remark that IRTA-FP could achieve better results had we transformed the final DAGs into NFJ instead of considering the original ones. In conjunction with an average increase in the individual critical path lengths, this also justifies the slow degradation when increasing $p_{add}$.

In Fig. 3.15, we illustrate how IRTA-FP performs when $m$ varies according to the sequence $[2,4,6,8,10,12,14,16]$, with $U_{tot}$ and $n$ scaling with $m$. Mel-DAG degrades for higher values of $m$, while IRTA-FP maintains a schedulabity ratio around 72%. Such improvement is due to the characterization of the carry-in and carry-out workload distribution. IRTA-FP exploits the internal structure of the DAGs to bound the parallelism of such jobs, hence limiting the number of cores on which they execute for larger $m$; whereas Mel-DAG assumes that all interfering jobs always use the $m$ cores.

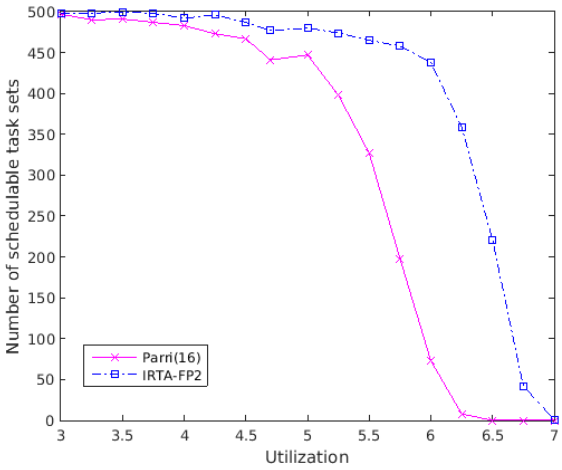



Figure 3.16: IRTA-FP2 varying $U_{tot}$.

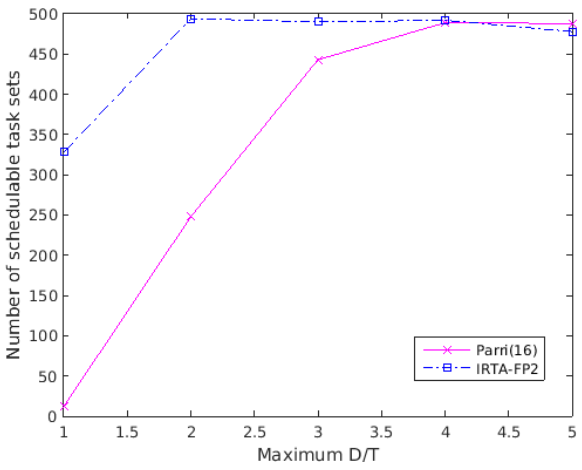



Figure 3.17: IRTA-FP2 varying $\alpha_{max}$.

### 3.8.2 Evaluation for arbitrary deadlines

We now compare the performance of our response time analysis for DAG tasks with arbitrary deadlines (referred to as IRTA-FP2) to the schedulability test proposed by Parri et al. (Parri et al., 2015) for GDM scheduling (referred to as Parri(16)), which was shown to outperform the tests in (Bonifaci et al., 2013), and hence, as far as we know, the only competitor to our test for arbitrary deadline DAG tasks. The number 16 added to Parri's test name denotes the maximum number of iterations allowed for the convergence of the outer loop in their RTA, which in most cases is sufficient to satisfy the convergence of the analysis, as suggested by the authors. Furthermore, since the analysis in (Parri et al., 2015) assumes that multiple jobs of the same DAG tasks may execute in parallel (instead of a job becoming ready only after the previous one completes its execution, as we do), for the sake of fairness, we enforce that no task is assigned with a period smaller than its maximum makespan. That is, $T_i \geq M_i$, $\forall \tau_i \in \tau$. By default, the deadline $D_i$ is uniformly selected in the interval $[T_i, \alpha_{max} T_i]$, with $\alpha_{max} = 3$ controlling the maximum ratio of $D_i/T_i$; meaning that $T_i \leq D_i \leq 3T_i$.

Fig. 3.16 reports the number of schedulable tasks sets as a function of the total utilization $U_{tot}$ for $m = 8$. While IRTA-FP2 stops deeming any task set schedulable at $U_{tot} = 7$, for Parri(16) such drop happens 10% earlier. Notably, when $U_{tot} \in [5.25, 6.75]$, IRTA-FP2 greatly outperforms Parri(16), with a schedulability gain peaking at 75%. This suggests that the way we handle the multiple interfering jobs carried-in by the higher priority tasks largely compensates the handicap on the self-interference component due to the different run-time assumptions.

In order to study the effectiveness of both approaches for different values of $D_i$, we varied $\alpha_{max}$ in the range $[1, 5]$. The results are depicted in Fig. 3.17 for constant values of $U_{tot}$, $n$ and $m$. In the case of implicit deadlines (i.e., $\alpha_{max} = 1 \implies D_i = T_i$), Parri(16) performs very poorly, confirming the author's observation that their analysis is specifically tailored for arbitrary deadlines and as such is overly pessimistic for more restrictive models. On the other hand, IRTA-FP2 is able to schedule 328 task sets as it was already witnessed in the constrained deadline case studied above.

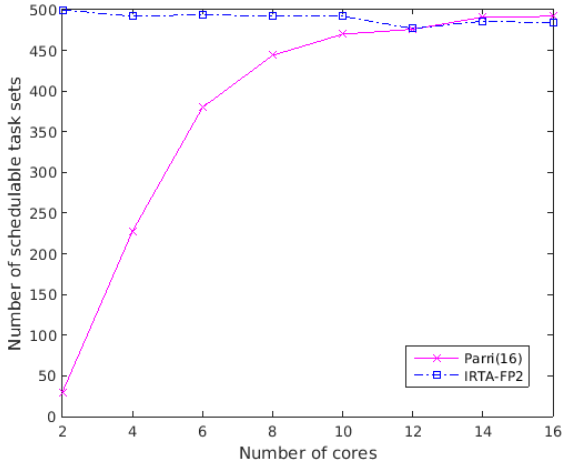



Figure 3.18: IRTA-FP2 varying $m$.

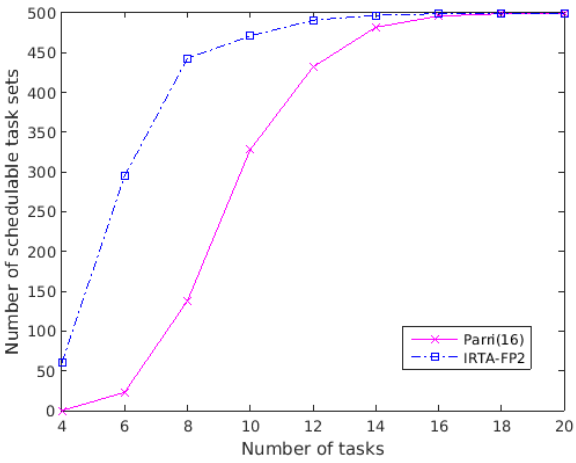



Figure 3.19: IRTA-FP2 varying $n$.

As $\alpha_{max}$ is increased, both tests rapidly achieve nearly full schedulablity. It is worth noting that larger values of $D_i$ strongly benefit Parri(16) since they assume that several jobs of the same task can execute in parallel, whereas in IRTA-FP2 assumes that a job cannot start executing before its preceding job has been completed.

In Fig. 3.18, we show the schedulability results as a function of the number of cores $m$. Both tests are robust to platforms with increased parallelism, although IRTA-FP2 succeedes in scheduling most task sets for any value of $m$, Parri(16) requires $m \geq 12$ to perform similarly. Finally, Fig. 3.19 illustrates how IRTA-FP2 performs when the number of tasks $n$ is varied according to the sequence $[2,4,6,8,10,12,14,16]$. IRTA-FP2 substantially outperforms Parri(16) when $n < 14$, with an average schedulability improvement close to 35%. Nevertheless, both approaches are indistinguishable when the amount of tasks is at least twice the number of cores. From these last sets of experiments, we can conclude that the workload distributions derived to characterize the carry-in and carry-out jobs are also effective for the analysis of DAG tasks with arbitrary deadlines.

## 3.9 Summary

In this chapter, we studied the sporadic DAG model under GFP scheduling. Motivated by the fact that a poor characterization of the higher priority interfering workload leads to pessimistic analysis of parallel task systems, we presented new techniques to model the worst-case carry-in and carry-out workload. These techniques exploit both the internal structure and worst-case execution patterns of the DAGs.

Following a sliding window strategy that leverages from such workload characterization, we then derived a schedulability analysis to compute an improved upper-bound on the WCRT of each DAG task. Experimental results not only attest the theoretical dominance of the proposed analysis over its state-of-the-art counterpart (in the constrained deadline case), but also showed that its effectiveness is independent of the number of cores and it substantially tightens the schedulability of DAG tasks on multiprocessor systems for both constrained and arbitrary deadline task sets.

# Chapter 4

# Schedulability Analyses for Partitioned Scheduling

In Chapter 3, we studied DAG tasks under a work-conserving scheduler, where every subtask may migrate among processors to guarantee that no processor is idle when there is backlogged workload to execute. Instead, this chapter considers the partitioned scheduling of a set of sporadic constrained deadline DAG tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, where each subtask $v_{i,j} \in V_i$ is statically assigned to a processor $P_{i,j}$, with $P_{i,j} \in [1, m]$. That is, subtasks are not allowed to migrate and thus they can run in parallel only with other independent subtasks mapped to different processors.

Next we present a response time analysis for FP scheduling that is not tied down to a specific subtask-to-core mapping strategy: the partitioning is assumed to be given. We show that a partitioned DAG task can be modeled as a set of self-suspending tasks. We then propose an algorithm to traverse a DAG and characterize such worst-case scheduling scenario. As a result, with minor modifications, any state-of-the-art uniprocessor RTA for sporadic self-suspending tasks can thus be used to analyze partitioned DAG tasks. We also present a second schedulability analysis, based on the concept of task-duplication, which is much simpler and results in reduced pessimism. More specifically, we developed a partitioning algorithm with the goal of minimizing the number of cores that guarantees feasibility and eliminating cross-core dependencies. Thanks to the duplication of key subtasks, all resulting partitions are independent of each other. Thus, the problem of scheduling a set of partitioned DAGs becomes equivalent to the problem of scheduling a set of sequential tasks on multiprocessors in a partitioned manner.

## 4.1 Motivation

As noted in (Altmeyer et al., 2015), on a multicore system there are strong inter-dependencies between timing and schedulability analysis, since the worst-case execution times are heavily dependent on the amount of cross-core interference generated on shared resources. This phenomenon is further amplified with parallel tasks due to the intense communication and concurrency between their sequential computational units. Due to its flexible execution environment, global schedul-

ing adds uncertainty and variability to the lower-level timing analysis, which then become overly pessimistic, spoiling most of the benefits of having a centralized scheduler. In this sense, it is our belief that partitioned scheduling is the most promising approach to support parallel tasks in hard real-time systems.

Partitioned scheduling is a well-studied topic in real-time distributed systems. Different response time analyses, priority assignment techniques and mapping heuristics that allow for task parallelism have been proposed in the past years (Tindell and Clark, 1994; Palencia et al., 1997; Palencia and Gonzalez Harbour, 1998; Palencia and Harbour, 1999). Although these works provide a strong understanding of the worst-case behavior of parallel tasks, they are inevitably less effective when applied to multicore systems due to the absence of the network component and local release jitters. Results concerning the schedulability of partitioned parallel tasks in multiprocessors are very limited. In (Axer et al., 2013), the authors presented a response time analysis for sporadic fork-join tasks with arbitrary deadlines under fixed-priority scheduling. Unfortunately, the paper was found flawed as shown in Appendix B. Therefore, to the best of our knowledge only the results transposed from distributed systems provide an answer to the problem of scheduling partitioned parallel tasks atop multiprocessors.

## 4.2 Additional definitions

Consider a set of $n$ sporadic DAG tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ with constrained deadlines to be scheduled by a partitioned scheme on a multiprocessor platform $\Pi$ composed of $m$ identical cores. First and foremost, we study FP scheduling as it was the case in the previous chapter for global scheduling. EDF scheduling is considered in Section 4.6.3. We differentiate partitioned DAG tasks from partitioned sequential tasks by not forcing an entire DAG task to be executed on one processor. Clearly, the sequential approach is ineffective when we have $W_i > D_i$ for any DAG task $\tau_i$. That is, for heavy tasks which have density greater than one.

While each DAG task $\tau_i$ is still characterized by a 3-tuple $(G_i, D_i, T_i)$ as defined in Section 1.3, we extended the properties of each individual subtask to cope with the execution restrictions. More precisely, a subtask $v_{i,j}$ is characterized not only by a WCET $C_{i,j}$ but also by the processor $P_{i,j}$ to which it is statically assigned. Formally, $v_{i,j} = (C_{i,j}, P_{i,j})$, $\forall v_{i,j} \in V_i$. We assume that each subtask can execute only on the single pre-defined core (subtask partitioning), and the complete subtask-to-core mapping to be given. The mapping assumption is relaxed in Section 4.6, where we propose a partitioning and allocation strategy. Note that a mapping phase is essential to derive tighter estimates on WCETs under such parallel settings due to the presence of shared resources and the magnitude of memory-alike transactions. Although the mapping of parallel tasks subject to strict timing constraints is an open problem, for schedulability purposes, it suffices to consider that the WCETs of the subtasks have been computed accordingly.

For any subtask $v_{i,j}$, its set of ancestors assigned to a particular core $p$ is given by $ances(v_{i,j}, p)$. Analogously, $desce(v_{i,j}, p)$ returns the set of descendants of $v_{i,j}$ assigned to core $p$. Independent subtasks may execute in parallel whenever they are mapped to different cores.

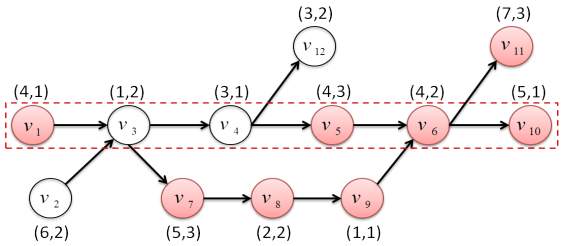



Figure 4.1: Example of a partitioned DAG task with twelve subtasks. Each label indicates the WCET and the core affinity of the corresponding node, respectively.

Let $\ell_i$ denote the number of different paths $\lambda_{i,\ell}$ that can be extracted from DAG $G_i$, i.e. $\ell \in \{1,2,\ldots,\ell_i\}$. The set $proc(\lambda_{i,\ell})$ contains all the distinct cores associated to a path $\lambda_{i,\ell}$, whereas $v_{i,a}^p$ and $v_{i,z}^p$ represent the first and the last subtasks in $\lambda_{i,\ell}$ assigned to core $p$, respectively. We further define the length of a path $len(\lambda_{i,\ell})$ as the sum of the WCET of all its subtasks. Formally, $len(\lambda_{i,\ell}) = \sum_{\forall v_{i,j} \in \lambda_{i,\ell}} C_{i,j}$.

Note that, under partitioned scheduling, $L_i$ may not represent the WCET of $\tau_i$ (also its WCRT in isolation) when the number of cores available is sufficiently large. Indeed the degree of parallelism is constrained by the subtask-to-core mapping, thus two concurrent subtasks assigned to the same core are forced to execute sequentially. Besides $L_i \leq D_i$, partitioned scheduling allows us to define further necessary feasibility conditions. Hence, we introduce the notion of *p-workload*.

**Definition 10** (p-workload)**.** *The worst-case workload of a DAG task $\tau_i$ on a core $p$, denoted by $W_i^p$, is the maximum processing time that any instance of $\tau_i$ requires from $p$. That is, $W_i^p = \sum_{j=1}^{n_i} \{C_{i,j} \mid P_{i,j} = p\}$.*

The following additional relations must then hold for the feasibility of a DAG task system: $W_i^p \leq D_i$ and $\sum_{i=1}^{n} \frac{W_i^p}{T_i} \leq 1, \forall p \in [1,m]$.

**Example 11.** *Fig. 4.1 illustrates our extended model for a DAG task $\tau_i$ comprised of twelve subtasks ($n_i = 12$) $V_i = \{v_1,\ldots,v_{12}\}$ and twelve precedence constraints. For simplicity, we omit the subscript i on the subtasks of $\tau_i$ as well as its dummy source and sink nodes. The label next to each node represents the 2-tuple $v_j = (C_j, P_j)$. For instance, subtask $v_6$ has a WCET of $C_6 = 4$ and is assigned to core $P_6 = 2$. There is a total of ten paths ($\ell_i = 10$) in the DAG. The critical path length of $\tau_i$ equals to $L_i = 26$ and is found on the path $\lambda_{i,\ell} = (v_2, v_3, v_7, v_8, v_9, v_6, v_{11})$. The workload of $\tau_i$ is $W_i = 45$, whereas its maximum p-workload resides on core $p = 3$ with the value $W_i^3 = 16$, which results from the subset of subtasks $\{v_5, v_7, v_{11}\}$. Note that this subset imposes no sequencing of subtasks and thus is not a path of the DAG. To clarify, both $v_5$ and $v_7$ are transitive predecessors (or ancestors) of $v_{11}$, but $v_5$ and $v_7$ are independent of each other.*

## 4.3 Response time analysis of partitioned DAG tasks

In this section, we present a schedulability analysis for sporadic DAG tasks with constrained deadlines scheduled in a partitioned fashion with any fixed-priority scheduling algorithm. The schedu-

lability analysis is based on the notion of per-path interference. Unlike its sequential counterpart, partitioned DAG tasks may allocate subtasks to different cores, potentially creating cross-core dependencies. Unless each path is entirely allocated to a single core, the traditional uniprocessor analysis for fixed-priority sequential tasks that relies on the concept of critical instant (Liu and Layland, 1973) cannot be applied on a per-path basis. Therefore, we introduce a novel RTA to cope with the notion of partitioned DAG tasks.

Recall that $\tau_k$ is the DAG task under analysis and $\mathcal{T}$ is sorted by decreasing priority. For simplicity and clarity of presentation, we assume that all the higher priority tasks are sequential. That is, $\tau_i \stackrel{\text{def}}{=} (C_i, P_i, D_i, T_i, J_i)$, $\forall i < k$; meaning $\tau_i$ has a release jitter $J_j$, and each of its jobs can execute only on core $P_i$ for at most $C_i$ time units. In Section 4.5, we generalize the analysis for the case where every task in the system is a DAG task.

We seek to derive an upper-bound on the WCRT of each path $\lambda_{k,\ell}$ of $\tau_k$. To do so, we first introduce some definitions to characterize the different worst-case interference contributions. We remark that the following definitions are a generalization of those presented in Section 3.2 for GFP scheduling, since $\lambda_{k,\ell}$ can suffer interference even when some processors are idle.

**Definition 11** (Inter-task interference)**.** *The inter-task interference $I^i(\lambda_{k,\ell})$ imposed by a higher priority task $\tau_i$ on the $\ell^{th}$ path of a partitioned DAG task $\tau_k$ is the maximum cumulative time during which any subtask $v_j \in \lambda_{k,\ell}$ is ready but cannot be scheduled because $\tau_i$ is executing on processor $P_j$.*

**Definition 12** (Self-interference)**.** *The self-interference $I^k(\lambda_{k,\ell})$ imposed by a partitioned DAG task $\tau_k$ on its own $\ell^{th}$ path is the maximum cumulative time during which any subtask $v_j \in \lambda_{k,\ell}$ is ready but cannot be scheduled because processor $P_j$ is busy executing other subtasks of $\tau_k$ that do not belong to $\lambda_{k,\ell}$.*

Defs. 11 and 12 combined provide a characterization of the maximal overall interference exerted on a path $\lambda_{k,\ell}$ of task $\tau_k$ during the execution of any of its jobs. The response time of the worst-case instance for $\lambda_{k,\ell}$ can then be expressed as follows.

**Theorem 8.** *The worst-case response time of a path $\lambda_{k,\ell}$ of a partitioned DAG task $\tau_k$ is given by*

$$R(\lambda_{k,\ell}) = len(\lambda_{k,\ell}) + I^k(\lambda_{k,\ell}) + \sum_{\forall i<k} I^i(\lambda_{k,\ell}) \tag{4.1}$$

*Proof.* Let $r_a$ be the release time of the first subtask in the path $\lambda_{k,\ell}$. In the scheduling window $[r_a, r_a + R(\lambda_{k,\ell})]$, the entire path requires at most $len(\lambda_{k,\ell})$ time units of execution. The maximum interference caused by self-interfering subtasks on $\lambda_{k,\ell}$ is $I^k(\lambda_{k,\ell})$ according to Def. 12. By Def. 11, the maximum interference exerted on $\lambda_{k,\ell}$ by each higher priority task (i.e., tasks $\tau_i$ such that $i < k$) is $I^i(\lambda_{k,\ell})$. The theorem follows by noting that the set of lower priority tasks cannot influence the schedule of $\tau_k$ in fixed-priority preemptive scheduling. □

To obtain the WCRT of $\tau_i$ we apply Equation 4.1 to each of its paths. The next corollary directly follows from Theorem 8.

**Corollary 2.** *The worst-case response time of a partitioned DAG task $\tau_k$ is given by*

$$R_k = \max_{\ell=1}^{\ell_i} R(\lambda_{k,\ell}). \tag{4.2}$$

As a direct consequence of Corollary 2, DAG task $\tau_k$ is deemed schedulable if $R_k \leq D_k$.

### 4.3.1 Self-interference

Conceptually, the self-interference corresponds to the delay on the completion time of task $\tau_k$ caused by the concurrency among its own subtasks. Since the computing resources are typically scarce, subtasks of $\tau_k$ that could execute in parallel (i.e., they share no direct or transitive precedence constraint) will eventually contend for core-access, thus increasing the response time of $\tau_k$ itself. Under partitioned scheduling this phenomenon is easier to observe as the subtask-to-core assignment dictates which independent subtasks may interfere with each other. In other words, there is no self-interference between two independent subtasks mapped to different processors.

On a particular path $\lambda_{k,\ell}$, $I^k(\lambda_{k,\ell})$ accounts for all the time intervals where any subtask $v_{j'} \in V_k \setminus \lambda_{k,\ell}$ is executing, while there is a ready subtask $v_j \in \lambda_{k,\ell}$ with pending work on the same core $P_j$. Thus, if $P_{j'} \notin proc(\lambda_{k,\ell})$, then $v_{j'}$ can never interfere with $\lambda_{k,\ell}$. Contrary to the inter-task interference, only one instance of such self-interfering subtasks have to be accounted in $I^k(\lambda_{k,\ell})$ because they belong to the same job of $\tau_k$. The absence of individual priorities assigned to the subtasks, together with variability in their execution times, makes self-interference a problem mutual to all the paths of $\tau_k$. That is, if a path $A$ interferes with a path $B$, then it is also possible to construct a scenario where path $B$ interferes with path $A$. Furthermore, we observe that when a subtask $v_{j'} \in V_k \setminus \lambda_{k,\ell}$ shares precedence constraints with every subtask $v_j \in \lambda_{k,\ell}$, such that $P_j = P_{j'}$, $v_{j'}$ cannot interfere with $\lambda_{k,\ell}$. This allows us to derive a less conservative upper-bound on $I^k(\lambda_{k,\ell})$.

**Lemma 11.** *For any partitioned constrained deadline DAG task $\tau_k$ partitioned on $m$ cores, an upper-bound on the interfering workload imposed by $\tau_k$ on its path $\lambda_{k,\ell}$ is given by*

$$I^k(\lambda_{k,\ell}) \leq \sum_{\forall p \in proc(\lambda_{k,\ell})} W_k^p - len(\lambda_{k,\ell}) - \sum_{\forall v_j \in \Theta_{k,\ell}} C_j, \tag{4.3}$$

*where $\Theta_{k,\ell}$ is the set of subtasks of $\tau_k$ that cannot interfere with $\lambda_{k,\ell}$ and is defined as*

$$\Theta_{k,\ell} \stackrel{\text{def}}{=} \bigcup_{\forall p \in proc(\lambda_{k,\ell})} ances(v_a^p, p) \cup desce(v_z^p, p). \tag{4.4}$$

*Proof.* By definition of a constrained deadline task, two instances of $\tau_k$ cannot interfere with each other when $D_k$ is met. Therefore, the maximum interference that $\tau_k$ can impose on its path $\lambda_{k,\ell}$ is upper-bounded by the sum of the WCET of all subtasks assigned to a core $p \in proc(\lambda_{k,\ell})$ excluding the subtasks in $\lambda_{k,\ell}$, i.e. $I^k(\lambda_{k,\ell}) \leq \sum_{\forall p \in proc(\lambda_{k,\ell})} W_k^p - len(\lambda_{k,\ell})$. Any subtask $v_j$ that

is a successor, either directly or transitively, of $v_z^p$ (i.e., the last subtask of $\lambda_{k,\ell}$ assigned to core $p$), where $p = P_j$, cannot interfere with $\lambda_{k,\ell}$ because $v_j$ becomes ready only after all subtasks in $\lambda_{k,\ell}$ assigned to $P_j$ complete. Similarly, any subtask $v_j$ that is a predecessor, either directly or transitively, of $v_a^p$ (i.e., the first subtask of $\lambda_{k,\ell}$ assigned to core $p$), where $p = P_j$, can be safely discarded because any delay caused by $v_j$ on the start of $v_a^p$ will be accounted in some path $\lambda_{k,\ell'}$, where $\ell' \neq \ell$ and $\{v_j, v_a^p\} \subseteq \lambda_{k,\ell'}$ (from Eq. 4.2). As given by Eq. 4.4, the set $\Theta_{k,\ell}$ contains all those non-interfering subtasks. Hence, $I^k(\lambda_{k,\ell}) \leq \sum_{\forall p \in proc(\lambda_{k,\ell})} W_k^p - len(\lambda_{k,\ell}) - \sum_{\forall v_j \in \Theta_{k,\ell}} C_j$, which concludes the proof. □

Henceforth, let $self(\lambda_{k,\ell})$ denote the set of self-interfering subtasks with $\lambda_{k,\ell}$.

**Example 12.** *Consider the DAG depicted in Fig. 4.1 and let $\lambda_{k,\ell} = (v_2, v_3, v_4, v_{12})$ be the path under analysis (the sequence of light nodes). This path spans over two cores $proc(\lambda_{k,\ell}) = \{1,2\}$ and has a length equal to $len(\lambda_{k,\ell}) = 13$. Since the path never reaches core $p = 3$, the p-workload on such core is not accounted as interfering workload. Moreover, subtasks $v_1$ and $v_{10}$ cannot interfere with the path on core $p = 1$ because they are a ancestor and a descendant of subtasks $v_4 = v_a^1 = v_z^1$, respectively; meaning $\Theta_{k,\ell} = \{v_1, v_{10}\}$. Thus, we have $I^k(\lambda_{k,\ell}) \leq 29 - 13 - 9 = 7$, which corresponds to the sum of the WCET of the subtasks in $self(\lambda_{k,\ell}) = \{v_6, v_8, v_9\}$.*

### 4.3.2 Inter-task interference

The inter-task interference $I^i(\lambda_{k,\ell})$ accounts for all the time intervals during which any subtask $v_j \in \lambda_{i,k}$ is ready but it cannot execute since higher priority task $\tau_i$ is holding the processor. Hence, $\tau_i$ may interfere with multiple subtasks in $\lambda_{k,\ell}$ as long as they are mapped to core $P_i$. Despite $\tau_i$ being any task with higher priority than $\tau_k$, if $P_i \notin proc(\lambda_{k,\ell})$, then $\tau_i$ cannot interfere with $\lambda_{k,\ell}$. We denote by $hp(\lambda_{k,\ell})$ the set of higher priority tasks that can effectively interfere with $\lambda_{k,\ell}$.

Although in global multiprocessor scheduling one must consider carry-in jobs as part of the individual interference contributions (Davis and Burns, 2011), this is not the case for the fixed-priority partitioned scheduling of a DAG task when the higher priority tasks are sequential, as the problem boils down to single core scheduling (or a collection of). In this context, $I^i(\lambda_{k,\ell})$ is a function of the maximum number of interfering jobs released by $\tau_i$ in a scheduling window $[r_a^p, f_z^p)$, where $r_a^p$ is the release time of subtask $v_a^p \in \lambda_{k,\ell}$ and $f_z^p$ is the completion time of subtask $v_z^p \in \lambda_{k,\ell}$ with $p = P_i$, as $\tau_i$ cannot interfere with subtasks on a core different than $P_i$. No job released by $\tau_i$ at any instant prior to $r_a^p$ interferes with $\lambda_{k,\ell}$ because otherwise there would exist a valid scheduling window $[r_a^p - t, f_z^p)$ that would increase the response time of the path.

A simple solution to upper-bound $I^i(\lambda_{k,\ell})$ is to assume that $\tau_i$ is allowed to release jobs synchronously with every subtask $v_j \in \lambda_{k,\ell}$ such that $P_i = P_j$. Subsequent job releases are then separated by $T_i$ time units. A similar technique to this independent worst-case scenario for each subtask in $\lambda_{k,\ell}$ was adopted in (Palencia et al., 1997). Clearly, it is not always possible for any $\tau_i$ to interfere with all the workload of $\lambda_{k,\ell}$ assigned to $P_i$, thus this technique is often overly pessimistic.

Unfortunately, finding tight upper-bounds on the interfering workload of the higher priority tasks is difficult. The challenge comes from the fact that the workload of the path $\lambda_{k,\ell}$ on a certain core $p$ may not be continuous, as some of the intermediate subtasks are assigned to different cores. As a result, cross-core dependencies on the execution flow of the path exist. The length of these discontinuous intervals is not fixed since the release of a transitive successor on $p$ depends on the response time of the intermediate subtasks mapped to other cores. Therefore, $I^i(\lambda_{k,\ell})$ is not only a function of the response time of the subtasks in $\lambda_{k,\ell}$ on core $P_i = p$, but it also has to account for the gaps within the different parts of the workload, and for the relation between the duration of the gaps and the response time of the subtasks in $\lambda_{k,\ell}$ on any core $p' \in proc(\lambda_{k,\ell}) \setminus P_i$.

To overcome this problem, we propose in the next section an alternative technique to Eq. 4.1, for computing an upper-bound on the response time of any path $\lambda_{k,\ell}$, which is based on the self-suspending tasks theory.

## 4.4 Worst-case response time (WCRT) of an execution path

We now explain how to bound the worst-case response time of each path $\lambda_{k,\ell}$ of task $\tau_k$. To capture the worst-case interference suffered by a path on the different cores to which it is mapped, we model a path as a set of self-suspending tasks. More precisely, one self-suspending task for each core. For the sake of notation conciseness, let $\lambda$ denote the specific path $\lambda_{i,k}$ under analysis. In the remainder of this section, we also assume that any two consecutive subtasks in $\lambda$ assigned to the same core are merged into a single subtask with a WCET equal to the sum of their individual WCETs.

### 4.4.1 Intuition

In the literature, a self-suspending task is often described as an interleaved sequence of execution and suspension regions, where an execution region is a portion of a sequential task that needs to be processed, and a suspension region corresponds to a period of time during which the task voluntarily yields the processor to perform a remote operation. The suspension regions are assumed to have given bounded durations. (A more insightful description is deferred to Appendix A.)

Regarding partitioned DAG tasks, we observe that the existence of precedence constraints between subtasks assigned to different cores leads to a similar behavior. This is easier to see on a path, since there is a direct precedence constraint between every two consecutive subtasks. Consider the partitioned schedule shown in Fig. 4.2 for the path composed of light nodes in the DAG of Fig. 4.1. Although only subtask $v_4$ is assigned to core $p = 1$, it cannot start executing before the joint subtask $v_1 + v_3$ completes execution on core $p = 2$. In turn, the execution of $v_4$ delays the release of the last portion of workload assigned to core $p = 2$ (i.e., subtask $v_7$), creating an intermediate interval of 3 time units where core $p = 2$ is free to execute lower priority tasks. Such time intervals can be seen as suspensions within the path on a certain core $p$.

In this sense, a path $\lambda$ can be modeled as a set of sporadic self-suspending tasks, one for each core reached by the path, i.e., $\forall p \in proc(\lambda)$. The trick is to treat the subtasks of $\lambda$ assigned to the

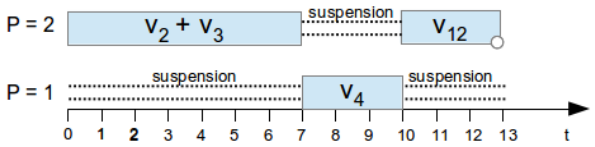



Figure 4.2: An example of a partitioned schedule for the path of Fig. 4.1 formed by the light nodes.

current core under analysis as execution regions and the response time of all its remaining subtasks as suspension regions. The problem of computing the WCRT of $\lambda$ on a multiprocessor platform becomes then equivalent to the RTA of $|proc(\lambda)|$ self-suspending tasks in a uniprocessor system. However, unlike previous works, the duration of the suspension regions are not known beforehand as they are in fact computations to be executed on different cores.

Subtasks assigned to a core $p' \neq p$ and released before the first execution region or after the last execution region of a self-suspending task formed on core $p$ must not be consider as part of its suspension regions, since they do not contribute to an increase on the interfering workload[1]. For instance, the suspensions on core $p = 1$ in Fig. 4.2 do not influence the response time of $v_4$, just its release time. Every other subtask assigned to a core $p' \neq p$ is henceforth called "remote subtask".

### 4.4.2 The self-suspending task model

For ease of understanding, we start with a generic model that closely relates to the ones considered in the literature (it follows the model adopted in Section A.3). Let $\tau^p$ denote the self-suspending task formed by a path $\lambda$ on core $p$. Each self-suspending task $\tau^p$, $\forall p \in proc(\lambda)$, consists of $q \geq 1$ *execution regions* and $q-1$ *suspension regions* such that any two consecutive execution regions are separated by a suspension region. Formally, $\tau^p \stackrel{\text{def}}{=} \{(C_1^p, S_1^p, C_2^p, \ldots, S_{q-1}^p, C_q^p), S^{p,ub}\}$, where $C_h^p$ is the WCET of the $h^{th}$ execution region of $\tau^p$, while an upper-bound on the duration of the $h^{th}$ suspension region of $\tau^p$ is given by $S_h^p$. The parameter $S^{p,ub}$ denotes an upper-bound on the overall suspension time.

Note that $S^{p,ub}$ is not necessarily equal to the maximum cumulative suspension time. That is, $S^{p,ub} \leq \sum_{h=1}^{q-1} S_h^p$. While it is easy to see that each $C_h^p$ corresponds to the WCET of the $h^{th}$ subtask of $\lambda$ assigned to core $p$, the values of each $S_h^p$ and $S^{p,ub}$ cannot be directly derived from $\lambda$ and must therefore be computed. We show next how these values can be expressed as functions of the WCRT of the remote subtasks of $\tau^p$.

Additionally, we represent by $E_h^p$ the $h^{th}$ execution region of $\tau^p$, $\forall h \in [1,q]$. A sequential task is a self-suspending task with no suspension regions. In this particular case, $\tau^p$ is represented as $\tau^p \stackrel{\text{def}}{=} \{(C_1^p), 0\}$.

[1] Such suspensions are relevant only when the self-suspending task is part of the interfering workload, which is not the case here.

### 4.4.3 Methodology

The purpose of the above model is to not tie the analysis of a partitioned DAG task to a particular work on self-suspending tasks. Thus, any existing uniprocessor RTA for fixed-priority sporadic self-suspending tasks can be used to derive the WCRT of a self-suspending task $\tau^p$, denoted by $R(\tau^p)$. As the WCRT of each $\tau^p$ encompasses the WCRT of all the subtasks assigned to core $p$, the WCRT of a path $\lambda$ can be expressed by its self-suspending tasks. Consequently, Eq. 4.1 is rewritten as follows.

**Theorem 9.** *The WCRT of a path $\lambda$ is upper-bounded by*

$$R(\lambda) \leq \sum_{\forall p \in proc(\lambda)} (R(\tau^p) - S^{p,ub}) \tag{4.5}$$

*Proof.* Each self-suspending task $\tau^p$ models the worst-case request of the path $\lambda$ on core $p$. By definition, $R(\tau^p)$ upper-bounds the cumulative response time of its execution and suspension regions. Thus $R(\tau^p)$ also upper-bounds the sum of the response time of the subtasks of $\lambda$ assigned to core $p$. By summing the WCRT of each $\tau^p$, $\forall p \in proc(\lambda)$, we therefore upper-bound the sum of the response time of all the subtasks in $\lambda$. Furthermore, because by definition of $\tau^p$ the suspension time of a task $\tau^p$ corresponds to the processing time of $\lambda$ on cores different than $p$, the maximum suspension time $S^{p,ub}$ should not be considered as part of $R(\tau^p)$ as it is already accounted by the other self-suspending tasks $\tau^{p'}$, where $p' \neq p$. □

An important aspect to consider is that the self-suspending tasks depend on each other through the suspension regions. Bounds on the suspension regions are necessary to analyze the WCRT of the execution regions, but the suspension regions are indeed execution regions on other cores. In the following we explain how to break this circular dependency by capturing the worst-case behavior of the remote subtasks of $\tau^p$.

A well-known result from the sporadic self-suspending tasks theory is that larger suspensions cannot lead to a decrease in the interference suffered by the task under analysis. Therefore, we are interested in finding the WCRT of all the remote subtasks of $\tau^p$ as a way to characterize the worst-case request of $\tau^p$. We first present how $S^{p,ub}$ can be computed.

Whenever multiple remote subtasks of $\tau^p$ are assigned to the same core $p' \neq p$, assuming an independent WCRT for each one of them is overly pessimistic. In fact, they form an inner self-suspending task within $\tau^p$. Let $\tau_{ss}$ denote that inner self-suspending task. Then, the WCRT of all such remote subtasks on $p'$ together is given by $R(\tau_{ss}) - S^{ub}_{ss}$.

A special case happens when there is only one remote subtask of $\tau^p$ allocated to core $P_j \neq p$. Let $v_j$ be that subtask. If no other remote subtask of $\tau^p$ is assigned to $P_j$, then the worst-case duration of the suspension region generated by $v_j$ is given by the worst-case response time $R(v_j)$ of $v_j$. Subtask $v_j$ can then be considered an independent sequential task for which the critical instant in uniprocessors holds.

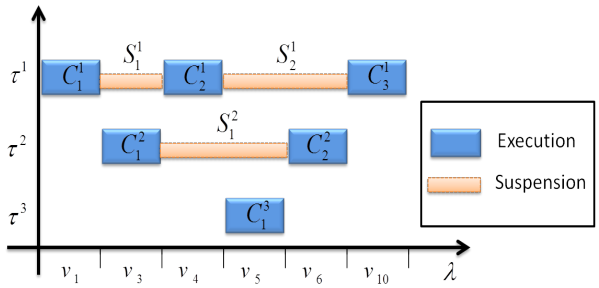



Figure 4.3: Overview of the path $\lambda$ highlighted in Fig. 4.1 as a set of self-suspending tasks.

Let $R(\tau_{ss}^o)$ denote the upper-bound on the WCRT of the remote subtasks of $\tau^p$ assigned to a core $o$. Then, the maximum overall suspension time experienced by $\tau^p$ is

$$S^{p,ub} = \sum_{\forall o \in proc(\lambda), o \neq p} (R(\tau_{ss}^o) - S_{ss}^{o,ub}). \tag{4.6}$$

Although parameter $S^{p,ub}$ provides no information about the relation between the different suspension regions, it can seamlessly serve as input to any self-suspending analysis that disregard the placement and duration of each suspension region specifically.

We now present how to compute an upper-bound on $S_h^p$, $\forall h \in [1, q-1]$. A simple solution is to compute an independent WCRT for each of the remote subtasks of $\tau^p$ and sum the resulting values of those that belong to the same suspension region. Thus, if the remote subtasks $v_j$ and $v_{j+1}$ separate the execution regions $E_1^p$ and $E_2^p$, then $S_1^p = R(v_j) + R(v_{j+1})$ (similarly to (Palencia et al., 1997)). The pessimism can be reduced by considering the fact that if a inner self-suspending task $\tau_{ss}^o$ resides inside a single suspension region of $\tau^p$, the cumulative WCRT of the remote subtasks of $\tau^p$ on core $o$ that correspond to execution regions in $\tau_{ss}^o$ can be replaced by $R(\tau_{ss}^o) - S_{ss}^{o,ub}$. Assume that only remote subtask $v_j$ separates such remote subtasks and $p \neq P_j \neq o$. Accordingly, $S_h^p = R(\tau_{ss}^o) - S_{ss}^{o,ub} + R(v_j)$.

**Example 13.** *Fig. 4.3 depicts how each core perceives the path $\lambda$ of task $\tau_i$ highlighted in Fig. 4.1. We describe through this example how to characterize each self-suspending task $\tau^p$ for that particular path. On core $p = 3$, all the remote subtasks appear before and after the execution region $E_1^3$ constituted of $v_5$, so $\tau^3$ is a sequential task with $C_1^3 = C_5$. On core $p = 2$, $\tau^2$ has $q = 2$ execution regions with $C_1^2 = C_3$ and $C_2^2 = C_6$ and a single suspension region comprised of two remote subtasks. The maximum duration of the suspension region is given by the independent upper-bounds on $v_4$ and $v_5$, i.e., $S_1^2 = R(v_4) + R(v_5)$. Since there is only one suspension region, $S^{2,ub} = S_1^2$. Finally, $\tau^1$ has $q = 3$ execution regions with $C_1^1 = C_1$, $C_2^1 = C_4$ and $C_3^1 = C_{10}$, and two suspension regions for which $S_1^1 = R(v_3)$ and $S_2^1 = R(v_5) + R(v_6)$. Note, however, that suspending subtasks $v_3$ and $v_6$ form in fact $\tau^2$. Thus, $S^{1,ub} = R(\tau^2) - S^{2,ub} + R(v_5)$.*

Given the above relations, it becomes clear that to capture the worst-case request of a self-suspending task $\tau^p$ many inner self-suspending tasks must be characterized and analyzed. This implies that an effective order of computations is necessary to drive the whole algorithm.

### 4.4.4 Unfolding the path

To overcome the dependency problem, we propose an algorithm that recursively divides a path into smaller ones, creating a tree of subpaths which represent self-suspending tasks. The tree reflects the hierarchy of dependencies. When a leaf is reached, the corresponding task has no suspension regions (i.e., it is a sequential task), thus its WCRT does not depend on anything else other than the interfering workload on that core and can be computed immediately. The computed values are then back propagated to the self-suspending tasks on the upper levels, so that their suspension time is no longer unknown. After setting the appropriate bounds on the suspension regions, the WCRT of the execution regions of the next self-suspending task is derived. The process continues level-by-level until the root is revisited. At this point, the WCRT of the original path can be computed thanks to Theorem 9.

Algorithm 4 shows the pseudo-code of the recursive algorithm for unfolding any path $\lambda$ and computing the required WCRTs. It takes as input a path $\lambda_{ss}$, and both the set of higher priority tasks $hp(\lambda)$ and the set of self-interfering subtasks $self(\lambda)$ regarding the path $\lambda$. Initially, $\lambda_{ss} = \lambda$. The algorithm starts by finding the number of subtasks in the path $\lambda_{ss}$ (line 2). If the path is spread over more than one core (lines 10-30), there are self-suspending tasks with suspension regions to be identified. Here, we consider two cases.

First (lines 11-16), if both the first and the last subtask ($v^{1st}$ and $v^{last}$ respectively) of $\lambda_{ss}$ reside on the same core (let $p$ be that core), then $\lambda_{ss}$ constitutes one of the self-suspending tasks[2] on core $p$. However, its response time cannot be computed straight away because the bounds on its suspension regions are not known yet. Thus, we exclude $v^{1st}$ and $v^{last}$ from $\lambda_{ss}$ and invoke the algorithm for the resulting subpath. The second case deals with paths that start and end on different cores (lines 17-29). In this situation, we just split the path $\lambda_{ss}$ in two subpaths to be analyzed: i) from the first subtask of $\lambda_{ss}$ on the same core as $v^{last}$ to $v^{last}$ (lines 18-26), and ii) $\lambda_{ss}$ except $v^{last}$ (lines 27-28).

The recursion on a path stops when there is only one subtask in $\lambda_{ss}$. The WCRT of a single subtask can then be computed by adding the self-interfering workload to the traditional equation for fixed-priority sequential tasks in uniprocessors. That is, the WCRT $R(\lambda_{ss})$ when $\lambda_{ss}$ contains only one subtask is given by Eq. 4.7 and reflected in line 8 of Algorithm 4.

$$R(\lambda_{ss}) = C^{1st} + \sum_{\forall \tau_i \in hp(\lambda_{ss})} \lceil \frac{R(\lambda_{ss}) + J_i}{T_i} \rceil \times C_i + \sum_{\forall v_j \in self(\lambda_{ss})} C_j \tag{4.7}$$

As soon as all the subpaths of a self-suspending task $\tau_{ss}$ on $p$ have been analyzed, the upper-bound on the total duration of its suspensions $S^{ub}_{ss}$ is given by the sum of the WCRT of its inner self-suspending tasks that are not on $p$ (as discussed in Section 4.4.3). All of these values have already been computed and available in the matrix $RTs$ returned by Algorithm 4. Every $C_{ss,h}$ is assigned with the WCET of the $h^{th}$ subtask in $\lambda_{ss}$ on $p$. Apart from the overall bound on the suspension time, an individual upper-bound $S_{ss,h}$ on each suspension region is also required. While parsing the path $\lambda_{ss}$, if the WCRT of a remote subtask is missing in $RTs$, we apply Eq. 4.7 to that particular subtask.

[2]Note that a self-suspending task $\tau^p$ may be decomposed into multiple inner self-suspending tasks, some of which constituting remote subtasks for a self-suspending task $\tau^{p'}$ where $p \neq p'$.

**Algorithm 4:** Computation of the WCRT of each self-suspending task in the path $\lambda$

1 **Function** *PathAnalysis* ($\lambda_{ss}$, $hp(\lambda)$, $self(\lambda)$) **is**
  **Inputs** : $\lambda_{ss}$ - (sub) path under analysis
    $hp(\lambda)$ - set of higher priority tasks w.r.t. path $\lambda$
    $self(\lambda)$ - set of self-interfering subtasks w.r.t path $\lambda$
  **Output**: $RTs$ - matrix $n_i \times n_i$ that stores the computed response time from a subtask $v_a$ to another subtask $v_b$
2  $size \leftarrow |\lambda_{ss}|$;
3  $v^{1st} \leftarrow$ first subtask in $\lambda_{ss}$;
4  $v^{last} \leftarrow$ last subtask in $\lambda_{ss}$;
5  $hp(\lambda_{ss}) \leftarrow \bigcup_{\forall \tau_i \in hp(\lambda)} \{\tau_i \mid P_i = P^{1st}\}$;
6  $self(\lambda_{ss}) \leftarrow \bigcup_{\forall v_j \in self(\lambda)} \{v_j \mid : P_j = P^{1st}\}$;
7  **if** $size = 1$ **then**
8  $R(\lambda_{ss}) = C^{1st} + \sum_{\forall \tau_i \in hp(\lambda_{ss})} \lceil \frac{R(\lambda_{ss})+J_i}{T_i} \rceil \times C_i + \sum_{\forall v_j \in self(\lambda_{ss})} C_j$;
9  $RTs(v^{1st}, v^{last}) \leftarrow R(\lambda_{ss})$;
10  **else**
11  **if** $P^{1st} = P^{last}$ **then**
12  $\lambda_{ss}^{sub} \leftarrow \lambda_{ss} \setminus \{v^{1st}, v^{last}\}$;
13  $PathAnalysis(\lambda_{ss}^{sub},\ hp(\lambda),\ self(\lambda))$;
14  $\tau_{ss} \leftarrow setSuspendingTask()$;
15  $R(\tau_{ss}) \leftarrow ssRT(\tau_{ss}, hp(\lambda_{ss}), self(\lambda_{ss}))$;
16  $RTs(v^{first}, v^{last}) \leftarrow R(\tau_{ss})$;
17  **else**
18  $\lambda_{ss}^{sub} \leftarrow \lambda_{ss}$;
19  **foreach** $v_j \in \lambda_{ss}$ **do**
20  **if** $P_j \neq P^{last}$ **then**
21  $\lambda_{ss}^{sub} \leftarrow \lambda_{ss}^{sub} \setminus \{v_j\}$;
22  **else**
23  break;
24  **end**
25  **end**
26  $PathAnalysis(\lambda_{ss}^{sub},\ hp(\lambda),\ self(\lambda))$;
27  $\lambda_{ss}^{sub} \leftarrow \lambda_{ss} \setminus \{v^{last}\}$;
28  $PathAnalysis(\lambda_{ss}^{sub},\ hp(\lambda),\ self(\lambda))$;
29  **end**
30  **end**
31  **return** $RTs$;
32 **end**

This allows us to compose the values $S_{ss,h}$ according to the relations of the remote subtasks with each suspension region. All such operations are performed by the function *setSuspendingTask* in line 14. With all the parameters in $\tau_{ss}$ defined, the WCRT of the self-suspending task is compute at line 15. Details about this particular RTA are provided in the subsection below. Finally, the procedure is repeated until all the self-suspending tasks are analyzed.

Applying Algorithm 4 to a path $\lambda$ guarantees that the WCRT of every self-suspending task $\tau^p$ defined from $\lambda$ has been correctly computed.

Fig. 4.4 illustrates the steps performed by Algorithm 4 to resolve the dependencies and derive the WCRT regarding the path $\lambda$ highlighted in Fig. 4.1.

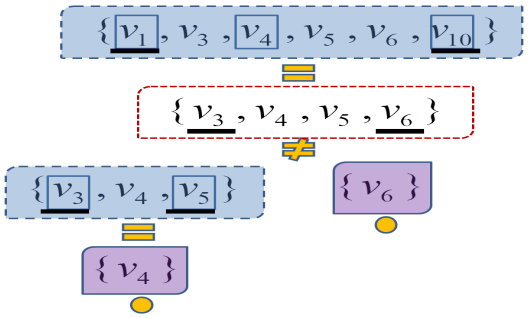



Figure 4.4: Unfolding the path $\lambda$ highlighted in Fig. 4.1.

### 4.4.5 WCRT of a self-suspending task

We now show how three different response time analyses (two from (Bletsas, 2007) and the one presented in Appendix A) for sporadic self-suspending tasks can be extended to cope with both the dependent suspension regions and the self-interfering workload discussed in the previous sections. The worst-case release pattern for the higher priority tasks follows directly from the results of these analyses. Without loss of generality, we consider the WCRT of a self-suspending task $\tau^p$. We start by proving the worst-case release pattern for the self-interfering subtasks in $self(\tau^p)$.

**Lemma 12.** *The contribution of $self(\tau^p)$ to the WCRT of a self-suspending task $\tau^p$ is upper-bounded by releasing a single instance of each self-interfering subtask in $self(\tau^p)$ synchronously with any one execution region $E_h^p$ of $\tau^p$.*

*Proof.* Following Lemma 11, a self-interfering subtask $v_j \in self(\tau^p)$ can delay the execution of a self-suspending task $\tau^p$ at most once for $C_j$ time units. Since all subtasks in DAG $\tau_k$ have the same priority, they cannot preempt each other. Thus, $v_j$ cannot execute when an execution region of $\tau^p$ is active. The maximum interference imposed by $self(\tau^p)$ on $\tau^p$ happens then when each $v_j \in self(\tau^p)$ is released synchronously with an execution region $E_h^p$. Although such self-interference is upper-bounded by $\sum_{v_j \in self(\tau^p)} C_j$ as a whole, the number of higher priority interfering jobs is influenced by the exact placement of each self-interfering subtask, as assuming that $v_j$ interferes with the execution region $E_h^p$ is equivalent to increase $C_h^p$ by $C_j$ time units. In general, enlarging different execution regions leads to different response times. Hence, allowing each of these self-interfering subtasks to be released with any one execution region of $\tau^p$ captures the worst-case contribution of $self(\tau^p)$ to $R(\tau^p)$. □

Suspension-oblivious analyses (Bletsas, 2007) treat suspension regions as part of the computations to be processed, making suspensions subject to the same sources of interference than the execution regions. In this sense, the entire self-suspending task is model as a single sequential task. Consequently, an upper-bound on the WCRT of $\tau^p$ is found when the overall suspension time is the largest (i.e., $S^{p,ub}$) and the self-interfering subtasks are released synchronously with

$E_1^p$. That is,

$$R(\tau^p) = \sum_{h=1}^{q} C_h^p + S^{p,ub} + \sum_{\forall \tau_i \in hp(\tau^p)} \lceil \frac{R(\tau^p) + J_i}{T_i} \rceil \times C_i + \sum_{\forall v_j \in self(\tau^p)} C_j \tag{4.8}$$

The simplest suspension-aware analysis (Bletsas, 2007) focus on upper-bounding the WCRT of each execution region independently. That is, the problem is reduced to a set of smaller sequential tasks by assuming that all the interfering workload releases jobs synchronously with each and every execution region. In this case, the self-interfering subtasks must be accounted once in each execution region $E_h^p$. Moreover, the suspension time has no influence on the interference generated. By using Eq. 4.7 to compute each $R(E_h^p)$, the WCRT of $\tau^p$ is given by

$$R(\tau^p) = S^{p,ub} + \sum_{h=1}^{q} R(E_h^p) \tag{4.9}$$

Both of the aforementioned tests run in pseudo-polynomial time but are substantially pessimistic. The pessimism is further aggravated because multiple self-suspending tasks need to be analyzed to bound the suspensions regions of $\tau^p$. Later in Section A.6 we propose a MILP formulation that finds tight upper-bounds (the best known) on the WCRT of a sporadic self-suspending task with multiple suspension regions. The formulation exploits the duration of the suspension regions to accurately upper-bound the number of jobs released by each higher priority task in each execution region. Therefore, a weak characterization of the relation between the different suspension regions, and also their own bounds, compromises the quality of the solution. As it is implicit in the relation $S^{p,ub} \leq \sum_{h=1}^{q-1} S_h^p$, some upper-bounds on the duration of the suspension regions of $\tau^p$ are over-estimated. Hence we discuss the limitations of our generic model and propose a more robust one that adheres to such formulation.

Consider the following example: $\tau^p$ has two suspension regions, each one of them comprised of a single remote subtask assigned to the same core; let $v_j$ and $v_{j+1}$ represent those remote subtasks, while $\tau_{ss}$ denotes the inner self-suspending task formed by them. If $R(\tau_{ss}) - S_{ss}^{ub} < R(v_j) + R(v_{j+1})$, then the first suspension region plus the second suspension region cannot exceed $R(\tau_{ss}) - S_{ss}^{ub}$. Consequently, one must find the trade off that satisfies $R(\tau_{ss}) - S_{ss}^{ub}$, while still representing a worst-case suspension pattern for $\tau^p$.

To address this issue, let $S_h^p$ denote instead the $h^{th}$ suspension region of $\tau^p$ and be characterized by the 2-tuple $(S_h^{p,lb}, S_h^{p,ub})$, $\forall h \in [1, q-1]$. The parameter $S_h^{p,lb}$ is a lower-bound on the duration of the suspension region $S_h^p$, whereas $S_h^{p,ub}$ is an upper-bound. We now prove the values for the individual lower-bounds.

**Lemma 13.** *Let $V_h^p$ denote the set of remote subtasks within $E_h^p$ and $E_{h+1}^p$ of $\tau^p$. A lower-bound on the suspension region $S_h^p$ is given by $S_h^{p,lb} = \sum_{v_j \in V_h^p} C_j$.*

*Proof.* The WCRT of $\tau^p$ is found when the response time of its remote subtasks is maximized. We must therefore prove that the WCRT of each inner self-suspending tasks $\tau_{ss}^o$ ($\forall o \in proc(\lambda)$, $o \neq p$) within $\tau_p$ cannot be met if a remote subtask $v_j \in V_h^p$ executes for less than $C_j$ time units. The proof is by contradiction. Consider a release pattern $\sigma$ for any inner self-suspending task $\tau_{ss}^o$ that maximizes $R(\tau_{ss}^o)$ but where $R(E_{ss,h}^o) < C_{ss,h}^o$. The execution region $E_{ss_h}^o$ of $\tau_{ss}^o$ corresponds to

a remote subtask $v_j \in V_h^p$. If $E_{ss,h}^o$ executes for its WCET $C_{ss,h}^o = C_j$, an equivalent release pattern $\sigma'$ can be obtained by delaying each subsequent interfering job release by $C_{ss,h}^o - R(E_{ss,h}^o)$ time units. This means that the entire window $[R(E_{ss,h}^o), R(\tau_{ss}^o)]$ in $\sigma$ is repeated after the initial $C_{ss,h}^o$ time units in $\sigma'$. Clearly, $R(\tau_{ss}^o)$ is not the worst-case response time of $\tau_{ss}^o$ which invalidates the hypothesis. □

If an inner self-suspending task $\tau_{ss}^o$ is comprised of a single remote subtask $v_j \in V_h^p$ (i.e., $\tau_{ss}^o$ is sequential), than the value of $S_h^{p,lb}$ can be improved by replacing the contribution of that particular remote subtask with $R(v_j)$ instead of $C_j$. Irrespectively of the type of the inner self-suspending tasks, the upper-bound $S_h^{p,ub}$ is computed as explained in Section 4.4.3. Based on these individual bounds on the suspension regions, the following property holds.

**Property 2.** *Let $R(\tau_{ss,h}^o)$ denote the total WCRT of the remote subtasks within $E_h^p$ and $E_{h+2}^p$ of $\tau^p$ assigned to core $o$, $\forall o \in proc(\lambda)$, $o \neq p$. Admitting any solution, such that (1) $S_h^p + S_{h+1}^p \leq \sum_{\forall o \in proc(\lambda), o \neq p} R(\tau_{ss,h}^o)$, (2) $S_h^{p,lb} \leq S_h^p \leq S_h^{p,ub}$, and (3) $S_{h+1}^{p,lb} \leq S_{h+1} \leq S_{h+1}^{p,ub}$, upper-bounds the worst-case suspension behavior of the suspension regions $S_h^p$ and $S_{h+1}^p$ of $\tau^p$.*

The reasoning behind Property 2 is that, in the general case, no technique exists yet to deem how the suspension time should be distributed between the suspension regions so that $R(\tau^p)$ is maximized. Hence, all the possible combinations should be considered. Property 2 must be applied to every sequence of suspensions regions in which an inner self-suspending task (not sequential) of $\tau^p$ can be identified. We denote by $\psi^p$ the set of constraints that restrict the duration of multiple suspension regions of $\tau^p$ together (i.e., constraints as $S_h^p + S_{h+1}^p \leq \sum_{\forall o \in proc(\lambda), o \neq p} R(\tau_{ss,h}^o)$). Note that an optimization problem is clearly adequate to address the complexity exposed by Lemma 12 and Property 2. Therefore, we describe how to integrate them in the formulation presented in Section A.6.

In the extended MILP formulation, the duration of each suspension region in $\tau^p$ is a real variable denoted $S_h$, while $Y_{j,h}$ is a binary variable that indicates whether ($Y_{j,h} = 1$) or not ($Y_{j,h} = 0$) a self-interfering subtask $v_j \in self(\tau^p)$ is released synchronously with the execution region $E_h^p$. Lemma 12 is formalized by Constraint 4.10

$$\forall v_j \in self(\tau^p) \; : \; \sum_{h=1}^{q} Y_{j,h} \leq 1, \tag{4.10}$$

which can then be integrated in the WCRT computation of each execution region as in Constraint 4.11 (which replaces Constraint A.10), where $NI_{i,h}$ is the number of interfering jobs of $\tau_i$ in $E_h^p$.

$$\forall E_h^p \in \tau^p \; :$$

$$R(E_h^p) = C_h^p + \sum_{\tau_i \in hp(\tau^p)} NI_{i,h} \times C_i + \sum_{v_j \in self(\tau^p)} Y_{j,h} \times C_j. \tag{4.11}$$

That is, each self-interfering subtask interferes with exactly one execution region for its WCET. The individual bounds on the duration of each suspension region are expressed by Constraint 4.12.

$$\forall h \in [1, q-1] \; : \qquad S_h^{p,lb} \leq S_h \leq S_h^{p,ub} \tag{4.12}$$

Constraint 4.13 enforces that the sum of the suspension regions cannot exceed the upper-bound on the overall suspension time. For a more accurate analysis, Constraint 4.13 should be replaced with all the global constraints defined in $\psi^p$.

$$\sum_{h=1}^{q-1} S_h \leq S^{p,ub} \tag{4.13}$$

## 4.5 Higher priority DAG tasks

In this section, we extend our analysis to cope with multiple partitioned DAG tasks interfering with each other. As the WCRT of a DAG task $\tau_k$ is ultimately derived through a collection of self-suspending tasks $\tau_k^p$, we restrict our attention to the worst-case interference imposed by the higher priority tasks on any $\tau_k^p$. Let $V_i^p$ denote the set of subtasks of the higher priority DAG task $\tau_i$ assigned to core $p$. We prove below that each $\tau_i$ can safely be replaced in the response time analysis of $\tau_k^p$ by a set $V_i^{p'}$ of $n_i^p$ sequential tasks, where $n_i^p = |V_i^p|$.

Let a sequential task $\tau_{i,o} \in V_i^{p'}$ correspond to the subtask $v_o \in V_i^p$. The task $\tau_{i,o}$ upper-bounds the worst-case request of the subtask $v_o$, $\forall o \in [1, n_i^p]$, and is defined as $\tau_{i,o} \stackrel{\text{def}}{=} \langle C_{i,o}, D_{i,o}, T_{i,o}, J_{i,o} \rangle$. The worst-case execution time $C_{i,o}$ of $\tau_{i,o}$ is given by the WCET of $v_o \in V_i^p$, that is, $C_{i,o} \stackrel{\text{def}}{=} C_o$. Both the deadline $D_{i,o}$ and the period $T_{i,o}$ are inherited from $\tau_i$. The parameter $J_{i,o}$ denotes the release jitter of $\tau_{i,o}$ and is defined as the difference between the WCRT of $\tau_i$ and the WCET of $v_o \in V_i^p$. Formally, $J_{i,o} \stackrel{\text{def}}{=} R_i - C_o$. This transformation eliminates the dependencies regarding the interfering workload of $\tau_i$ towards $\tau_k^p$ by assuming that each subtask of $\tau_i$ assigned to core $p$ is released independently. A similar method was already proposed in (Palencia et al., 1997) with improved release jitters. Note that the method in (Palencia et al., 1997) could be used here too. The results would in fact be more precise, but at the cost of additional computational complexity.

**Theorem 10.** *The interference exerted by a DAG task $\tau_i \in hp(\tau_k)$ on a self-suspending task $\tau_k^p$ is upper-bounded by the sum of the interferences imposed on $\tau_k^p$ by each sequential task $\tau_{i,o} \in V_i^{p'}$, where $\tau_{i,o} \stackrel{\text{def}}{=} \langle C_{i,o}, D_{i,o}, T_{i,o}, J_{i,o} \rangle$ as defined above.*

*Proof.* The interference exerted by $\tau_i$ on $\tau_k^p$ is equal to the sum of the interference caused by each of the subtasks $v_o \in V_i^p$. We must therefore prove that the interference caused by each task $\tau_{i,o} \in V_i^{p'}$ upper-bounds the interference generated by each $v_o \in V_i^p$. The proof is by contradiction. Let us assume that $v_o$ causes more interference than $\tau_{i,o}$. There might be only two reasons for this to be true: (i) a job released by $v_o$ creates more interference than a job released by $\tau_{i,o}$ , or (ii) $v_o$ releases more jobs than $\tau_{i,o}$ in a given time window.

Since $\tau_{i,o}$ and $v_o$ are both non-self-suspending and the WCET of $\tau_{i,o}$ is equal to the WCET of $v_o$, (i) cannot be true. As the minimum inter-arrival time of $v_o$ is identical to that of its corre-

sponding sequential task in $V_i^{p'}$, only their jitters may cause (ii) to be true. Now, let us compute the maximum jitter that can be experienced by the subtask $v_o$. Let $a_i$ denote the arrival time of any job of $\tau_i$. Since $R_i$ assumes that each subtask of $\tau_i$ executes for its WCET, it means that a subtask $v_o$ of $\tau_i$ cannot start executing later than $R_i - C_o$ after $a_i$ (otherwise it would complete later than $a_i + R_i$ and $R_i$ would not be the WCRT of $\tau_i$). The release jitter of $v_o$ is therefore upper-bounded by $J_o \stackrel{\text{def}}{=} R_i - C_o$. This contradicts (ii) and hence proves the lemma. □

## 4.6 Mapping techniques

While the schedulability analysis presented in the previous sections has the merit to work for any arbitrary mapping and provides invaluable knowledge regarding the cross-core effects, it does so through an exhaustive parsing of the paths in a DAG. As the number of paths tends to grow significantly with the number of edges added to a DAG, the computational complexity of this method becomes prohibitively expensive in the general case. Moreover, a concrete mapping may reduce the interactions between subtasks, thus simplifying the problem at hand. Therefore, in this section, we propose a specific subtask-to-core assignment and respective schedulability test that favors efficiency over (average) performance of the parallel computations.

### 4.6.1 Pessimism in the previous analysis

To motivate our choice of a very simple and linear mapping, we now discuss the different sources of pessimism inherent to the RTA derived for general partitioned DAG tasks scheduled by FP.

- **Self-interference**. Eq. 4.3 includes the full contribution of all the subtasks of $\tau_i$ that are assigned to any core $p \in proc(\lambda_{i,\ell})$, as long as each of such subtasks is not an ancestor or descendant of every subtask in the path $\lambda_{i,\ell}$ assigned to the same core. Consider a fork-join task composed of two sequential segments separated by a parallel segment with two subtasks. One of the two paths is mapped entirely on a single core, while the remaining subtask is designated to execute elsewhere. According to Eq. 4.3 both paths suffer self-interference from the remaining subtask, thus by Eq. 4.1 their response time is equal to the workload of the task if there is no inter-task interference. However, in this situation, it is clear that the WCRT cannot be larger than the critical path length.

  Furthermore, each of the self-interfering subtask in the set $self(\lambda_{i,\ell})$ is allowed to interfere with any execution region when the path $\lambda_{i,\ell}$ is analyzed as a self-suspending task. As some subtasks eventually cannot interfere with certain execution regions due to the precedence constraints, this oversight may increase the response time of the path.

- **Results on self-suspending tasks**. While the uniprocessor RTA for sporadic sequential tasks is exact, the same result is hardly available for sporadic self-suspending tasks. In Section A.5, we present an algorithm to compute the exact WCRT for a self-suspending tasks with one suspension regions when the higher priority tasks are sequential. However,

the algorithm suffers from exponential time complexity even for such limited model. No exact methods exist for the general case. This means that the WCRT of each self-suspending task is likely to be overestimated. Provided each path results in multiple self-suspending tasks, the pessimism is then cumulative.

- **Length of the suspension regions**. Under a sporadic release pattern, longer suspensions cannot lead to reduced interference; quite the opposite. The issue with this special type of self-suspending tasks is that the suspension time distribution is dynamic: the WCRT of a task bounds the total suspension time of another, but the two concrete worst-case scenarios are often incomparable. We solved the problem by allowing the length of each suspension region to vary from its WCET to its independent WCRT, as long as the total duration respects the end-to-end timing constraint giving by the inner self-suspending task. For higher accuracy, the length of an execution region would have to match the length of the corresponding suspension region.

- **Inter-task interference**. Each higher priority partitioned DAG task is decomposed into a set of independent sequential tasks, one for each subtask in a DAG. With this transformation process, the order in which the subtasks must execute is lost as the release times are not constrained according to their functional dependencies. As a result, we have that if one subtask of a DAG job causes interference then all of its subtasks assigned to the same core will contribute to the interfering workload. The pessimism in the computation of the inter-task interference is further increased due to the conservative release jitter assigned to every subtask.

The aforementioned pessimism stems from the fact that a path may be spread over several cores, thus creating cross-core dependencies which are difficult to correlate and quantify with precision, specially in the presence of higher priority workload. By Eq. 4.1 and the schedulability condition $R_i \leq D_i$, we get that a path $\lambda_{i,\ell}$ is allowed to be delayed by at most $D_i - len(\lambda_{i,\ell})$ time units independently of the number of cores used. The more subtasks we map to different cores, the more susceptible the DAG task becomes to inter-task interference. Based on these two observations, we seek to derive a mapping process such that each path uses the least number of cores and paths of a same DAG task execute together if possible.

### 4.6.2 DAG partitioning

We start by addressing the problem of partitioning each DAG task $\tau_i$ independently with the goal of minimizing the processor count that guarantees feasibility. DAG partitioning consists in the agglomeration of multiple subtasks according to a specific criteria, allowing resulting partitions to be effectively distributed among processors without violating the desired semantics. A plethora of DAG partitioning algorithms is available in the graph-theoretical literature (Kwok and Ahmad, 1999), however they do not contemplate real-time constraints, focusing instead on the optimization of communication costs, schedule length or load balancing. Another key differentiator is that

a specific subtask ordering is usually enforced, so that subtasks always execute according to sequence defined in the partition. Although some of these algorithms are designed for scheduling on bounded number of cores, they do not provide a mechanism to minimize such metric. To our knowledge, this highly unexplored aspect of DAG partitioning and scheduling has been addressed firstly by Bozdag et al. (Bozdağ et al., 2009). Nevertheless, the work in Bozdağ et al. (2009) targets DAGs with communication weights and aims at preserving the original schedule length.

Recall that $\lambda_i = \{\lambda_{i,1}, \ldots, \lambda_{i,\ell}\}$ corresponds to the set of all $\ell_i$ paths found in the DAG $G_i$. Now let $P_i = \{P_i^1, \ldots, P_i^h\}$ be a set of $h_i$ notional processors, with $h_i \leq \ell_i$, such that each path could be associated to a dedicated notional processor in the worst-case. The notional processors are completely unrelated to the platform $\Pi$ and they will be mapped to specific cores in a second phase (next subsection). Each notional processor $P_i^h$ contains the set of subtasks $v_j \in V_i$ that have been designated to run virtually on it. We allow the same subtask to be assigned to multiple notional processors. We further define $W(P_i^h)$ as the total workload assigned to $P_i^h$, and $I_{i,h}^x$ as the total workload of those subtasks that are in $P_i^h$ but are not also assigned to a notional processor $P_i^x$. Formally, $W(P_i^h) = \sum_{v_j \in P_i^h} C_j$ and $I_{i,h}^x = W(P_i^h \setminus P_i^x)$, $1 \leq h, x \leq h_i$, $h \neq x$.

We aim at reducing the number of notional processors required to execute a partitioned DAG task $\tau_i$ such that its deadline $D_i$ will always be met. If $\tau_i$ is a light task (i.e., $density \leq 1$), then the entire DAG can be executed on a single processor since $W_i \leq D_i$. In this trivial case, all paths can be merged into a single notional processor, thus $P_i = P_i^1 = V_i$. Instead, when $\tau_i$ is a heavy task (i.e., $density > 1$), finding a DAG partitioning that yields optimal number of processors is challenging. For this reason, we propose an iterative path merging heuristic, starting with $P_i^h = \lambda_{i,\ell}$ for each $h = \ell$, defined according to the following steps:

1. Compute every $I_{i,h}^x$.
2. Let $P_i^a$ (origin) and $P_i^b$ (destination) be a pair of eligible notional processors such that the following quantity is maximal:
$$W(P_i^a) - I_{i,a}^b \tag{4.14}$$
Ties are broken according to (i) $\min W(P_i^a)$, (ii) $\min W(P_i^b)$ and (iii) arbitrarily.
3. If $W(P_i^b) + I_{i,a}^b > D_i$, mark the pair $(P_i^a, P_i^b)$ as ineligible; go to step 2.
4. $P_i^b = P_i^b \cup P_i^a$, $P_i = P_i \setminus P_i^a$, $h_i = h_i - 1$; go to step 1.

   **Stop condition:** Every pair of candidates is ineligible.

The heuristic above attempts to merge as many pairs of notational processors as possible by iteratively prioritizing the pair with the largest common workload. Initially, each notional processor is formed by a distinct path in the DAG $G_i$, thus $h_i = \ell_i$. After computing $I_{i,h}^x$ for every pair $(P_i^h, P_i^x)$ with $h \neq x$, their common workload is given by Eq. 4.14.

At the very first iteration, all pairs are considered eligible for merging. The criteria used to select the candidate pair is based on the similarities between two (sets of) paths. We choose the

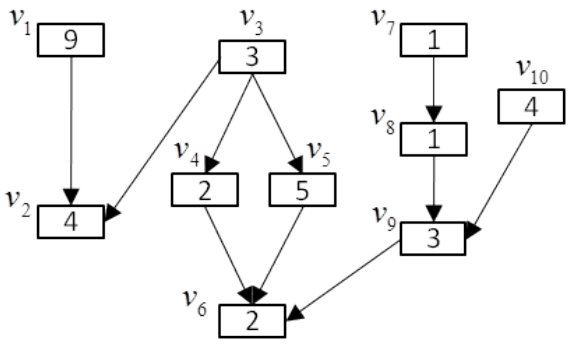



Figure 4.5: A DAG task $\tau_i$ with $W_i = 34$ and $D_i = 16$.

eligible pair $(P_i^a, P_i^b)$ with the maximum duplicated workload because any other pair either (i) has more incompatibilities (larger value of $I_{i,h}^x$), or (ii) their total workload is smaller. As a result, merging $P_i^a$ with $P_i^b$ reduces as much as possible the cumulative workload of all the partitions at that particular step. In case of a tie, we prioritize the less incompatible pair.

Before merging, we check the feasibility of the operation — a pair of notional processors cannot be merged whenever their total non-duplicated workload exceeds the task deadline. Accordingly, if $W(P_i^b) + I_{i,a}^b > D_i$, then $(P_i^a, P_i^b)$ is ineligible for merging and a new candidate pair must be selected. Otherwise, we proceed by merging $P_i^a$ with $P_i^b$: $P_i^b$ receives $I_{i,a}^b$ units of workload due to the union of the two sets and $P_i^a$ is discarded from $P_i$, thus the number of partitions required to schedule $\tau_i$ is decreased by 1. The procedure is repeated until all pairs have been declared ineligible. Note that the equality $h_i = 1$ is never met when the input DAG corresponds to a heavy task, since $W_i > D_i$. Moreover, we remark that the efficiency of the algorithm can be drastically improved since the merging operation is commutative ($W(P_i^h \cup P_i^x) = W(P_i^x \cup P_i^h)$) and only a very small subset of values $I_{i,h}^x$ need to be recomputed after a successful merge.

**Example 14.** *Consider the DAG task $\tau_i$ depicted in Fig. 4.5. This DAG task is heavy ($W_i/D_i = 34/16 > 1$) and has 6 paths. Thus, we start by creating $h_i = 6$ notional processors and assign a distinct path to each of which, resulting in $P_i^1 = \{v_1, v_2\}, P_i^2 = \{v_3, v_2\}, P_i^3 = \{v_3, v_4, v_6\}, P_i^4 = \{v_3, v_5, v_6\}, P_i^5 = \{v_7, v_8, v_9, v_6\}, P_i^6 = \{v_{10}, v_9, v_6\}$. We illustrate the DAG partitioning procedure in Fig. 4.6.*

*At the first iteration, and after computing the values of $I_{i,h}^x$, two pairs of notational processors lead to the maximization of Eq. 4.14: $(P_i^5, P_i^6)$ and $(P_i^3, P_i^4)$ with $W(P_i^5) - I_{i,5}^6 = W(P_i^3) - I_{i,3}^4 = 7 - 2 = 5$. The pair $(P_i^5, P_i^6)$ wins the tiebreaks according to criteria (ii) since $W(P_i^6) = 9 < W(P_i^4) = 10$, and is selected for merging. As $W(P_i^6) + I_{i,5}^6 = 9 + 2 \leq D_i = 16$, the feasibility constraint is satisfied and the pair $(P_i^5, P_i^6)$ is merged into a single notional processor $P_i^5 = \{v_7, v_8, v_{10}, v_9, v_6\}$. The DAG partitioning heuristic continues to successfully merge pairs of notional processors for the next two iterations, reducing $h_i$ to 3 with $P_i^1 = \{v_1, v_3, v_2\}, P_i^2 = \{v_3, v_4, v_5, v_6\}, P_i^3 = \{v_7, v_8, v_{10}, v_9, v_6\}$. At this point, the pair $(P_i^2, P_i^1)$ is selected for merging but is declared ineligible due to $W(P_i^b) + I_{i,a}^b > D_i$. The same happens to the pairs $(P_i^3, P_i^2)$ and $(P_i^3, P_i^1)$. Consequently, the procedure stops as no pair of notional processors can be further*

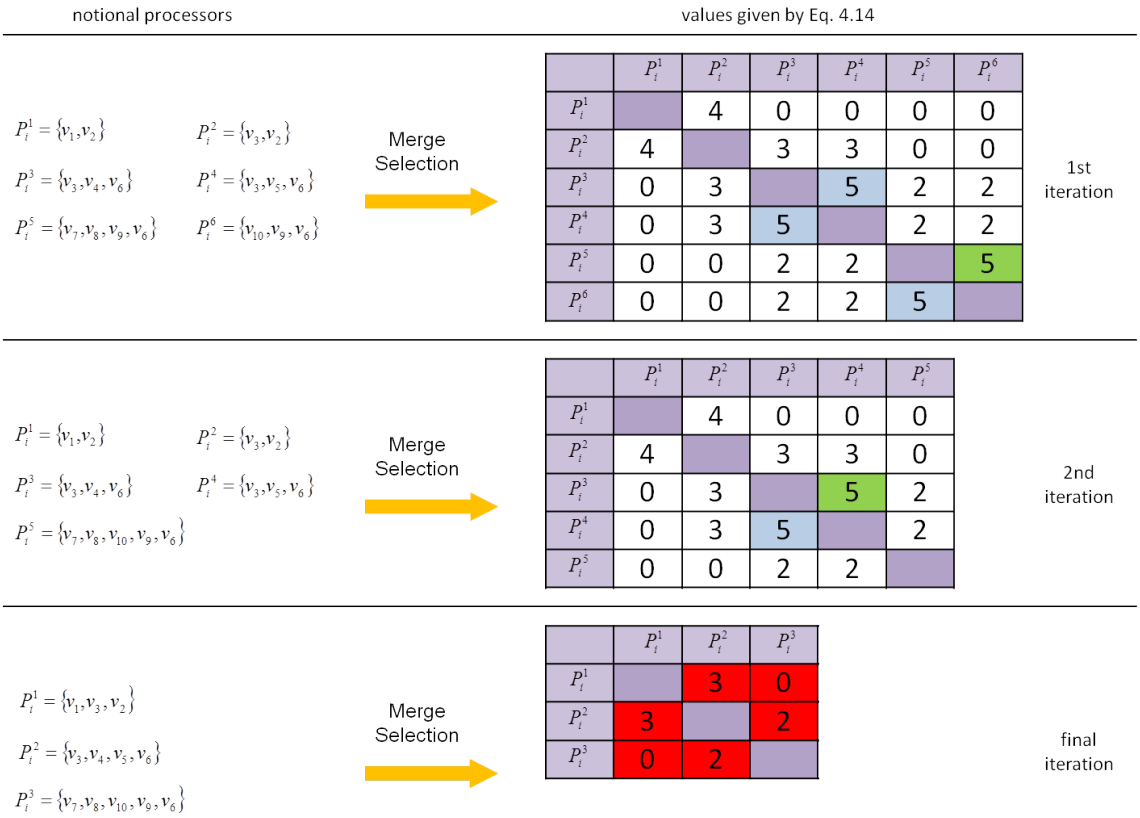



Figure 4.6: Phases of the partitioning heuristic applied to the DAG task $\tau_i$ of Fig. 4.5. Green cells represent the selected eligible pair, blue cells the pairs that reached tiebreaks, and red cells ineligible pairs.

*merged without resulting in a violation of $\tau_i$'s deadline. The final partitioning of $\tau_i$ is then given by $P_i = \{P_i^1, P_i^2, P_i^3\}$.*

It emerges from the DAG partitioning heuristic that a heavy DAG task is feasible on $h_i$ cores by assigning (the subtask of) each final notional processor to a dedicated a core. Since our heuristic is designed to minimize $h_i$, this result allows to save important computing resources when task isolation is considered. For instance, in federated scheduling (Li et al., 2014) a subset of $\lceil \gamma_i \rceil$ processors are exclusively allocated to each heavy task $\tau_i$ with a minimal capacity requirement $\gamma_i = \frac{W_i - L_i}{D_i - L_i}$. Semi-federated scheduling (Jiang et al., 2017) dedicates $\lfloor \gamma_i \rfloor$ processor instead, and schedules the remaining fractional part $\gamma_i - \lfloor \gamma_i \rfloor$ together with the light tasks. However, the lower-bound on the capacity requirement is not tight. Using the example above, we get $\gamma_i = \frac{34-13}{16-13} = 7$, hence both techniques would dedicate 7 processors to guarantee $\tau_i$'s schedulability. In contrast, our heuristic ensures that only 3 cores are required.

### 4.6.3 Allocation and scheduling

Unfortunately, the proposed partitioning does not comply with the model defined in Section 4.2 as some subtasks eventually appear in multiple notional processors. A one-to-one relation is neces-

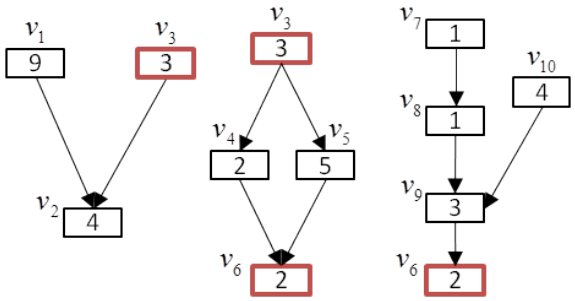



Figure 4.7: Representation of the partitioned DAG task $\tau_i$ after subtask duplication. Duplicated subtasks have their borders colored red.

sary to enable the use of the response time analysis presented from Section 4.3 to Section 4.5. The problem is that the feasibility property may not hold under PFP scheduling in the general case, after deciding on which notional processor is more suitable for each duplicated subtask. Even if a task is feasible under such circumstances, it still has to be deemed PFP-schedulable by our sufficient schedulability test. To circumvent this issue and also address the limitations described in Section 4.6.1, we propose a *subtask duplication* approach that greatly simplifies the partitioned scheduling framework.

Task (or subtask) duplication (Ahmad and Kwok, 1998) is a DAG scheduling technique that assigns subtasks redundantly on multiple cores in order to eliminate a large portion of inter-processor communication. Duplication-based algorithms have been shown to provide much better solutions than non duplication-based ones when the goal is to minimize schedule length (Kwok and Ahmad, 1999). Here, we employ subtask duplication solely to avoid cross-core dependencies. The procedure is based on the outcome of the DAG partitioning heuristic and works in the following manner: (i) each subtask is replicated as many times as it appears within the set of notional processors, (ii) a duplicated subtask becomes ready as soon as all its predecessors partitioned into the same notional processor finish execution, and (iii) a duplicated subtask cannot precede subtasks partitioned into different notional processors. Thus every path is self-contained. Following the DAG depicted in Fig. 4.5 and respective partitioning as discussed in Example 14, Fig. 4.7 provides the visual representation of $\tau_i$ considering subtask duplication. Note that the subtasks assigned to each notational processor form a sub-DAG which is completely independent of the other sub-DAGs.

Since all precedence constraints are now local, the workload within a notional processor executes sequentially no matter how independent subtasks are prioritized. As a result, the problem of scheduling a partitioned DAG task becomes equivalent to the problem of scheduling a set of partitioned sequential tasks (each notional processor is treated as a sequential task). The downside of this approach resides on an increase of the system utilization due to the duplicated subtasks.

Many solutions are available in the literature to decide on the allocation of sequential tasks to cores and to schedule such tasks under the partitioned paradigm (López et al., 2004; Baruah and Fisher, 2005; Davis and Burns, 2011). For this particular problem, we choose to use PEDF as the scheduling algorithm due to the optimality of EDF for preemptive uniprocessor scheduling of

sporadic sequential tasks (Liu and Layland, 1973). Similarly to the work in (Jiang et al., 2017), we adopt the well-known Worst-Fit bin-packing heuristic (Johnson, 1974) to allocate notional processors to cores. Notional processors are considered in the non-increasing order of their density. The density of a notional processor $P_i^h$ is equal to $W(P_i^h)/D_i$.

Specifically, we map the notional processors to the physical cores in $\pi$ at design time according to a EDF schedulability condition. For simplicity, we consider the straightforward density bound, however a more accurate test (such as the demand bound function approximation proposed in (Baruah and Fisher, 2007)) could be used at the cost of increased computational complexity. At each iteration, we select the not-yet-mapped notional processor with the largest density and assign it to a core $p$ ($p \in [1,m]$) such that $p$ has the least total density $\sigma_p$ and $\sigma_p \leq 1$ holds after allocating this notional processor. If a notional processor is assigned to a core $p$, then all subtasks that have been partitioned to it are statically assigned to $p$ and cannot execute elsewhere during run-time. Whenever none of the $m$ cores is able to accommodate a notional processor, the mapping process is aborted and declared infeasible. Otherwise, the mapping is successful and every DAG task $\tau_i$ is schedulable by PEDF.

## 4.7 Experimental results

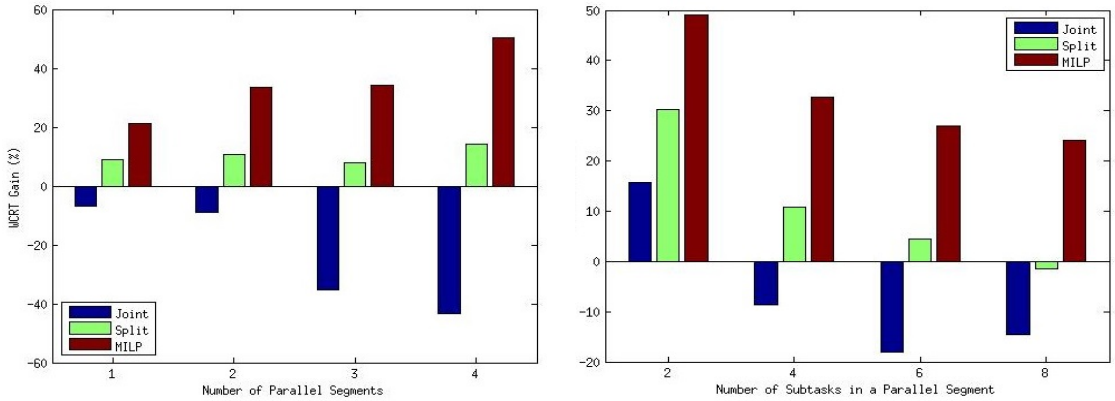
In this section, we evaluate the effectiveness of our two proposed schedulability tests for partitioned DAG tasks comparatively to the state-of-the-art using randomly generated task sets.

### 4.7.1 Evaluation of the response time analysis

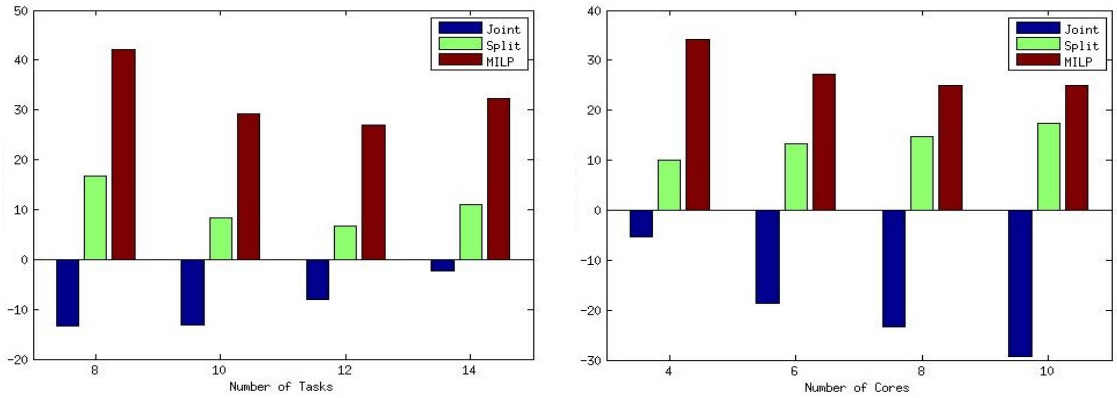
Firstly, we devote our attention to the RTA for fixed-priority scheduling presented from Section 4.3 to 4.5 by evaluating (i) its performance according to the different analysis for self-suspending tasks discussed in Section 4.4.5, and (ii) the gain in terms of WCRT in comparison with an existing analysis for partitioned parallel tasks derived in the context of distributed system. Although most of the available results for distributed systems have been implemented in the MAST tool chain (Universidad de Cantabria, SPAIN, 2014), only the holistic analysis originally developed by Tindell and Clark (Tindell and Clark, 1994) and refined by Palencia et al. (Palencia et al., 1997) has support for multi-path constructs in the form of fork-join tasks (not general DAGs). Thus, we restrict our attention to task sets comprised of $n-1$ sequential tasks and 1 fork-join task, where the fork-join task is the task under analysis.

All the task sets were generated using the `randfixedsum` algorithm (Emberson et al., 2010), allowing us to choose a constant total task set utilization for a given number of tasks and bounded per-task utilization. The total utilization was set to 50% of the platform capacity. For the sequential tasks, the per-task utilization ranged from $[0.05, 0.70]$, while periods were uniformly distributed over $[100, 1000]$. The task execution requirements were calculated from the respective periods and utilizations. For the fork-join task, its workload was set to half of the maximum period (i.e., 500), whereas the WCET of each subtask was uniformly distributed over $[1, 100]$. By default, the fork-join task had 2 parallel segments with 4 subtasks within each segment. All the mapping

decisions were completely random. For this reason, we study the computed WCRT and not the schedulability of the task. Thus, the period of the fork-join task was arbitrarily large. We generated 100 task sets per combination of parameters, while ensuring that all the sequential tasks were always schedulable.




(a) $n = 12$ and $m = 4$.




(b) 2 parallel segments with 4 subtasks each.

Figure 4.8: Average WCRT gain (a)–(b) found by our approach under various system config.

For the first set of experiments, we fixed the number of cores to $m = 4$ and the number of tasks to $n = 12$, while varying the number of parallel segments from 1 to 4 and the number of subtasks within a parallel segment from 2 to 8. Fig. 4.8 inset (a) show the average gain (w.r.t. the WCRT) attained by our analysis comparatively to the holistic analysis, when using the three different tests for self-suspending tasks. These tests are referred to as Joint, Split and MILP, respectively to the order they were presented in Section 4.4.5. To clarify, (i) Joint is the suspension-oblivious approach, (ii) Split assumes an independent worst-case scenario for each subtask, and (iii) MILP finds a worst-case release pattern considering the multiple suspensions. MILP is the only test that outperforms entirely the holistic analysis, with average gains within 20 to 50%. Interestingly, Split exhibits a considerable gain when the number of parallel segments is minimized but ends up in deficit. This behavior can be justified by a higher number of self-interfering subtasks, since they are accounted once in each execution region. Joint has a drastic performance degradation when

the varying parameters are increased because the number of self-suspending tasks observed in the paths grows significantly.

We then study the importance of light and heavy tasks to the inter-task interference. Hence, the second set of experiments had the number of parallel segments fixed to 2 and the number of subtasks in a parallel segment fixed to 4, while we varied the number of tasks in the range $[5,20]$ and the number of cores in the range $[4,10]$. The results are depicted in Fig. 4.8 inset (b). Both MILP and Split outperform the holistic analysis with average gains close to 30% and 10%, respectively. The holistic analysis was derived for arbitrary deadline systems, thus assumes that some subtasks may interfere with themselves. An additional source of pessimism is the individual worst-case scenarios assumed for the subtasks. As the utilization of the interfering tasks increase, Split becomes more competitive mainly because the upper-bounds on suspension time tend to also increase. Inversely, Joint performs very poorly with average losses within 2 to 30% as more workload is assumed to interfere with the suspensions. Although not reported here, Joint becomes a reasonable alternative solution to the MILP when the ratio between the workload of the parallel task and the periods of the interfering tasks is smaller.

### 4.7.2 Evaluation of the duplication-based schedulability test

We now compare the performance of our duplication-based EDF schedulability test for partitioned DAG tasks presented in Section 4.6 (referred to as PPEDF) to other scheduling paradigms for constrained deadline DAGs. Namely, federated scheduling (referred to as Federated) introduced in (Li et al., 2014), semi-federated scheduling (referred to as Jiang) proposed in (Jiang et al., 2017), and our GFP schedulability test (referred to as IRTA-FP) presented in Chapter 3. Our response time analysis for partitioned DAGs is left out of this comparison since it was greatly outperformed by PPEDF across the board. Similar to PPEDF, in both federated approaches (discussed in Section 2.4), light tasks are partitioned according to the Worst-Fit decreasing density heuristic and scheduled by partitioned EDF based on the density bound. Thus, analytically, these three methods are only differentiated by the way heavy tasks are handled. Jiang uses the simpler algorithm SF[x+1] to schedule the fractional part of a heavy task since it performs similarly to SF[x+2] as shown in (Jiang et al., 2017).

Each DAG task is generated according to the method described in Section 3.8. That is, a DAG is obtained from a nested fork-join DAG (see Def. 9 to recall the definition of a NFJ-DAG) by adding directed edges between pairs of nodes according to a probability $p_{add}$. The NFJ-DAG is constructed by recursively expanding its nodes. Each node may spawn up to $n_{par}$ parallel branches with a probability equal to $p_{par}$ or join with a probability equal to $p_{term}$. The maximum number of nested forks is given by $depth$. By default, these parameters are set $p_{par} = 0.8$, $p_{term} = 0.2$, $depth = 2$ and $p_{add} = 0.2$. The key difference is that now the period $T_i$ of a DAG task $\tau_i$ is lower-bounded by $L_i$ instead of $M_i$. Thus, when $U_{tot}$ is varied, $T_i$ is uniformly chosen in the interval $[L_i, W_i/\beta]$, with $\beta = 0.035 \times m$. This lower-bound also holds when we generate a fixed number of tasks for a constant $U_{tot}$ using UUnifast. For each parameter configuration, we generate and evaluate the schedulability of 500 task sets composed of $n$ DAG tasks.

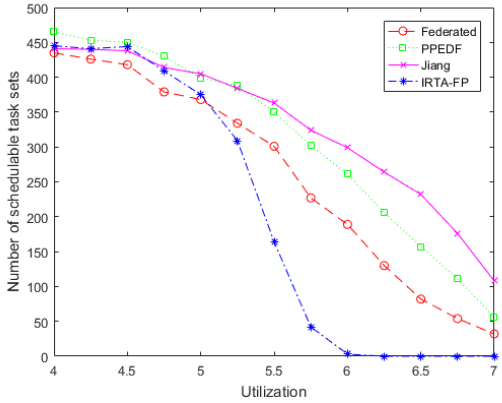



Figure 4.9: PPEDF varying $U_{tot}$ for implicit deadline tasks.

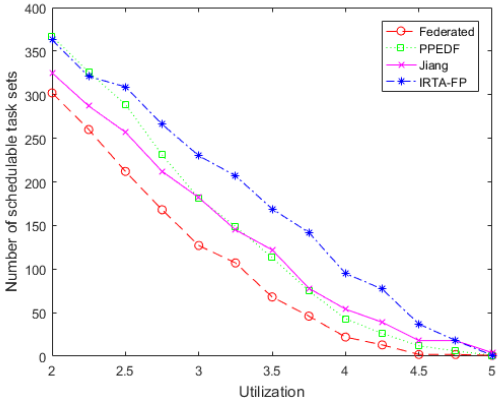



Figure 4.10: PPEDF varying $U_{tot}$ for constrained deadline tasks.

We start by investigating the relation between the period and the deadline, when varying the total utilization $U_{tot}$ and setting $m = 8$. Fig. 4.9 reports the number of schedulable task sets in the case of implicit deadlines and $U_{tot} \in [4,7]$. PPEDF outperforms both IRTA-FP and Federated for any value of $U_{tot}$, with an average steady gain of around 10% comparatively to Federated and huge schedulability improvements comparatively to IRTA-FP when $U_{tot} \geq 5.25$. In comparison to Jiang, PPEDF performs slightly better until $U_{tot} = 5.25$ but then cannot keep the pace, eventually peaking at a 12% performance loss. This trend highlights the drawback of introducing additional workload into a rather congested system by the duplication technique. IRTA-FP is very competitive up to $U_{tot} = 5$ but has a early breakdown utilization at the 75% mark, whereas the other schedulability tests still admit task sets at 90% utilization. Results for the case of constrained deadlines are reported in Fig. 4.10 with $U_{tot} \in [2,5]$. While similar trends can be observed between PPEDF and both federated approaches, these schedulability tests suffer a tremendous performance degradation: they exhibit a breakdown utilization at $U_{tot} = 5$ and cannot schedule even 50% of the task sets when $U_{tot} > 2.5$. That is because they require a significantly larger number of cores to schedule heavy tasks as deadlines become shorter and the schedulability condition is no longer exact. Interestingly, IRTA-FP substantially outperforms every other method, which suggests that global scheduling is an appealing alternative in the more general case of $D_i \leq T_i, \forall \tau_i \in \tau$.

For the next sets of experiments, we restrict our attention to implicit deadline DAG tasks, the total utilization is kept constant at $0.7m$, and both the number of tasks and cores are fixed (by default, we set $n = 1.5m$ and $m = 8$). Fig. 4.11 shows results for various values of $n$. PPEDF performs very poorly for low values of $n$: in particular, when $n \leq 4$, PPEDF struggles to schedule any task set, while both federated approaches are able to admit half of the task sets in average. This behavior indicates that PPEDF fails to partition a DAG with very high utilization on $(W_i - L_i)/(D_i - L_i)$ or less cores, which increases the number of duplicated tasks as the number of successful merges decreases. However, the effectiveness of PPEDF's partitioning algorithm rapidly increases, overcoming both Federated and Jiang when $n \geq 10$ and converging to full schedulability early at $n = 14$ as opposed to $n = 20$. IRTA-FP is also suitable for larger values of $n$, but it is significantly out-

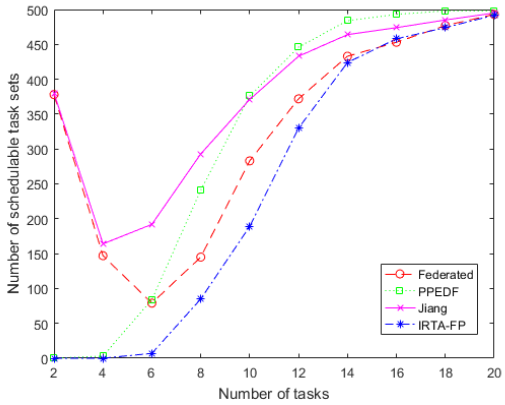



Figure 4.11: PPEDF varying $n$.

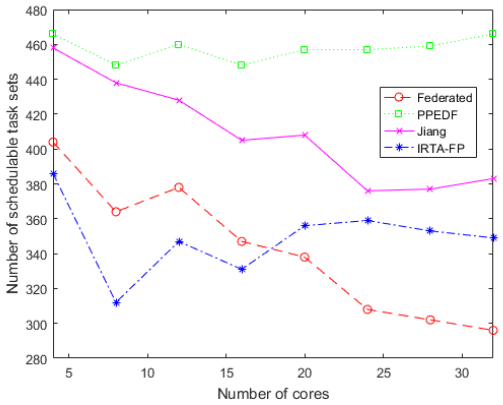



Figure 4.12: PPEDF varying $m$.

performed by PPEDF. For example, IRTA-FP accepts 38% of the task sets when $n = 10$, while PPEDF has an acceptance ratio of 78%. Schedulability results for different values of $m$ are depicted in Fig. 4.12. PPEDF reaches nearly full schedulability and greatly outperforms all the other methods for any values of $m$: in average, it achieves a 13%, 26% and 23% schedulability improvement over Jiang, Federated and IRTA-FP, respectively. Since PPEDF does not dedicate processors to the heavy tasks, this allows for higher flexibility during the bin-packing phase, resulting in an efficient partitioning of the low utilization DAG tasks. As a salient trait, both IRTA-FP and PPEDF are robust to large multiprocessor systems, while Jiang and Federated slightly deteriorate as the core count increases.

Fig. 4.13 presents the number of schedulable task sets when varying the maximum number of parallel branches $n_{par}$ in the range $[2,8]$. Contrary to the other schedulability tests, PPEDF is barely impact by the parallelism in the DAGs and has a very high acceptance ratio (around 88%) for any value of $n_{par}$. Although all the methods tend to converge for $n_{par} \geq 8$, PPEDF offers considerable improvements for DAGs with limited parallelism (i.e., $n_{par} \leq 4$). In particular, when most of the DAG's workload is sequential (i.e., $n_{par} = 2$), IRTA-FP exhibits a 79% loss and only

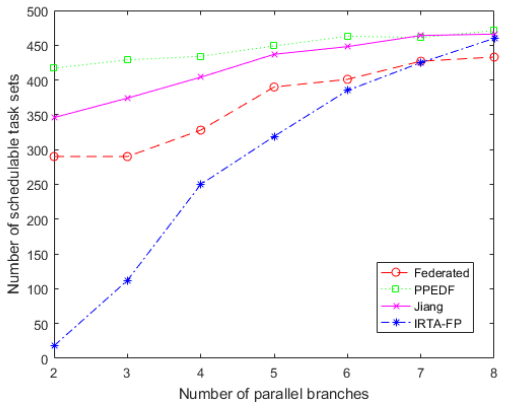



Figure 4.13: PPEDF varying $n_{par}$.

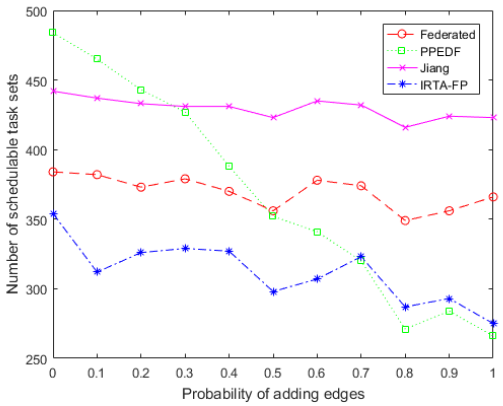



Figure 4.14: PPEDF varying $p_{add}$.

manages to schedule 4% of the task sets, which reflects the well-known larger degree of pessimism present in the analysis of GFP scheduling over the analysis of partitioned scheduling for sequential tasks (Sun and Di Natale, 2018). Fig. 4.14 depicts the results for different DAG structures, since we varied the probability of adding additional edges between nodes of the original DAG from 0 (i.e., NFJ-DAG) to 1 (i.e., synchronous parallel task). PPEDF outperforms all the other methods when $p_{add} < 0.3$, but the situation is reversed when $p_{add} > 0.7$. Comparatively to Jiang, PPEDF shows in average a deficit of 24% for $p_{add} \geq 0.4$. Specifically, the end-to-end performance of PPEDF drops close to 50%, from 484 schedulable task sets in the case of NFJ-DAGs to 266 in the case of synchronous parallel tasks. This strong degradation is justified by a severe increase in the number of paths as the DAG's connectivity is boosted, some of which cannot be merged by the DAG partitioning algorithm, leading to the generation of more notional processors and duplicated subtasks within such tasks.

## 4.8 Summary

Although parallel tasks have recently received considerable attention from the real-time community, most of the available results focus on multiprocessor global scheduling as addressed in Chapter 3. Instead, in this chapter, we studied parallel tasks under a static subtask-to-core mapping as a way to minimize the expected negative effects of such highly parallel models on the lower-level context-dependent timing analysis. We proposed a novel response time analysis for sporadic DAG tasks to be fixed-priority scheduled on a multiprocessor platform in a partitioned fashion. As the analysis is based on the self-suspending tasks theory, we derived a method to model and characterize the worst-case scheduling scenario of a partitioned DAG task as a set of self-suspending tasks. Furthermore, we showed how to transform existing response time analysis for sporadic self-suspending tasks in uniprocessors to analyze partitioned DAG tasks; both simple and more complex techniques. Experiments among randomly generated task sets show that our approach obtains a significant gain in terms of computed WCRT comparatively to the state-of-the-art in distributed real-time systems.

Due to the complexity and pessimism in the previous approach, we then proposed a second schedulability analysis for DAG tasks scheduled by EDF under the partitioned paradigm. We developed a partitioning algorithm that maps similar paths of a DAG to the same processor, aiming to minimize the number of cores that guarantees feasibility and to eliminate cross-core dependencies. Thanks to the duplication of key subtasks, all resulting partitions are independent of each other. Thus, the problem of scheduling a set of partitioned DAGs becomes equivalent to the problem of scheduling a set of sequential tasks on multiprocessors in a partitioned manner. Experimental evaluation demonstrated that this new schedulability test is very effective, achieving a relatively high schedulability ratio and outperforming both global and federated scheduling under most configurations. Moreover, its performance is competitive with semi-federated, while avoiding additional run-time mechanisms and overheads. This suggests that subtask-duplication is a promising solution to the problem of scheduling partitioned DAG tasks.

# Chapter 5

# A Conditional Model for DAG Tasks

Both the schedulabity problems addressed in Chapters 3 and 4 consider a DAG task model where all nodes are triggered and thus executed for every task activation. However, non-trivial real-world applications feature conditional operations that depend on run-time data, leading to workloads with variable sizes and compositions. In this chapter, we identify the need of explicitly modeling such control-flow information for parallel tasks scheduled atop multiprocessor systems. Notably, the contributions of this work have opened a new research direction for the real-time community.

Accordingly, we propose a multi-DAG model where each task $\tau_i \in \mathcal{T}$ is characterized by a collection of execution flows $\mathcal{F}_i = \{F_{i,1}, \ldots, F_{i,n_i}\}$, each of which represented as a separate DAG. Due to conditional statements, only one of such execution flows is taken at run-time by each instance of $\tau_i$. We derive a two-step solution that constructs a single synchronous DAG of servers for a multi-DAG task $\tau_i$ and show that these servers are able to supply every $\tau_i$'s execution flow with the required cpu-budget so that $\tau_i$ can execute entirely, irrespective of the execution flow taken at run-time, while satisfying its precedence constraints. As a result, each multi-DAG task can be modeled by its single DAG of servers, which facilitates in leveraging the existing single-DAG schedulability analysis techniques for analyzing the schedulability of conditional DAG tasks.

## 5.1 Motivation

As we described in Chapter 2, several real-time task models have recently been proposed to cope with different forms of parallelism exposed by modern embedded applications. Typically, these applications also manifest conditional clauses due to control structures (e.g., "if-then-else" statements) within the tasks code; meaning that different activations of a same task may execute different parts of its code. Unfortunately, none of such models (kindly note that the conditional DAG model presented in Section 2.5 was introduced subsequently) explicitly capture these different flows of execution that a parallel task most likely will take during its recurrent activity. Instead, all those models represent a task as a single graph for which *all the subtasks must execute each time the task is released*. In other words, those models are unable to express any control-flow

```
x = A();
y = 0;
if (x > 1) {
  #pragma omp parallel for reduction(+:y)
  for (i = 0; i < 4; i++)
    y += B(i);
} else {
  #pragma omp parallel for reduction(+:y)
  for (i = 0; i < 2; i++)
    y += C(i);
}
z = D(y)
```

Figure 5.1: Example of a conditional parallel program using OpenMP.

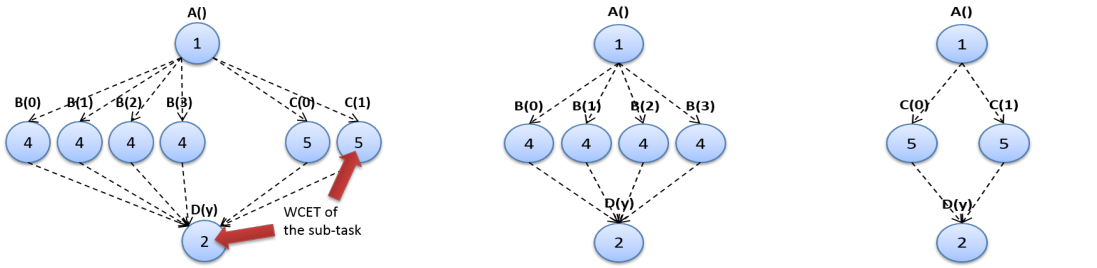
information, such as conditional statements, because they assume a single non-variable internal task structure that has to be fully scheduled and executed at every task release.

Let us consider a simple program (see Fig. 5.1) starting with a call to a function A followed by an if-then-else statement. If the condition of the statement is satisfied, then the program executes in parallel four instances of a function B, otherwise it executes in parallel two instances of a function C. It is trivial to see that an actual run of this code can only take one of the two execution paths (referred to as "execution flows" hereafter). That is, it can either take the *if* execution flow or the *else* execution flow, but never both. If each instance of these functions is modeled as a subtask, then the mutual exclusion between subtasks of B and C (resulting from the conditional clause) cannot be expressed by the parallel models proposed so far in the real-time literature, inadvertently leading to one of the two cases:

**Case 1**: model function $A$ as a node connected to four nodes $B$ and two nodes $C$, which are in turn connected to node $D$ (see Fig. 5.2a). The resulting graph is then composed of 8 nodes (i.e., every possible node) that are all assumed to be executed each time the task is run. The total workload is thus 29, while in fact the task requires at most 19 units of processing time (by taking the "if" branch). This corresponds to an obvious over-approximation of the task's worst-case workload, which makes the high-level analyses (like the schedulability analysis) more pessimistic.

**Case 2**: consider only the "worst-case execution flow" of the program and model that single flow as a graph of subtasks that must all execute. However, determining the worst-case execution flow of a parallel task is extremely challenging since both its response time and interfering contribution may vary significantly and in a conflicting manner from job to job. Assuming a sufficiently large $m$ and no higher priority tasks, the WCRT of a task is reached by executing the flow with maximum "critical path length". In our example, indeed such a WCRT ($= 1+5+2 = 8$) is reached by taking the "else" flow depicted in Fig. 5.2c. Yet, this flow is not necessarily the one exerting the maximum interference on the lower priority tasks because there may exist a shorter flow demanding more processing resources or spanning a higher number of cores. The "if" flow depicted in Fig. 5.2b illustrates such hypothesis, as it spawns 4 parallel subtasks instead of 2 and has workload of 19 as opposed to 13. In the next section, we delve deeper into this scheduling issue.

This dissertation identifies the need to support modeling of such applications for which each run may execute a different set of subtasks and all these subtasks may have different dependencies




(a) Complete graph. (b) "If" execution flow. (c) "Else" execution flow.

Figure 5.2: Possible representations of the program in Fig. 5.1.

from one run to another. The main difference comparatively to the existing parallel task models is that we represent a task as *a set of DAGs* where each DAG models a single execution flow of the application. We believe that such a model is more accurate and reliable than the state-of-the-art as it does *not* create a single DAG by cross-cutting the structures of multiple execution flows with different parameters (for instance, we model the "if" and "else" flows of Fig. 5.2b and Fig. 5.2c as two separated DAGs instead of modeling the task by using the graph of Fig. 5.2a). We also believe that one of the first computation phases of the currently-available WCET analysis tools (Wilhelm et al., 2008), during which the control flow graph is reconstructed by parsing the code of the application, can be adapted to our task model such that the tool will identify every feasible execution flow. However, instead of modeling an execution flow as a simple sequence of instructions or basic blocks (as the tools currently do), a flow will be modeled as a DAG of subtasks with precedence constraints between them.

## 5.2 Scheduling issue

On a *single-core* platform, the fact that a parallel task can take different execution flows during different runs is not a major issue for performing schedulability analysis. To analyze the schedulability of a task set on a *single-core* it is known that for each task only the flow with the largest workload must be considered in the analysis provided parameters such as period and deadline are fixed, because such flow (i) executes sequentially for the longest time and thus (ii) causes the worst-case interference on the other tasks. Consequently, the number of scheduling scenarios to be considered stays within reasonable bounds. Note that this result transposes from the case of classical sequential tasks, where the WCET of a task corresponds to the length of the longest path among every feasible conditional paths and suffices to characterize the worst-case processing and interfering times. Nevertheless, uniprocessor feasibility analysis have been proposed for sequential tasks whose conditional execution is explicitly modeled (Baruah, 1998; Anand et al., 2008).

In contrast, conditional parallel tasks running on a *multicore* architecture introduce a significant problem for the schedulability analysis. First, the interference between two or more graphs of subtasks is much more difficult to capture and analyze, because the interference not only depends

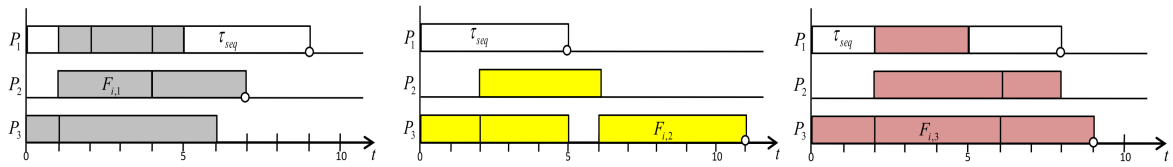



Figure 5.3: GFP schedule of the 3 different execution flows $F_{i,j}$ of the conditional task $\tau_i$ (see Fig. 5.5) and a sequential task $\tau_{seq}$ with $C_{seq} = 5$ on 3 cores.

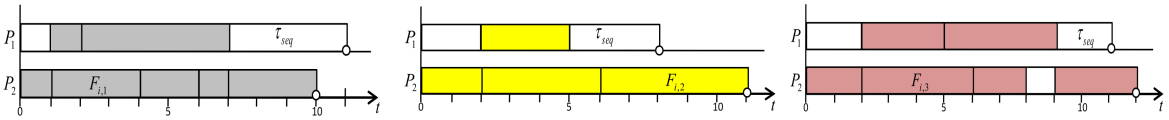



Figure 5.4: GFP schedule of the 3 different execution flows $F_{i,j}$ of the conditional task $\tau_i$ (see Fig. 5.5) and a sequential task $\tau_{seq}$ with $C_{seq} = 5$ on 2 cores.

on the execution time of all the subtasks in each graph, but also on the inherent structure of the graphs themselves. The existence of multiple conditional branches with arbitrary length and parallelism, result in high variance in both the response time and the interfering workload attained by different jobs of the conditional parallel task. As the maximization of these two quantities may be a conflicting goal, identifying the worst-case scheduling scenario is challenging. We substantiate the issue through the example below.

**Example 15.** *Consider*[1] *a higher priority conditional parallel task $\tau_i$ modeled by a multi-DAG as in Fig. 5.5, and a lower priority sequential task $\tau_{seq}$ with a WCET of 5 time units. Depending on the conditional statements, each job of $\tau_i$ can take any of three execution flows. A GFP schedule on $m = 3$ cores for every execution flow is depicted in Fig. 5.3. In this case, the largest response time of $\tau_i$ is equal to 11 and given by the second execution flow, whereas $\tau_{seq}$ suffers the most interference from the first execution flow resulting in a response time of 9 time units. This example suggests that the worst-case self- and inter-task interference may reside in different execution flows. Importantly, the conclusions derived above do not hold if the tasks are GFP scheduled on $m = 2$ cores instead. As illustrated in Fig. 5.4, a reduction in the number of cores led the response times of $\tau_i$ and $\tau_{seq}$ (12 and 11 time units, respectively) to be maximized by the third execution flow. Note that adding more interfering tasks may even reverse the worst-case scenario found for a certain configuration.*

Therefore, for a given scheduling algorithm, platform and set of parallel tasks, where each task may take different execution flows at run-time, an exact schedulability test would have to consider every feasible interference scenario between all combinations of execution flows from each task. Clearly, this would lead to a *combinatorial explosion* in the number of scenarios to be considered, which is *prohibitively expensive* in terms of computational time.

To the best of our knowledge, the real-time research community so far has only managed to address this scheduling problem to a limited extent. That is, although current works do not ex-

[1] For ease of understanding, assume that tasks have harmonic periods and synchronous releases.

plicitly deal with conditional execution of subtasks, some works have derived results that may still hold under such circumstances due to the fact that the internal structure of the graphs is completely ignored. For example, the authors of (Li et al., 2013) have derived a capacity augmentation bound based on the workload and the critical path length of a task modeled as a single DAG of subtasks. In the case of a task with multiple execution flows, where each flow is modeled as a separate DAG, identical results can be derived considering the maximum workload and critical path length among all the task's execution flows; the pessimism in the analysis itself reduces the number of scenarios to be considered. Nevertheless, the control-flow information needs to be integrated in the task model in order to compute the metrics of interest.

## 5.3 Model extensions

In order to cope with the conditional execution of DAG tasks, we extend the base model presented in Section 1.3 to a multi-DAG model, which integrates control-flow information in the form of distinct execution flows. We still consider a set of $n$ sporadic tasks with constrained deadlines to be scheduled on $m$ identical cores. While no restrictions on the priority assignment policy are imposed, we target only work-conserving scheduling algorithms.

The main difference is that we replace the DAG parameter $G_i$ by a multi-DAG parameter $\mathcal{F}_i$, thus $\tau_i = (\mathcal{F}_i, T_i, D_i)$. The multi-DAG $\mathcal{F}_i = \{F_{i,1}, F_{i,2}, \ldots, F_{i,n_i}\}$ denotes the set of all $n_i$[2] feasible execution flows of a a conditional DAG task $\tau_i$. Recall that an execution flow corresponds to a distinct path taken by a job throughout $\tau_i$'s code during its execution, which may vary from job to job due to the conditional statements within $\tau_i$.

Each execution flow $F_{i,k} \in \mathcal{F}_i$ corresponds to a separate DAG of subtasks, which models the non-conditional parallel structure of $\tau_i$ when $F_{i,k}$ is taken at run-time. That is, each job of $\tau_i$ behaves as a single DAG $F_{i,k}$ that has to be fully executed, although multiple jobs may follow different execution flows (as opposed to a non-conditional parallel task $\tau_h$ where every job executes according to $G_h$). Formally, $F_{i,k} = \langle \mathcal{V}_{i,k}, \mathcal{E}_{i,k} \rangle$ , where $\mathcal{V}_{i,k}$ is a set of $n_{i,k}$ subtasks and $\mathcal{E}_{i,k}$ is a set of directed edges. For ease of understanding, we assume that every subtask $v_j \in \mathcal{V}_{i,k}$ executes for *exactly* $C_j$ time units (we will further discuss this assumption in Section 5.6.3).

We further define two functions to quantify the critical path length and workload of each execution flow. Let $\lambda$ denote a path in a DAG $F_{i,k}$.

**Definition 13** (Critical path length of an execution flow)**.** *The critical path length* $\mathrm{CP}(F_{i,k})$ *of an execution flow* $F_{i,k}$ *of a task* $\tau_i$ *is defined as the maximum cumulative execution requirement of all the subtasks belonging to any longest path of* $F_{i,k}$*. That is,*

$$\mathrm{CP}(F_{i,k}) \stackrel{\text{def}}{=} \max_{\lambda \in F_{i,k}} \sum_{v_j \in \lambda} C_j$$

[2]Note that $n_i$ is redefined as the number of feasible execution flows of task $\tau_i$. Previously, it was used to denote the number of subtasks of $G_i$.

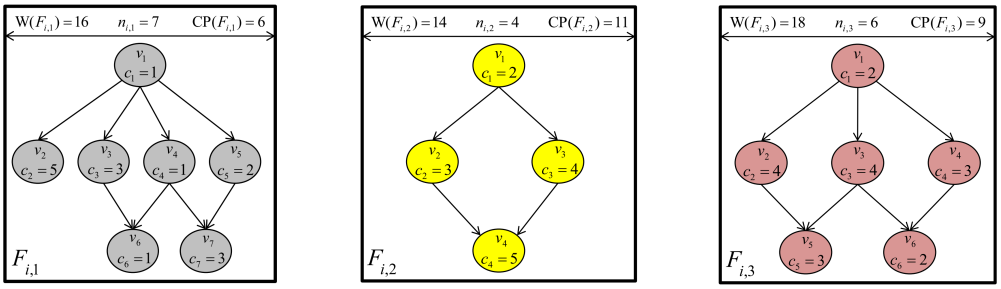



Figure 5.5: Example of a task $\tau_i$ with $\mathcal{F}_i = \{F_{i,1}, F_{i,2}, F_{i,3}\}, T_i = 30, D_i = 20$. Note that subtask $v_j$ denotes the $j$'th node in each execution flow, thus subtask $v_1$ of these three flows may or may not refer to the same subtask of $\tau_i$.

**Definition 14** (Workload of an execution flow)**.** *The workload* $\mathrm{W}(F_{i,k})$ *of an execution flow* $F_{i,k}$ *of a task* $\tau_i$ *is defined as the maximum cumulative execution requirement of all the subtasks in* $\mathcal{V}_{i,k}$*. That is,*

$$\mathrm{W}(F_{i,k}) \stackrel{\text{def}}{=} \sum_{v_j \in \mathcal{V}_{i,k}} C_j$$

Given the above definitions, the critical path length and workload of $\tau_i$ correspond to the worst-case values of these two parameters among all $\tau_i$'s execution flows. That is, $L_i = \max_{F_{i,k} \in \mathcal{F}_i} \mathrm{CP}(F_{i,k})$ and $W_i = \max_{F_{i,k} \in \mathcal{F}_i} \mathrm{W}(F_{i,k})$. Note that $L_i$ and $W_i$ may be derived from different executions flows, which highlights one of the challenges facing the schedulability problem.

**Example 16.** *Fig. 5.5 illustrates our multi-DAG model for a conditional parallel task* $\tau_i$ *comprised of three execution flows; meaning that each of* $\tau_i$*'s jobs executes in accordance with one and only one of the three DAGs* $F_{i,k}$*,* $k \in \{1,2,3\}$*. The maximum critical path length is found on the second execution flow with* $L_i = \mathrm{CP}(F_{i,2}) = 11$*, whereas the maximum workload is obtained by the third execution flow with* $W_i = \mathrm{W}(F_{i,3}) = 18$*. The first execution flow yields both the maximum parallelism and number of subtasks.*

Instead of deriving schedulability analysis for a set of conditional DAG tasks modeled as multi-DAGs, the objective of the work conducted in this chapter is to develop a transformation technique that bridges the gap between existing analysis and such general task model. In this sense, next section shows how to transform each $F_{i,k}$ in a synchronous DAG of servers and introduce a mapping rule to arbitrate the assignment of subtasks to servers at run-time. Recall that servers are the entities to be scheduled on the cores (Baruah et al., 2002). Each server has a pre-defined cpu-budget to be "consumed" through the execution of ready subtasks, every time a server is granted a core. A ready subtask cannot execute within a server if its budget is exhausted. Moreover, a DAG of servers only executes subtasks of the task from which it was derived, and each server is released only if its precedence constraints have been satisfied.

We also define some properties to assert the correctness of that transformation, setting this way the basis for Section 5.5 in which we will present a method that merges all the DAGs of servers defined for every execution flow $F_{i,k}$ into a single synchronous DAG of servers for each task $\tau_i$. Together with the mapping rule, the servers will be able to provide enough cpu-budget to the

subtasks released by each job of $\tau_i$ irrespective of the execution flow taken, while respecting every precedence constraint. As a result, efficient schedulability analysis can then be safely performed over the resulting DAG of servers (i.e., one for each task).

## 5.4 Per-flow server graph

For each execution flow $F_{i,k}$ of every task $\tau_i$, we derive a synchronous DAG of servers referred to as *synchronous server graph* (SSG) and is denoted by $F_{i,k}^{\text{SSG}}$. Formally, we define an SSG as follows:

**Definition 15** (SSG)**.** *A Synchronous Server Graph is a synchronous DAG of nodes (here the nodes are the servers) organized as a set $\{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ of $r$ segments. Each segment $\sigma_\ell$ with $\ell \in [1, r]$ is characterized by a pair $\langle b_\ell, q_\ell \rangle$, where $q_\ell$ is the number of servers in $\sigma_\ell$ and $b_\ell$ is the cpu-budget associated to each of these $q_\ell$ servers. Directed edges exist only between nodes of adjacent segments. Specifically, every node within a segment is connected to every node of the next segment (if any).*

Informally, the purpose of the method developed in this section is to be able to represent each execution flow of a given task $\tau_i$ as a synchronous DAG of servers such that, when $\tau_i$ takes one of its execution flows $F_{i,k}$ at run-time, the corresponding SSG $F_{i,k}^{\text{SSG}}$ provides the required budget to finish the execution of all the subtasks of $F_{i,k}$ without violating any precedence constraint. This is an *intermediate* step in our approach; in the next section we develop a second step (based on this first one) that assigns a single synchronous DAG to *each task*, rather than one SSG for each flow.

The mechanism to handle these per-flow SSG at run-time works as follows: each segment of an SSG is a collection of servers whose budget is used to execute exclusively the ready subtasks of the execution flow from which the SSG has been derived. The servers are the entities to compete for, and to be scheduled on, the $m$ cores by the scheduling algorithm of the operating system. Each time a server is granted a core, its budget is used to execute a ready subtask. Each time a job of a task is released (and thus executes one of its execution flows), the first segment of the corresponding SSG "releases" all its servers, in the sense that they become ready to provide budget to the subtasks of that particular flow. Then, each of the subsequent segments releases all its servers only after all the servers from the previous segment have exhausted their budgets. That is, servers belonging to a segment $\sigma_\ell$ are allowed to provide cpu-budget to the subtasks of the dedicated execution flow only when all the servers from segment $\sigma_{\ell-1}$ have exhausted their budget. Since at some point in time we may have several subtasks from the same execution flow that are ready-to-execute and several servers in the corresponding SSG that are ready to provide budget, there must be a mapping rule to define which subtask is granted budget from which server.

Firstly, we state the generic conditions that assert the *validity* of an SSG toward a given execution flow through Property 3. Then, we define a simple, yet efficient, mapping rule which is used throughout the work to arbitrate the assignment of ready subtasks to servers. From that point onward, every time we refer to a valid SSG it implies that the mapping rule given by Definition 16

is enforced. Finally, we present an algorithm to construct an SSG for each execution flow and prove its correctness.

**Property 3** (Validity)**.** *For a platform $\pi$, a scheduling algorithm A, a mapping rule R, and an execution flow $F_{i,k}$ of a task $\tau_i$, an SSG $F_{i,k}^{\text{SSG}}$ is said to be valid for $F_{i,k}$ according to R if and only if for any schedule of the servers of $F_{i,k}^{\text{SSG}}$ produced by A on $\pi$, at run-time all the nodes of $F_{i,k}$ are guaranteed to be mapped by R to the server nodes of $F_{i,k}^{\text{SSG}}$ in such a way that (1) all the dependencies between the nodes of $F_{i,k}$ are satisfied, and (2) $F_{i,k}$ receives the required budget to execute all its nodes.*

**Definition 16** (Mapping rule)**.** *Let $F_{i,k}$ be an execution flow of a task $\tau_i$ and let $F_{i,k}^{\text{SSG}}$ be the corresponding SSG constructed using Algorithm 5. A server $s_{\ell,x} \in \sigma_\ell \subseteq F_{i,k}^{\text{SSG}}$, with $x \in [1, q_\ell]$, can execute a ready subtask $v_j \in \mathcal{V}_{i,k}$ if and only if $v_j$ has not been executed by a server $s_{\ell,y} \neq s_{\ell,x}$ such that $s_{\ell,y} \in \sigma_\ell$ as well.*

**Algorithm 5:** generateSSG($F_{i,k}$)

**Input** : $F_{i,k}$ - An execution flow of task $\tau_i$
**Output**: $F_{i,k}^{\text{SSG}}$ - An SSG for $F_{i,k}$

1 $F_{i,k}^{\text{SSG}} \leftarrow \emptyset$ ;
2 **while** $\mathcal{V}_{i,k} \neq \emptyset$ **do**
3 $\quad S^{\text{curr}} \leftarrow \{v_j \in \mathcal{V}_{i,k} | \text{pred}(v_j) = \emptyset\}$;
4 $\quad C^{\min} \leftarrow \min\{C_j | v_j \in S^{\text{curr}}\}$ ;
5 $\quad F_{i,k}^{\text{SSG}} \leftarrow F_{i,k}^{\text{SSG}} \otimes \{\langle C^{\min}, |S^{\text{curr}}| \rangle\}$ ;
6 $\quad$ **foreach** $v_j \in S^{\text{curr}}$ **do**
7 $\quad\quad C_j \leftarrow C_j - C^{\min}$;
8 $\quad\quad$ **if** $C_j = 0$ **then**
9 $\quad\quad\quad \mathcal{V}_{i,k} \leftarrow \mathcal{V}_{i,k} \setminus \{v_j\}$ ;
10 $\quad\quad\quad \mathcal{E}_{i,k} \leftarrow \mathcal{E}_{i,k} \setminus \{(v_j, *)\}$ ;
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 **end**
14 **return** $F_{i,k}^{\text{SSG}}$;

The pseudo-code of the SSG creation algorithm is shown in Algorithm 5, whereas Fig. 5.6 depicts the resulting SSG[3] for the execution flow $F_{i,1}$ illustrated in Fig. 5.5. This algorithm takes an execution flow $F_{i,k}$ of task $\tau_i$ as input and outputs an SSG $F_{i,k}^{\text{SSG}}$ for that flow, working as follows. The algorithm traverses the DAG $F_{i,k}$ by starting at its unique source node (first iteration at line 3). At each iteration in the while loop, the algorithm adds a new segment at the end of $F_{i,k}^{\text{SSG}}$ (line 5). The addition is represented by the operator $\otimes\langle b, q\rangle$ which appends a segment of $q$ servers, each with a budget of $b$. This new segment has as many servers as there are subtasks with no predecessor(s) in $\mathcal{V}_{i,k}$ (i.e., ready subtasks) and each of these servers is assigned a budget equal to the minimum execution requirement among these sub-tasks (computed at line 4). The algorithm then proceeds by updating the DAG $F_{i,k}$ and "simulating" the execution of its subtasks within the

[3] As a coincidence, in this example, all segments of the SSG have unitary budgets although it may not be the case in general.

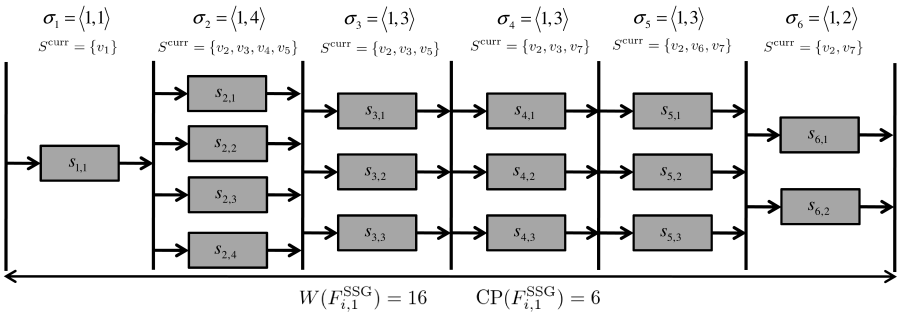



Figure 5.6: SSG $F_{i,1}^{\text{SSG}}$ obtained by running Algorithm 5 with input $F_{i,1}$.

created servers. That is, for each subtask with no predecessor, its execution time is decreased by $C^{\min}$ time units (line 7), thus reflecting its execution within that dedicated server. The number of servers per segment is basically tied to the number of subtasks that are guaranteed to be ready at that point in time, at run-time. Subtasks reaching zero execution requirement are removed from the input DAG $F_{i,k}$, as well as their respective outgoing edges (lines 8–10). Algorithm 5 is guaranteed to terminate as $\mathcal{V}_{i,k}$ eventually becomes empty.

We now prove that the SSG output by Algorithm 5 is always valid (see Property 3) for its input execution flow.

**Lemma 14.** $\mathrm{W}(F_{i,k}^{\text{SSG}}) = \mathrm{W}(F_{i,k})$.

*Proof.* At each iteration in the while loop in Algorithm 5, $|S^{\text{curr}}| \times C^{\min}$ units of workload are added to $F_{i,k}^{\text{SSG}}$ (at lines 5), and the same amount is then iteratively subtracted from $F_{i,k}$ (lines 6 and 7). From this and the while loop termination condition, the claim trivially holds. □

**Theorem 11.** *The SSG $F_{i,k}^{\text{SSG}}$, obtained by running Algorithm 5 with input $F_{i,k}$, is valid for the execution flow $F_{i,k}$.*

*Proof.* According to the validity property, we need to show that (a) all the dependencies in $F_{i,k}$ are preserved and (b) $F_{i,k}$ is provided the required budget to finish the execution of all its subtasks.

**Proof of (a):** It can be easily seen that all the precedence constraints in $F_{i,k}$ are preserved by construction since the servers are created for only those subtasks which are ready to execute (lines 3–5 in Algorithm 5), and the mapping rule ensures that no subtask is assigned to more than one server within the same segment.

**Proof of (b):** Let us recall the run-time management mechanism of an SSG: all the servers within a segment of an SSG become "ready" to provide budgets only when all the servers from the previous segment have exhausted their budgets. Given this run-time mechanism, we prove by induction on the number of segments that no budget provided by the servers is wasted, i.e., all the servers of each segment use their entire budget to execute subtasks of $F_{i,k}$ that are ready-to-execute. Therefore, since at the end no budget is wasted and the total amount of budget provided by the SSG is equal to the workload of $F_{i,k}$ (from Lemma 14) the claim holds true. The detailed proof follows.

**Base case.** In the first iteration of the while loop, there is only one subtask with no predecessor (remember that there is only one source to any execution flow as it follows the base DAG semantics formalized in Section 1.3) and thus only one server is created and added to $F_{i,k}^{\text{SSG}}$ at line 5. This server has a budget $C^{\min}$ equal to the WCET $C_j$ of that subtask (line 4), and at runtime this single server will provide budget to that single subtask as soon as it is released, i.e., when $F_{i,k}$ is taken for execution. Hence, this first subtask will execute entirely within the budget of that first server and no budget is wasted in this first segment. In addition, the algorithm "simulates" the completion of this first subtask as it is removed from $F_{i,k}$ at line 9 and 10, implying that at the next iteration $\mathcal{V}_{i,k}$ will contain only the subtasks that have not completed yet.

**Inductive step.** Assume that at run-time, the $\ell$'th segment just released all its servers and no budget has been wasted by the servers of all the previous segments. Also (as mentioned above), at the $\ell$'th iteration of the while loop, $\mathcal{V}_{i,k}$ contains only the subtasks that have not completed yet and $S^{\text{curr}}$ therefore contains the set of all the uncompleted subtasks that are ready-to-execute at the release of the servers of the $\ell$'th segment. As seen in line 5, Algorithm 5 creates in segment $\sigma_\ell$ as many servers as there are ready subtasks, i.e., $|S^{\text{curr}}|$ servers are created in $\sigma_\ell$. Each of these $|S^{\text{curr}}|$ servers is assigned a budget of $C^{\min}$, which corresponds to the minimum remaining WCET of all the ready subtasks. At run-time, the mapping rule guarantees that each one of the $|S^{\text{curr}}|$ ready subtasks will be allocated to one (and only one) of the $|S^{\text{curr}}|$ servers, and they will all execute for $C^{\min}$ time units, which is "simulated" at lines 6 and 7 of Algorithm 5. Here again, no budget is wasted in the $\ell$'th segment and since the tasks that complete at the end of this segment are removed from $F_{i,k}$ at lines 8–10, at the next iteration of the while loop, $\mathcal{V}_{i,k}$ will once more contain only the uncompleted subtasks.

The algorithm terminates when $\mathcal{V}_{i,k}$ is empty, which means that there are no more uncompleted subtasks. In every segment, no budget has been wasted and since we have $\text{W}(F_{i,k}^{\text{SSG}}) = \text{W}(F_{i,k})$ by Lemma 14, it holds that all the subtasks of $F_{i,k}$ have been executed entirely. □

Note that upon applying Algorithm 5 to each execution flow of a task $\tau_i$, we obtain a set of SSGs for that task, where each SSG is defined and proven valid for one of $\tau_i$'s execution flow. With that, we now describe how to construct a single synchronous DAG of servers for each *task* which accommodates all of its execution flows through its SSGs.

## 5.5 Per-task server graph

The algorithm presented in the previous section has paved the way for the second and final step of our approach. In this section, we present how to merge all the SSGs $F_{i,k}^{\text{SSG}}$, created for a task $\tau_i$ , into a single synchronous DAG of servers, called *"global synchronous server graph"* (GSSG) and denoted by $\mathcal{F}_i^{\text{GSSG}}$. Such a GSSG must ensure that every execution flow $F_{i,k}$ of task $\tau_i$ can be entirely executed within its servers, i.e., $\mathcal{F}_i^{\text{GSSG}}$ must be valid for every execution flow $F_{i,k}$ of $\tau_i$. With that, state-of-the-art schedulability techniques and analysis that have been developed for DAG tasks can be straightforwardly applied over a new task set comprised of GSSGs (one

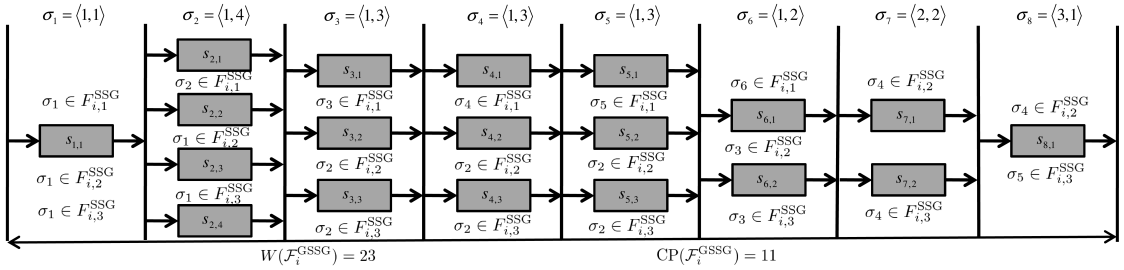



Figure 5.7: GSSG $\mathcal{F}_i^{\text{GSSG}}$ obtained by running Algorithm 6 with input $F_{i,k}^{\text{SSG}} \in L$ where $k \in [1,2,3]$. The notation $\sigma_\ell \in F_{i,k}^{\text{SSG}}$ shows which segments of the SSGs are mapped to the different segments of $\mathcal{F}_i^{\text{GSSG}}$.

derived for each task), as our approach transforms the multi-DAG model into a common parallel synchronous task model. That is, for each task $\tau_i$ , its set $\mathcal{F}_i$ of DAGs is executed within a single synchronous DAG $\mathcal{F}_i^{\text{GSSG}}$ (period and deadline are inherited). Note that a GSSG is still an SSG (as defined in Definition 15) and therefore all the definitions and properties presented in Section 5.4 remain in effect.

**Algorithm 6:** generateGSSG($L$)

**Input** : $L$ - A list with a valid SSG $F_{i,k}^{\text{SSG}}$ for each execution flow $F_{i,k}$ of task $\tau_i$
**Output**: $\mathcal{F}_i^{\text{GSSG}}$ - A GSSG for task $\tau_i$

1 $\mathcal{F}_i^{\text{GSSG}} \leftarrow \emptyset$ ;
2 **while** $L \neq \emptyset$ **do**
3 $\quad B^{\min} \leftarrow \infty$ ;
4 $\quad Q^{\max} \leftarrow 0$ ;
5 $\quad$ **foreach** $F_{i,k}^{\text{SSG}} \in L$ **do**
6 $\quad\quad \sigma_{i,k}^{\text{curr}} \leftarrow \left\{ \sigma_\ell \in F_{i,k}^{\text{SSG}} \mid \text{pred}(\sigma_\ell) = \emptyset \right\}$;
7 $\quad\quad$ **if** $B^{\min} > b_{i,k}^{\text{curr}}$ **then** $B^{\min} \leftarrow b_{i,k}^{\text{curr}}$;
8 $\quad\quad$ **if** $Q^{\max} < q_{i,k}^{\text{curr}}$ **then** $Q^{\max} \leftarrow q_{i,k}^{\text{curr}}$;
9 $\quad$ **end**
10 $\quad \mathcal{F}_i^{\text{GSSG}} \leftarrow \mathcal{F}_i^{\text{GSSG}} \otimes \left\{ \langle B^{\min}, Q^{\max} \rangle \right\}$ ;
11 $\quad$ **foreach** $F_{i,k}^{\text{SSG}} \in L$ **do**
12 $\quad\quad$ **if** $b_{i,k}^{\text{curr}} - B^{\min} = 0$ **then** $F_{i,k}^{\text{SSG}} \leftarrow F_{i,k}^{\text{SSG}} \setminus \sigma_{i,k}^{\text{curr}}$ ;
13 $\quad\quad$ **else** $\sigma_{i,k}^{\text{curr}} \leftarrow (b_{i,k}^{\text{curr}} - B^{\min}, q_{i,k}^{\text{curr}})$ ;
14 $\quad\quad$ **if** $F_{i,k}^{\text{SSG}} = \emptyset$ **then** $L \leftarrow L \setminus \left\{ F_{i,k}^{\text{SSG}} \right\}$ ;
15 $\quad$ **end**
16 **end**
17 **return** $\mathcal{F}_i^{\text{GSSG}}$ ;

Algorithm 6 shows the pseudo-code of the GSSG creation algorithm, whereas Fig. 5.7 depicts the resulting GSSG for task $\tau_i$ once provided its execution flows have been converted into SSGs by Algorithm 5, as exemplified in Fig. 5.6. For a given task $\tau_i$, this algorithm takes as input the SSGs $F_{i,k}^{\text{SSG}}$ derived by Algorithm 5 for each of its execution flows $F_{i,k}$, and outputs a unique GSSG $\mathcal{F}_i^{\text{GSSG}}$ that can accommodate all the referred flows, working as follows. The algorithm keeps on iterating in the while loop at line 2 until the list $L$ of SSGs given as input is empty. At each of these iterations, a new segment of servers is added to the output GSSG (line 10). This new segment is

composed of $Q^{\max}$ servers, each with a budget of $B^{\min}$. These two parameters $B^{\min}$ and $Q^{\max}$ are computed in lines 3–9. Specifically, line 6 records in $\sigma_{i,k}^{\text{curr}}$ the segment from every input $F_{i,k}^{\text{SSG}}$ that has no predecessors, therefore implicitly providing the pair $(b_{i,k}^{\text{curr}}, q_{i,k}^{\text{curr}})$. Here $b_{i,k}^{\text{curr}}$ is the remaining budget of any of the $q_{i,k}^{\text{curr}}$ servers. Note that by the definition of an SSG (see Definition 15) all the servers within a segment will always have the same initial budget and only the servers of one segment are ready at any time instant. On line 7, $B^{\min}$ is set to the minimum remaining budget of all these servers in the "not-yet-processed" segment ($\sigma_{i,k}^{\text{curr}}$) of all SSGs and $Q^{\max}$ records the maximum number of servers in all these segments.

The algorithm then updates all the SSGs (lines 11–13). For every SSG $F_{i,k}^{\text{SSG}}$, if the remaining budget of all the servers in the "not-yet-processed" segment $\sigma_{i,k}^{\text{curr}}$ is equal to $B^{\min}$ then all the servers of that segment are now considered as "processed", as $Q^{\max} \geq q_{i,k}^{\text{curr}}$ servers of budget $B^{\min}$ have been added to the output GSSG at line 10. All these servers are thus removed from their SSG $F_{i,k}^{\text{SSG}}$, and here we can see that the next iteration of the foreach loop (lines 5–9) will again give at line 6 a $\sigma_{i,k}^{\text{curr}}$ for each $F_{i,k}^{\text{SSG}}$ equal to its next segment. Otherwise, if the remaining budget of all the servers in the "not-yet-processed" segment $\sigma_{i,k}^{\text{curr}}$ is higher than $B^{\min}$ (it cannot be lower by definition of $B^{\min}$ at line 7), then these remaining budgets are simply decremented by $B^{\min}$ units of workload. At line 14, if all the servers have been removed from $F_{i,k}^{\text{SSG}}$ then this SSG is removed from the list $L$.

We now prove that Algorithm 6 produces a valid GSSG for task $\tau_i$. To do so, we define two operations called *splitting* and *expanding* that transform an input SSG derived from Algorithm 5 into another SSG, and we show that both operations preserve the validity of the input SSG. Then, we prove in Theorem 12 that the output GSSG $\mathcal{F}_i^{\text{GSSG}}$ from Algorithm 6 can always be obtained from any of its input SSGs $F_{i,k}^{\text{SSG}}$ by applying a sequence of splitting and expanding operations and therefore $\mathcal{F}_i^{\text{GSSG}}$ is also valid for all the execution flows $F_{i,k}$ of task $\tau_i$.

**Definition 17** (Splitting operation)**.** *A splitting operation replaces any segment $\sigma_\ell = (b_\ell, q_\ell)$ of an SSG $F_{i,k}^{\text{SSG}}$ with a list of consecutive segments $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r)$ such that $\forall j \in [1,r]$ it holds that $\sigma_\ell^j = \left\langle b_\ell^j, q_\ell^j \right\rangle$, where*

$$q_\ell^j = q_\ell \tag{5.1}$$

$$\sum_{j=1}^{r} b_\ell^j = b_\ell \tag{5.2}$$

**Lemma 15.** *Let $F_{i,k}^{\text{SSG}}$ be a valid SSG derived by Algorithm 5 for a given execution flow $F_{i,k}$ of task $\tau_i$ and let $F_{i,k}^{\text{SSG}'}$ be the SSG obtained from $F_{i,k}^{\text{SSG}}$ after applying an arbitrary series of splitting operations on the segments of $F_{i,k}^{\text{SSG}}$. It holds that $F_{i,k}^{\text{SSG}'}$ is also valid for $F_{i,k}$. That is, the splitting operation preserves the validity of the original SSG $F_{i,k}^{\text{SSG}}$.*

*Proof.* In Theorem 11, the validity of the SSG $F_{i,k}^{\text{SSG}}$ obtained by Algorithm 5 for the execution flow $F_{i,k}$ is proven by showing that (i) $\text{W}(F_{i,k}^{\text{SSG}}) = \text{W}(F_{i,k})$ and (ii) no budget of $F_{i,k}^{\text{SSG}}$ is wasted at run-time. Regarding the workload of the SSG $F_{i,k}^{\text{SSG}'}$, for any segment $\sigma_\ell \in F_{i,k}^{\text{SSG}}$ that has been

broken into a series $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r) \in F_{i,k}^{\mathrm{SSG}'}$, it holds from Eq. 5.2 that $\sum_{j=1}^{r} b_\ell^j = b_\ell$ and since from Eq. 5.1 $q_\ell^j = q_\ell$ for all $j \in [1,r]$, it is easy to see that $\mathrm{W}(F_{i,k}^{\mathrm{SSG}'}) = \mathrm{W}(F_{i,k}^{\mathrm{SSG}}) = \mathrm{W}(F_{i,k})$.

Second, it is shown in Theorem 11 that every segment of $F_{i,k}^{\mathrm{SSG}}$ has as many servers as the number of subtasks that will be ready-to-execute at run-time when the segment will be allowed to provide budget. Therefore, since no budget of $F_{i,k}^{\mathrm{SSG}}$ is wasted, for any segment $\sigma_\ell \in F_{i,k}^{\mathrm{SSG}}$ that has been broken into a series $(\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^r) \in F_{i,k}^{\mathrm{SSG}'}$, there will be exactly $q_\ell$ ready subtasks of remaining WCET $\geq b_\ell$ competing for these $q_\ell$ servers of budget $b_\ell$. From Eq. 5.1 and 5.2, and by the mapping rule, there will also be $q_\ell$ ready subtasks competing for the $q_\ell$ servers of every segment $\sigma_\ell^j$, with $j \in [1,r]$, and such that at every segment $\sigma_\ell^j$ the remaining WCET of these $q_\ell$ ready subtasks will be $\geq b_\ell^j$. As a result, no budget will ever be wasted in these new segments $\sigma_\ell^j$, $j \in [1,r]$. □

**Definition 18** (Expanding operation)**.** *An expanding operation consists in supplying any segment $\sigma_\ell = (b_\ell, q_\ell)$ of an SSG $F_{i,k}^{\mathrm{SSG}}$ with an arbitrary number of extra servers of budget $b_\ell$.*

**Lemma 16.** *Let $F_{i,k}^{\mathrm{SSG}}$ be a valid SSG for a given execution flow $F_{i,k}$ of task $\tau_i$ obtained by Algorithm 5 (and possibly after an arbitrary series of splitting operations). Let $F_{i,k}^{\mathrm{SSG}'}$ be the SSG obtained from $F_{i,k}^{\mathrm{SSG}}$ after applying an arbitrary series of expanding operations on the segments of $F_{i,k}^{\mathrm{SSG}}$. It holds that $F_{i,k}^{\mathrm{SSG}'}$ is also valid for $F_{i,k}$.*

*Proof.* Adding new servers to a segment $\sigma_\ell \in F_{i,k}^{\mathrm{SSG}}$ leads to $\mathrm{W}(F_{i,k}^{\mathrm{SSG}'}) > \mathrm{W}(F_{i,k}^{\mathrm{SSG}})$ and at run-time $q_\ell$ becomes greater than the number of ready subtasks at segment $\sigma_\ell$. However, the mapping rule of Definition 16 enforces that no subtask is assigned to two different servers within a same segment. As a consequence the extra servers will simply be ignored at run-time, and the precedence constraints between the subtasks of $F_{i,k}$ will still be satisfied as the order of execution of the subtasks remains unchanged. In short, the budget of all the servers added to $F_{i,k}^{\mathrm{SSG}'}$ will be entirely wasted and thus the validity is preserved. □

**Theorem 12.** *The GSSG $\mathcal{F}_i^{\mathrm{GSSG}}$ obtained by running Algorithm 6 is valid for task $\tau_i$ as it is valid for every of its execution flows $F_{i,k}$.*

*Proof.* The proof is a direct consequence of Lemmas 15 and 16, and the fact that the output $\mathcal{F}_i^{\mathrm{GSSG}}$ of Algorithm 6 can be obtained from every input $F_{i,k}^{\mathrm{SSG}} \in L$ by applying a series of splitting and expanding operations. Let $F_{i,k}^{\mathrm{SSG}}$ be *any* of the input SSGs and let $\sigma_{i,k}^{\mathrm{curr}}$ be its current segment with no predecessor (line 6). By definition of $B^{\min}$ at line 7, we have $b_{i,k}^{\mathrm{curr}} \geq B^{\min}$ and by definition of $Q^{\max}$ at line 8, we have $q_{i,k}^{\mathrm{curr}} \leq Q^{\max}$. Therefore, the addition of a new segment of $Q^{\max}$ servers of budget $B^{\min}$ at line 10 (alongside the corresponding reduction of $B^{\min}$ units at line 13) can be seen as a splitting operation performed on segment $\sigma_{i,k}^{\mathrm{curr}}$. Also, if $q_{i,k}^{\mathrm{curr}} < Q^{\max}$ then the addition of $Q^{\max}$ servers can be seen as an expanding operation. Finally, note that if $F_{i,k}^{\mathrm{SSG}}$ is not the last SSG to be removed from $L$ at line 14, then the addition of extra segments to the output $\mathcal{F}_i^{\mathrm{GSSG}}$ in the next iterations can also be seen as applying an expanding operation on arbitrary empty segments. □

As a last result, the following corollary holds true from the validity of the GSSGs created by Algorithm 6. The corollary states that the schedulability of the original task set composed of parallel tasks with multiple execution flows (multi-DAGs) can now be assessed by applying any existing schedulability test (the only restriction on the scheduling algorithm is to be work-conserving) over the set of derived GSSGs as long as the test conforms with the final parallel model (i.e., synchronous or general DAGs). The results from (Chwa et al., 2013; Maia et al., 2014) are examples of such tests tailored specifically for synchronous DAGs. Note that the GFP schedulability analysis presented in Chapter 3 also fulfill the requirement and thus can be applied, while both analysis of partitioned scheduling presented in Chapter 4 do not.

**Corollary 3.** *If a valid GSSG $\mathcal{F}_i^{\text{GSSG}}$ is deemed schedulable by a schedulability test of a work-conserving scheduling algorithm A, so does the conditional DAG task $\tau_i$ from which it was derived.*

## 5.6 Pros and cons

Since this dissertation takes the very first steps in addressing the schedulability of a set of DAG tasks with conditional execution, we now provide a discussion on the advantages and disadvantages of the proposed transformation technique.

### 5.6.1 Schedulability

From a schedulability point of view, current scheduling techniques for parallel tasks can be broadly categorized into two categories: decomposition method and direct analysis. In decomposition method, each subtask of a DAG is assigned an intermediate offset and a deadline based on the structure of the DAG. With this, each subtask can be treated as an individual sequential task. The parallel task scheduling problem then reduces to the traditional sequential task schedulability problem on a multiprocessor system, for which there is a plethora of scheduling algorithms and schedulability tests in the literature.

In direct analysis, schedulability conditions are derived directly from the properties of the DAG. Some analysis techniques consider the precedence constraints on the DAG to study the execution requirements at different time instants, whereas others simply rely on the workload and critical path length values to create a synthetic worst-case scenario that upper-bounds the interference. For the latter case, our contribution brings no benefit since we often end up increasing the worst-case workload of the tasks. However, it has been shown in (Qamhieh et al., 2013) that considering the internal structure of a DAG (as we do in this dissertation) may improve the schedulability tests. Hence, for all the other cases which rely on the internal structure of the DAG (including the decomposition methods) our contribution directly enables the application of such schedulability analysis methods to generic real-time parallel applications with conditional execution without having to assume that all the subtasks of every flow must execute. This allows to tighten the schedulability of conditional DAG tasks while reducing the number of interference scenarios that must be checked by orders of magnitude.

### 5.6.2 Optimization

From an optimization viewpoint, it is worth noting that Algorithm 6 is a very simple algorithm that merges a collection of valid SSGs into a single valid GSSG. We chose to present this algorithm in its simplest form for ease of understanding and proving the validity of its output. However, it can be seen that the algorithm can be further improved, particularly with respect to tightening of the GSSG's workload. For example, at each iteration of the while loop, if there is an SSG in $L$, say $F_{i,k}^{\text{SSG}}$, for which the remaining workload is lesser or equal to the remaining critical path length of the SSG with the longest critical path, say $F_{i,h}^{\text{SSG}}$, then the workload of $F_{i,k}^{\text{SSG}}$ can be entirely executed (even in a sequential manner) within the servers that will be created in the next iterations to accommodate the remaining subtasks of $F_{i,h}^{\text{SSG}}$. Therefore, $F_{i,k}^{\text{SSG}}$ can safely be removed from $L$, which may reduce the resulting workload of the output GSSG if $F_{i,k}^{\text{SSG}}$ contributed to $Q^{\max}$ in a next iteration.

Following the above reasoning, there exists a trade-off between the critical path length and the workload of the GSSG output by Algorithm 6, in the sense that it is sometimes possible to reduce its workload by increasing its critical path length, while preserving its validity. Unfortunately, it is hard to predict which modification affects schedulability the most.

Another concern stems from the requirement of explicitly enumerating all the feasible executions flows of each conditional DAG task. As there may exist exponentially many execution flows, the transformation algorithms have prohibitive computational complexity. It is our belief that this problem can be solved by replacing the original multi-DAG model by the conditional DAG model (which integrates special pairs of nodes denoting the conditional opportunities) proposed later in (Melani et al., 2015) and (Baruah et al., 2015), as the the construction of SSGs (this intermediate step is just for clarity of presentation) can be skipped. In this sense, Algorithm 6 would be adapted to traverse and parse a DAG with conditional semantics, iteratively computing the necessary servers according to the logic presented in the previous section.

### 5.6.3 Practicality

Apart from the complexity associated to the run-time mechanisms to manage the servers, our approach benefits from the strong points of server-based techniques: servers have become a standard implementation to guarantee a symmetric temporal isolation between tasks (Abeni and Buttazzo, 2004; Lin et al., 2006) and are also used as fault containers to limit the propagation of errors in case of faulty components.

We shall now briefly discuss the assumption made in Section 5.3, according to which every subtask $v_j$ executes for exactly $C_j$ time units. It is certain that at run-time the subtasks will most of the time execute for less than their WCET and unfortunately our methodology is not sustainable in these circumstances. Nevertheless, the results obtained in this chapter can be made sustainable w.r.t. WCET through the addition of run-time mechanisms. For example, consider the following mechanism: when a subtask $v_j$ completes earlier than indicated by its WCET $C_j$, the mechanism checks the GSSG of the corresponding task and determines in how many segments, say $r$, that

subtask $v_j$ was supposed to complete its execution. Then, the mechanism immediately locks all the successor subtasks of $v_j$ (thus preventing them to become ready) for the next $r$ segments. This way, the system behaves as if all the subtasks execute for their WCET.

## 5.7 Summary

In this chapter, we acknowledge the schedulability problem posed by parallel tasks conditional constructs. Taking a first step in tackling this open challenge, we proposed a multi-DAG model (a generalization of the sporadic DAG model) in which each real-time parallel task is characterized by a collection of execution flows, each of which modeled as a separate DAG. Due to the conditional statement in the task code, at every job release only one of these DAGs is executed but we do not know *a priori* which one. To avoid the pitfall of analyzing the maximum interference between all possible combinations of execution flows for all the tasks, we derive a two-step solution to construct a DAG of servers (GSSG) for each task that can accommodate *all its execution flows*, in the sense that at run-time this GSSG will always provide enough CPU budget to the task from which it is derived so that the task can always execute and complete by its deadline, irrespective of which execution flow it takes during run-time.

We also define a mapping rule to decide which subtask must be assigned to which server at run-time. Each time a job is released, the budgets of all the servers of its corresponding GSSG are replenished and the servers' budgets and the mapping rule are defined such that, *for any execution flow taken by the task at runtime*, every subtask gets enough budget to finish its execution and all the precedence constraints between the subtasks are respected. Therefore, the multi-DAG parameter $\mathcal{F}_i$ assumed in the task model can be replaced for its corresponding GSSG $\mathcal{F}_i^{\text{GSSG}}$, while the period and the deadline remain unchanged.

With this, there is no need to consider every feasible interference scenario between all combinations of execution flows of all the tasks in order to derive a schedulability test based on the internal structure of the tasks, as the single DAG $\mathcal{F}_i^{\text{GSSG}}$ naturally allows to upper-bound the on-core interference that a task $\tau_i$ causes on the other tasks. Moreover, a GSSG is a special case of the synchronous parallel task model, which in turn is a special case of the DAG model. As a result, existing multicore work-conserving scheduling techniques suited for any of these classes of parallel tasks can be leveraged to ascertain the schedulability of a task set comprised of conditional DAG tasks as the ones modeled by a multi-DAG.

# Chapter 6

# Concluding Remarks

Few years ago, the frontier separating the real-time embedded domain from the high-performance computing domain was neat and clearly defined. Nowadays, many contemporary applications no longer find their place in either category as they manifest both strict timing constraints and work-intensive computational demands. The only way forward to cope with such orthogonal requirements is to embrace the parallel execution programming paradigm on the emergent scalable and energy-efficient multi/many-core architectures. However, parallelization adds another dimension to the already challenging problem of multiprocessor real-time scheduling.

In order to tackle this parallelization challenge, the real-time community has recently started to develop new task models, analysis and methodologies, enriching classical schedulability theory with solutions that guarantee the timeliness of parallel task systems. In this dissertation, we considered the sporadic DAG model, where a parallel task consists in a set of concurrent subtasks whose execution has to obey to a set of precedence constraints. Such task model reflects general features of parallelism characteristic of widely used parallel programming models (e.g., OpenMP). Priorities are assigned only at the task-level to further increase the synergy with current run-time environments for parallel workloads. Accordingly, we addressed a set of schedulability analysis problems for multiprocessor systems under classical preemptive scheduling algorithms by exploring the internal structure of the DAGs. Namely, both global and partitioned paradigms, as well as applications with conditional execution.

The first solution, presented in Chapter 3, concerns the problem of scheduling a set of DAG tasks according to GFP. Building on top of the work proposed in (Melani et al., 2017), we introduced a new worst-case scenario to analyze the interference generated by higher priority tasks which captures the execution pattern of their carry-in and carry-out jobs. By taking into account the precedence constraints, we presented innovative techniques to more accurately characterize and upper-bound the worst-case carry-in and carry-out workloads. This allowed us to derive improved response time analysis for constrained deadline DAG tasks, which we then extended to the general case of arbitrary deadlines, where each job may also be subjected to interference from preceding jobs of the same task. Experimental results not only attest the theoretical dominance of the proposed analysis over its state-of-the-art counterpart (in the constrained deadline case), but also

showed that it is robust to multiprocessor systems with increasing number of cores (a weakness of the work in (Melani et al., 2017)) and it substantially tightens the schedulability of DAG tasks on multiprocessor systems for both constrained and arbitrary deadline task sets.

The next two solutions, presented in Chapter 4, relate to the problem of deriving schedulability analysis for a set of partitioned DAG tasks. In this context, partitioning means statically assigning each subtask to a specific core, yet allowing multiple subtasks of the same DAG task to run on different cores. To the best of our knowledge, this is an highly unexplored scheduling paradigm for real-time parallel tasks. First, we introduced a novel response time analysis under PFP scheduling, based on the interference suffered by each path of a DAG, that works for any given mapping. We showed that the inter-task interference exerted on a partitioned DAG task can be estimated by modeling the DAG as a set of self-suspending tasks, and then presented an algorithm to define and order the resulting self-suspending tasks such that the WCRT of the partitioned DAGs can be safely constructed recursively. Since this technique relies on the enhancement of existing uniprocessor RTA for self-suspending tasks, it led us to discover a critical flaw in the related literature (the repercussions of the incorrectness of (Lakshmanan and Rajkumar, 2010) is discussed extensively in (Chen et al., 2016)). To bridge the identified gap, we proposed a RTA for a set of sporadic self-suspending tasks with multiple suspension regions running on a uniprocessor system, based on a MILP formulation, as included in Appendix A. With this particular result, our RTA for PFP scheduling is able to reduce the WCRT of a partitioned parallel task (fork-join tasks in the tested scenarios) 20 to 50 percent in average, in comparison to the state-of-the-art in distributed real-time systems.

Due to the complexity and pessimism in the previous approach, we then proposed a second schedulability analysis for DAG tasks scheduled by EDF under the partitioned paradigm. We developed a partitioning algorithm that maps similar paths of a DAG to the same processor, aiming to minimize the number of cores that guarantees feasibility and to eliminate cross-core dependencies. Thanks to the duplication of key subtasks, all resulting partitions are independent of each other. Thus, the problem of scheduling a set of partitioned DAGs becomes equivalent to the problem of scheduling a set of sequential tasks on multiprocessors in a partitioned manner. Experimental evaluation demonstrated that this new schedulability test is very effective, achieving a relatively high schedulability ratio and outperforming federated scheduling under most configurations. Moreover, its performance is competitive with semi-federated scheduling recently proposed in (Jiang et al., 2017), which requires rather complicated run-time mechanisms and incurs in both migrations and additional preemptions. Performance losses are observed for highly connected DAG tasks and systems with very high utilization due to the downside of duplicating subtasks to guarantee independence per core. Our GFP schedulability test was shown to be superior in the case of constrained deadline task sets. However, we suspect such result would not hold had we considered a more accurate EDF schedulability test (e.g., the demand bound function approximation of (Baruah and Fisher, 2007)) instead of the straightforward density bound. This suggests that subtask-duplication is a promising technique to address the schedulability of partitioned DAG tasks.

The last solution, presented in Chapter 5, deals with the problem of modeling and scheduling a set of DAG tasks with conditional execution. It is expected that industrial applications feature conditional operations that depend on run-time data, leading to workloads whose sizes and compositions vary with the different instances. However, in the DAG model, every job spawns and executes all nodes pertaining to its task. While conditional statements were not a major concern in the case of sequential tasks, we show that they are detrimental for the timing analysis of parallel tasks and thus should be explicitly modeled. Importantly, this finding has opened a new research avenue in real-time systems. As a generalization, we proposed a multi-DAG model where each conditional parallel task is characterized by a set of execution flows, each of which represented as a separate DAG. Due to conditional statements, only one of such execution flows is taken at run-time by a job. We derived a two-step algorithm that constructs a single synchronous DAG of servers for a multi-DAG task and proved that these servers are able to safely and fully execute any of its execution flows. As a result, each multi-DAG task can be modeled by its single DAG of servers, which facilitates in leveraging the existing single-DAG (or more restrictive models) schedulability analysis techniques for analyzing the schedulability of conditional DAG tasks. This method offers a trade-off in terms of analysis accuracy: on one hand, the synchronous DAG of servers provides a less general internal structure that is easier to analyze and can be exploited to reduce pessimism; on the other hand, there is a over-provisioning of resources to abstract the actual interference imposed by the different execution flows.

Considering the above contributions, listed in detail in Section 1.5, we firmly believe that the central proposition of this thesis stated in Section 1.4 is successfully fulfilled. Indeed, we managed to derive a set of schedulability tests for DAG tasks under a range of multiprocessor scheduling problems, while exploring the internal structure of the DAGs, which not only guarantees that every task will meet its deadline but also allows for a more effective use of the platform capacities. We conclude that the outcome of this research work will contribute to advance the state-of-the-art on the design and analysis of real-time systems composed of parallel workloads. Nevertheless, we acknowledge that there is plenty of room for improvements and extensions, as well as open problems. In the following, we discuss future research directions and perspectives.

For GFP scheduling, we plan to better characterize the self-interfering workload as well as the interference generated by body jobs. We believe that most of the pessimism remaining in the analysis is located in those two terms. The self-interference could be improved by assigning distinct fixed priorities to the subtasks of each DAG, which would result in a two-level scheduler. However, this comes at the cost of additional preemptions and run-time modifications. If the improvements are not significant, such approach may not be worthwhile. The other issue is transverse to the concept of "problem window", since all the workload within the window is accounted as interference instead of checking for potential workload executing in parallel with the critical path of the analyzed task. As a first step, we are currently investigating new techniques to more accurately estimate the inter-task interference imposed by sequential tasks on a DAG task. Furthermore, we are considering extending our results to GEDF scheduling. We expect that the poor performance of GEDF reported by the authors of (Melani et al., 2017) may be attenuated when the carry-in and

carry-out interfering workloads are modeled by a workload distribution.

Relatively to partitioned scheduling, we intend to mix our two approaches with the goal of further minimizing the number of cores required for the feasibility of every heavy DAG task. The idea is to restrict the number of duplicated subtasks and model some partitions as self-suspending tasks. Partitions would then be the scheduling entities, and we are primarily interested in assigning fixed-priorities to the partitions so that we can control the interference they are subjected to. Finally, we believe that both the schedulability analysis for GFP and duplication-based PEDF can be extended to the general case of conditional DAGs. For this purpose, we remark that the conditional DAG model proposed in (Baruah et al., 2015; Melani et al., 2015) is more efficient than our multi-DAG model.

# Appendix A

# Uniprocessor Response Time Analysis for Sporadic Self-Suspending Tasks

Many real-time systems include tasks that need to suspend their execution in order to externalize some of their operations or to wait for data, events or shared resources. As presented in Chapter 4, such self-suspending tasks can also be used effectively to model partitioned DAG tasks. Although commonly encountered in real-world systems, study of their worst-case scheduling behavior is still limited due to the complexity of the problem. This chapter considers the fixed-priority scheduling of a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of $n$ sporadic self-suspending tasks running on a uniprocessor system, where each task $\tau_i$ is represented as a interleaved sequence of execution regions and suspension regions: $(C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, \ldots, S_{i,m_i-1}, C_{i,m_i},)$.

We start by discussing a couple of misconceptions about the critical instant for a sporadic self-suspending task when higher priority tasks are sequential, which invalidate a well-accepted claim from an earlier work. Based on a new worst-case scenario, we then present an algorithm to compute the exact WCRT for a self-suspending tasks with one suspension region. As the algorithm becomes rapidly intractable in function of the number of higher priority tasks, we formulate a RTA for a self-suspending task with multiple suspension regions as an optimization problem, which we also extend to the case of multiple self-suspending tasks interfering with each other.

## A.1 Motivation

Real-time tasks often involve processing operations which may take considerable time to finish if executed solely on a generic purpose uniprocessor platform. System designers have been achieving significant improvement in the efficiency of these operations by offloading them to specialized hardware accelerators (e.g., Graphical or Network Processing Units), leaving the main processor available for other tasks. The offloading phases represent suspension delays for the task that initiates them. Suspension delays can also be observed when tasks are synchronizing, exchanging data through communication interfaces, or accessing external shared resources such as I/O devices. All such tasks that may at some point in their execution suspend their computation to wait for exter-

nal data, events or resources are called *self-suspending* tasks. As the systems grow increasingly complex, self-suspending tasks become a promising solution to capture the interactions between different dependent components. An encouraging example is the RTA proposed in Section 4.4 for parallel tasks under partitioned scheduling.

A self-suspending task is composed of a set of *execution regions* interleaved with *suspension regions*. Traditional real-time systems theory (Liu, 2000) has accounted the duration of suspension regions as part of the task worst-case execution time while doing timing analysis. However, if the suspension regions are lengthy, such suspension-oblivious analysis typically become very pessimistic, leading to severe utilization loss and possibly jeopardizing the schedulability of the system. Hence, recent works (Lakshmanan and Rajkumar, 2010; Kim et al., 2013b; Liu and Anderson, 2009, 2012, 2013) have focused on suspension-aware analysis techniques which explicitly model the suspension placement and durations in order to reduce the pessimism on the calculated worst-case interference exerted by higher priority tasks, thereby offering opportunities for a potential schedulability improvement. Nevertheless, suspension-aware analysis are rather complex and sometimes assume the existence of additional operating system facilities (e.g., phase enforcement mechanisms or execution control policies) to ease the analysis and minimize the pessimism while considering the suspension regions.

In this dissertation, we start by studying the timing analysis of self-suspending tasks, under fixed-priority scheduling, assuming that all the interfering tasks are non-self-suspending sporadic tasks. Contrary to what has been claimed in a previous work (Lakshmanan and Rajkumar, 2010), we show that it is not simple to characterize a critical instant even for this limited model. Based on this key observation, we identify the exponential number of scenarios that need to be considered, and then present appropriate methods to compute accurately the WCRT of a self-suspending task under different task systems.

## A.2 Related work

Negative results on the feasibility problem of scheduling *periodic* self-suspending tasks on a uniprocessor have been presented in (Ridouard et al., 2004, 2006). The problem was shown to be NP-Hard in the strong sense even for the special case in which each implicit deadline task may have at most one suspension region. Works presented in Tindell and Clark (1994); Kim et al. (1995); Palencia and Gonzalez Harbour (1998) have proposed schedulability analysis for dependent tasks and thus are applicable for task that self-suspend. However, the pessimism in those analysis leads to significant utilization loss. Recently, new schedulability tests for synchronous self-suspending tasks with harmonic periods have shown to exhibit low utilization loss under RM (Liu et al., 2014).

The sporadic self-suspending task model has received considerable more attention from the real-time community (Audsley and Bletsas, 2004; Bletsas and Audsley, 2005; Lakshmanan and Rajkumar, 2010; Kim et al., 2013b). In (Lakshmanan and Rajkumar, 2010), authors attempted to characterize the critical instant for self-suspending tasks with respect to the interference exerted by

higher priority non-self-suspending tasks. It builds on the fact that sporadic tasks may delay their job releases, to prove that an *exact* characterization of the worst-case scheduling scenario leading to the WCRT of the self-suspending task is simpler to achieve in comparison to the periodic case. It therefore became a common belief that the WCRT of self-suspending task co-running with sporadic non-self-suspending tasks can be obtained in pseudo-polynomial time. However, as we show in Section A.4, the worst-case release pattern (also referred to as critical instant) is more complex to identify than what was claimed in (Lakshmanan and Rajkumar, 2010), deeming the results in that paper incorrect.

Still on sporadic tasks, early research (Audsley and Bletsas, 2004; Bletsas and Audsley, 2005) considered fixed-priority scheduling of limited parallel systems, where parts of a process are executed in parallel in software and hardware, and therefore can be modeled as a set of sporadic self-suspending tasks. The authors introduced the notion of synthetic worst-case execution distribution for higher priority tasks and derived upper-bounds on WCRTs. Response time analysis for a segment-fixed priority scheduling scheme, which assigns different priorities to each computing segment and enforces phase offsets to predict the different segment's releases, were developed in (Kim et al., 2013b). In the same paper, the effectiveness of RM scheduling for periodic self-suspending tasks with specific properties was demonstrated.

Lately, there has also been relevant work on global scheduling self-suspending tasks atop multiprocessors, in particular for soft real-time task systems (Liu and Anderson, 2009, 2012, 2013). In (Liu and Anderson, 2013), the first suspension-aware schedulability analysis for sporadic self-suspending tasks in a multicore hard-real setting was presented.

## A.3 System model

Consider a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ constrained deadline self-suspending sporadic tasks scheduled on a single processor. Each task $\tau_i$ releases a (potentially infinite) sequence of *jobs*, with the first job released at any time during the system execution and subsequent jobs released *at least* $T_i$ time units apart. Each job released by $\tau_i$ has to complete its execution within $D_i \leq T_i$ time units from its release. A self-suspending task $\tau_i$ consists of $m_i \geq 1$ *execution regions* and $m_i - 1$ *suspension regions* such that any two consecutive execution regions are separated by a suspension region as shown in Fig. A.1.

Formally, each task $\tau_i$ is characterized as $\tau_i \stackrel{\text{def}}{=} \langle (C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, \ldots, S_{i,m_i-1}, C_{i,m_i},), D_i, T_i \rangle$ where (i) $C_{i,j}$ denotes the worst-case execution time of the $j^{th}$ execution region; (ii) $S_{i,j}$ denotes the worst-case duration of the $j^{th}$ suspension region; (iii) $m_i$ denotes the number of execution regions separated by $m_i - 1$ suspension regions; (iv) $D_i \leq T_i$ denotes the deadline before which all the execution regions need to finish their execution and (v) $T_i$ denotes the minimum inter-arrival time between two successive jobs of $\tau_i$. We call *non-self-suspending task*, a task with no suspension region. A non-self-suspending task $\tau_k$ is represented as: $\tau_k \stackrel{\text{def}}{=} \langle (C_{k,1}), D_k, T_k \rangle$.

We assume that a fixed-priority scheduling policy is used to schedule the tasks on the processor. For convenience, we denote by $\tau_{i,j}$ the $j^{th}$ execution region of task $\tau_i$, and the overall

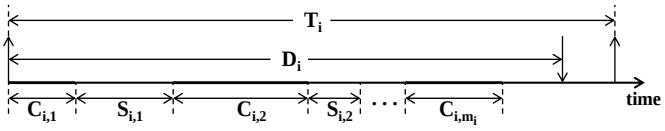


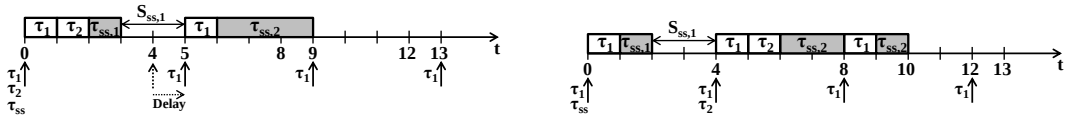

Figure A.1: Constrained deadline self-suspending sporadic task.

worst-case execution time of $\tau_i$ is defined as $C_i \stackrel{\text{def}}{=} \sum_{j=1}^{m_i} C_{i,j}$. The execution region $\tau_{i,1}$ of a job of task $\tau_i$ becomes ready for execution (also referred to as the *arrival time* of $\tau_{i,1}$ denoted by $a_{i,1}$) as soon as a job of task $\tau_i$ is released. The response time $R_{i,1}$ of the execution region $\tau_{i,1}$ is the difference between its *completion time* (denoted by $f_{i,1}$) and the arrival time of the job; formally, $R_{i,1} = f_{i,1} - a_{i,1}$. For $2 \leq j \leq m_i$, the execution region $\tau_{i,j}$ of a job of task $\tau_i$ becomes ready for execution at time $a_{i,j} \stackrel{\text{def}}{=} f_{i,j-1} + s_{i,j-1}$ (where $s_{i,j-1} \leq S_{i,j-1}$ is the self-suspending time of the $(j-1)^{\text{th}}$ suspension region of the job of $\tau_i$) and its response time $R_{i,j}$ is given by the difference between its completion time and its arrival time; formally, $R_{i,j} = f_{i,j} - a_{i,j}$. The response time $R_i$ of a job of task $\tau_i$ is the sum of the response times of all its execution regions and the total duration of all its suspension regions, that is, $R_i \stackrel{\text{def}}{=} \sum_{j=1}^{m_i} R_{i,j} + \sum_{j=1}^{m_i-1} s_{i,j}$. Finally, the *worst-case response time* $\text{WCRT}_i$ *of a task* $\tau_i$ is defined as the largest response time that any job of $\tau_i$ may ever experience.

In Sections A.4 to A.6, we consider the case in which the task set $\tau$ has only one self-suspending task and all the other tasks are non-self-suspending. The self-suspending task is denoted $\tau_{\text{ss}}$ and has the lowest priority, i.e., all the non-self-suspending tasks in $\tau$ have a higher priority than $\tau_{\text{ss}}$. We denote the set of higher priority non-self-suspending tasks as $\text{hp}(\tau_{ss})$. The restriction of having only one self-suspending task in the taskset is relaxed in Section A.7.

## A.4 Misconceptions in the schedulability analysis of self-suspending tasks

From this section onwards, it is assumed that there is only one self-suspending task $\tau_{\text{ss}}$ in the taskset $\tau$. This task has the lowest priority and suffers interference from a set $\text{hp}(\tau_{ss})$ of higher priority non-self-suspending tasks. The worst-case response time analysis of such a system was studied in a previous work (Lakshmanan and Rajkumar, 2010) and was deemed solved. However, in this section, we prove that such work was based on a couple of wrong observations (which have established themselves as facts over the years), namely (i) that the worst-case interference suffered by $\tau_{\text{ss}}$ is generated when all the higher priority tasks are released synchronously with $\tau_{\text{ss}}$, and (ii) that releasing the jobs of higher priority tasks as often as possible (respecting the minimum inter-arrival times) in each execution region maximizes the overall interference on the self-suspending task. Unfortunately, however intuitive these observations may seem, they are incorrect and have led to flawed analysis in that work (Lakshmanan and Rajkumar, 2010).




(a) Scenario 1. Response-time analysis when the job release pattern *is* $\Phi_{ss}$.

(b) Scenario 2. Response-time analysis when the job release pattern *is not* $\Phi_{ss}$.

Figure A.2: Counter-example to $\Phi_{ss}$ being the critical instant of $\tau_{ss}$.

### A.4.1 On the synchronous release with the first execution region

The worst-case response time analysis is based on the notion of *critical instant*. The critical instant for a task $\tau_i$ is defined as an instant at which a request for that task will have the largest response time. Since the response time of a task is dependent on the higher priority tasks, a critical instant for a task $\tau_i$ is generally concerned with the release pattern of higher priority tasks.

In (Lakshmanan and Rajkumar, 2010), Lakshmanan et. al. argue that the release pattern $\Phi_{ss}$ is a critical instant for a self-suspending task $\tau_{ss}$, where $\Phi_{ss}$ is defined as follows:

- every higher priority non-self-suspending task $\tau_h \stackrel{\text{def}}{=} \langle (C_h), D_h, T_h \rangle$ is released simultaneously with $\tau_{ss}$;
- jobs of $\tau_h$ eligible to be released during any $j^{\text{th}}$ $(1 \leq j < m_i)$ suspension region of $\tau_{ss}$ are delayed to be aligned with the release of the subsequent $(j+1)^{\text{th}}$ execution region of $\tau_{ss}$; and
- all remaining jobs of $\tau_h$ are released every $T_h$.

We prove with a counter-example that $\Phi_{ss}$ is not a critical instant for a self-suspending task $\tau_{ss}$.

**Example 17.** *Consider a task set $\tau = \{\tau_1, \tau_2, \tau_{ss}\}$ of three constrained deadline sporadic tasks scheduled on a single processor — $\tau_1$ and $\tau_2$ are non-self-suspending tasks and $\tau_{ss}$ is a self-suspending task. Let the characteristics of these tasks be as follows: $\tau_1 \stackrel{\text{def}}{=} \langle (1), 4, 4 \rangle$; $\tau_2 \stackrel{\text{def}}{=} \langle (1), 100, 100 \rangle$ and $\tau_{ss} \stackrel{\text{def}}{=} \langle (1,2,3), 1000, 1000 \rangle$. Let the priorities of the tasks be assigned using the Rate Monotonic policy (i.e., smaller the period, higher the priority); this implies that task $\tau_1$ has the highest priority and $\tau_{ss}$ the lowest. Let us compute the response time of task $\tau_{ss}$ considering two different job release patterns: (i) a job release pattern $\Phi_{ss}$ compliant with its definition made in (Lakshmanan and Rajkumar, 2010) and (ii) a job release pattern different than $\Phi_{ss}$. We show that there exists a job release pattern which is not $\Phi_{ss}$ and for which the response-time of task $\tau_{ss}$ is higher than its response time when the job release pattern is $\Phi_{ss}$.*

*Scenario 1. Let us consider the job release pattern $\Phi_{ss}$ as shown in Fig. A.2a.*

*Using the standard response time expression, we obtain the response time $R_{ss,1} = 3$ for the execution region $\tau_{ss,1}$ and $R_{ss,2} = 4$ for the execution region $\tau_{ss,2}$. Hence, for this scenario, we obtain the response time of task $\tau_{ss}$ to be: $R_{ss} = R_{ss,1} + S_{ss,1} + R_{ss,2} = 3 + 2 + 4 = 9$. Since the release pattern in $\Phi_{ss}$, this is the worst-case response time of $\tau_{ss}$ as per work in (Lakshmanan and Rajkumar, 2010).*

*Scenario 2. Let us consider a job release pattern as shown in Fig. A.2b. Observe that this release pattern is not $\Phi_{ss}$ since the task $\tau_2$ is not released synchronously with task $\tau_1$. For this scenario, we obtain: $R_{ss,1} = 2$ for the execution region $\tau_{ss,1}$ and $R_{ss,2} = 6$ for the second execution region. Hence, the response time of task $\tau_{ss}$ is given by: $R_{ss} = 2+2+6 = 10$.*

*Clearly, the response time of task $\tau_{ss}$ obtained in Scenario 2 is higher than the response time of $\tau_{ss}$ obtained in Scenario 1. Hence, the claim of Lakshmanan et. al. (Lakshmanan and Rajkumar, 2010) that $\Phi_{ss}$ is the critical instant for a self-suspending task $\tau_{ss}$ is incorrect.* □

This counter-example proves the following Lemma.

**Lemma 17.** *The worst-case response time of a lower priority self-suspending task $\tau_{ss}$ suffering interference from a set of higher priority non-self-suspending tasks is not given by $\Phi_{ss}$.*

We now prove that the critical instant of a self-suspending task happens when each higher priority task releases a job synchronously with the beginning of the execution of at least one of the execution regions of the self-suspending task under consideration, although not all higher priority tasks must necessarily release a job synchronously with the same execution region.

**Lemma 18.** *Let $\tau_{ss}$ be the self-suspending task under analysis and let $\mathrm{hp}(\tau_{ss})$ be the set of non-self-suspending tasks of higher priority than $\tau_{ss}$. From any feasible release pattern RP of the tasks in $\mathrm{hp}(\tau_{ss})$, we can construct a feasible release pattern RP$'$ from RP such that:*

**(1)** *In RP$'$, at least one job of every task in $\mathrm{hp}(\tau_{ss})$ is released synchronously with the release of an execution region of $\tau_{ss}$;*

**(2)** RP$'$ *entails a higher (or equivalent) response time of task $\tau_{ss}$ than* RP.

*Proof.* The proof consists in generating RP$'$ from RP such that (1) holds by construction and (2) holds true from the modifications made to RP.

Let us assume that $\tau_{ss}$ is scheduled to execute concurrently with a set $\mathrm{hp}(\tau_{ss})$ of higher priority tasks and suppose that those tasks are released according to the release pattern RP. We denote by $a_{ss,j}$ and $R_{ss,j}$ the release and response time of the $j^{\text{th}}$ execution region of $\tau_{ss}$ and $f_{ss,j} \stackrel{\text{def}}{=} a_{ss,j} + R_{ss,j}$ denotes the completion time of its execution. We denote by $\mathrm{Wb}_k$ and $\mathrm{We}_k$ the beginning and end of the $k^{\text{th}}$ time window during which only tasks in $\mathrm{hp}(\tau_{ss})$ are executed. That is, $\tau_{ss}$ does not execute at all in the time intervals defined by $[\mathrm{Wb}_k, \mathrm{We}_k]$, $\forall k > 0$. Those intervals will be referred to as the *higher priority tasks busy windows*.

Fig. A.3 (top part) shows these notations with a simple example that will be used throughout the proof to illustrate the process of creating RP$'$ from RP. This example assumes that $\mathrm{hp}(\tau_{ss})$ consists of three sporadic tasks. The interference by those tasks on the self-suspending task $\tau_{ss}$ is represented by light rectangles on the first line of Fig. A.3. Dark rectangles correspond to the execution of the execution regions of $\tau_{ss}$. The busy windows generated by tasks in $\mathrm{hp}(\tau_{ss})$ are shown by arrow filled rectangles on the second line of Fig. A.3. Note that only the jobs potentially contributing to the response time of $\tau_{ss}$ are depicted in Fig. A.3.

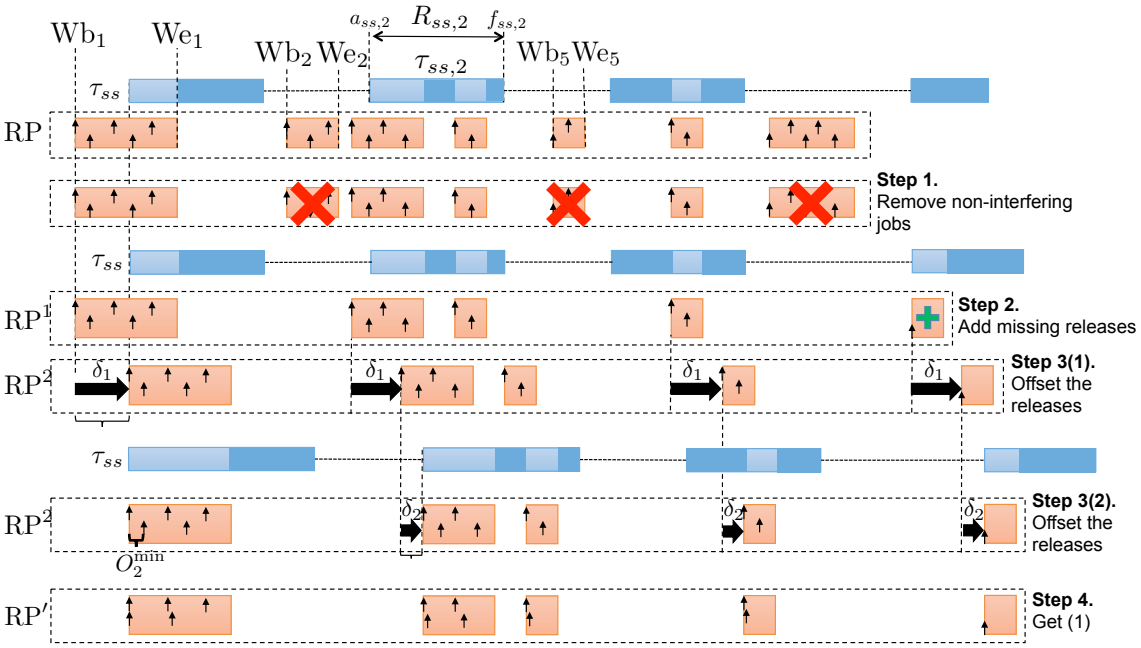



Figure A.3: Illustration of the notation and process described in Lemma 18.

First of all, we remove from RP all the releases from the tasks in $\text{hp}(\tau_{ss})$ that occur in a busy window $[\text{Wb}_k, \text{We}_k]$ that does not overlap with any execution region of $\tau_{\text{ss}}$ (see Step 1 in Figure A.3). Note that removing these releases along with the execution of the corresponding jobs does not alter the schedule of $\tau_{\text{ss}}$ (i.e. it does not impact the response time of any of its execution regions) or that of the jobs of any higher priority task released in any other busy window in RP. As a result, the response time of $\tau_{\text{ss}}$ is not impacted by this modification of RP. We define the resulting release pattern as $\text{RP}^1$.

In order to get (1), each task in $\text{hp}(\tau_{ss})$ must release at least one job in RP′. Since there may be some tasks in $\text{hp}(\tau_{ss})$ that do not release a job in $\text{RP}^1$, one job release of each of those tasks is added to $\text{RP}^1$ such that it coincides with the arrival of the last execution region of $\tau_{\text{ss}}$ (see Step 2 on Fig. A.3). This transformation of $\text{RP}^1$ trivially increases the response time of the last execution region as compared to RP and consequently also increases the overall response time of $\tau_{\text{ss}}$.

The next step to construct RP′ from RP consists in considering all the execution regions of $\tau_{\text{ss}}$ one-by-one, starting from the first one until the last one, and for each region $\tau_{\text{ss},j}$ do the following: if there is a busy window $k$ such that $\text{Wb}_k \leq a_{\text{ss},j} \leq \text{We}_k$ (i.e. $\tau_{\text{ss},j}$ is released within $[\text{Wb}_k, \text{We}_k]$), we then compute the offset $\delta_j$ between the arrival of $\tau_{\text{ss},j}$ and $\text{Wb}_k$, i.e. $\delta_j \stackrel{\text{def}}{=} a_{\text{ss},j} - \text{Wb}_k$. Note that by definition, $\delta_\ell > 0$. If such an overlap exists, we postpone all the higher priority job releases that occur at or after $\text{Wb}_k$ by $\delta_j$ time units. This shift in the job releases makes $\delta_j$ additional units of workload from the tasks in $\text{hp}(\tau_{ss})$ interfere with the execution of $\tau_{\text{ss},j}$. As a consequence, the response time of $\tau_{\text{ss},j}$ increases by $\delta_j$ (i.e. $R_{\text{ss},j} \leftarrow R_{\text{ss},j} + \delta_j$), and so does the time $f_{\text{ss},j}$ at which it finishes its execution ($f_{\text{ss},j} \leftarrow f_{\text{ss},j} + \delta_j$) and, in a cascade effect, the times at which the next regions are released (i.e., $\forall \ell \geq j$, $a_{\text{ss},\ell} \leftarrow a_{\text{ss},\ell} + \delta_j$). Step 3(1) in Fig. A.3) illustrates this process for the first region of $\tau_{\text{ss}}$. At that step all the task releases are delayed by $\delta_1$ time units. Then,

Step 3(2) illustrates the second and last iteration of that process when the second region of $\tau_{ss}$ is considered and all releases occurring at or after $Wb_2$ get postponed by $\delta_2$ time units. For clarity, we have redrawn the interference pattern on $\tau_{ss}$ resulting from that step.

Note that at each iteration of the transformation described above, the response time of the currently considered region $\tau_{ss,j}$ of $\tau_{ss}$ increases by $\delta_j$ time units. However, given that along with this increase, we also delay by $\delta_j$ time units the release of all the subsequent regions of $\tau_{ss}$ and the releases of all the jobs of the tasks in $\text{hp}(\tau_{ss})$ that interfere with those regions, there is no variation in the interference suffered by those execution regions and their response time is not impacted by the transformation. After each iteration, the overall response time of $\tau_{ss}$ therefore increases by $\delta_j$ time units. The release pattern from this transformation is now referred to as $\text{RP}^2$. One can notice that, all the jobs released in $\text{RP}^2$ are released in one of the execution regions of $\tau_{ss}$.

As already explained the response time of every region of $\tau_{ss}$ may have only increased (or remained the same) during the process of constructing $\text{RP}^2$ as described above. Finally, in order to obtain $\text{RP}'$, $\text{RP}^2$ is further modified as follows. For each task $\tau_h \in \text{hp}(\tau_{ss})$, let $\mathcal{R}_h$ denote the set of all its release time-instants in the pattern $\text{RP}^2$. We know that for each of these instants $\text{rel}_{h,x}$ there exists a region $\tau_{ss,j}$ of $\tau_{ss}$ such that $a_{ss,j} \leq \text{rel}_{h,x} < f_{ss,j}$, i.e. that release of $\tau_j$ happens while there is a region $\tau_{ss,j}$ of $\tau_{ss}$ which is running or waiting for the CPU. Now, for each of release in $\mathcal{R}_h$, we compute the offset $O_{h,x}$ of $\text{rel}_{h,x}$ relative to the release of the region of $\tau_{ss}$ which is active at that time. That is, for each $\text{rel}_{h,x} \in \mathcal{R}_h$ we compute $O_{h,x} \stackrel{\text{def}}{=} \text{rel}_{h,x} - a_{ss,j}$ where $j$ is such that $a_{ss,j} \leq \text{rel}_{h,x} < f_{ss,j}$. We then compute the minimum offset $O_h^{\min}$ for $\tau_h$ such that $O_h^{\min} \stackrel{\text{def}}{=} \min_{\forall x}\{O_{h,x}\}$ and shift to the left all the releases of $\tau_h$ by that minimum offset, i.e. for all $\text{rel}_{h,x} \in \mathcal{R}_h$, we impose $\text{rel}_{h,x} \leftarrow \text{rel}_{h,x} - O_h^{\min}$. As a result, none of the releases of $\tau_h$ exit its "encompassing" $\tau_{ss}$'s execution region and, as a consequence, the interference on $\tau_{ss}$ is not modified when passing from $\text{RP}^2$ to $\text{RP}'$. Moreover, because the releases of all the jobs of $\tau_h$ are shifted by the same amount of time, the minimum inter-arrival time between all those jobs is still respected. Finally, at least one job of each task $\tau_k \in \text{hp}(\tau_{ss})$ is now synchronous with the release of an execution region of $\tau_{ss}$ (the one[s] for which the relative offset was minimum, i.e. $O_{h,x} = O_h^{\min}$). This last step of the proof is depicted on the last line of Fig. A.3 for the second task in $\text{hp}(\tau_{ss})$. From the entire discussion, it can be seen that (1) and (2) hold true. Hence the lemma. □

### A.4.2 On maximizing the number of releases in each execution region

In the previous subsection, we proved that the critical instant for a self-suspending task $\tau_{ss}$ suffering interference from non-self-suspending tasks happens when every higher priority task release a job synchronously with at least one execution region of $\tau_{ss}$. Let us now define a set of synchronous release constraints $\text{Synch}^{ss}$ as follows.

**Definition 19** (Set of synchronous release constraints)**.** *A set of synchronous release constraints* $\text{Synch}^{ss}$ *is a set of constraints that impose with which of the execution regions of the self-suspending task* $\tau_{ss}$ *each higher priority non-self-suspending task in* $\text{hp}(\tau_{ss})$ *releases one of its jobs synchronously.*

With the above definition, we now prove the counter-intuitive property that, even considering the set of synchronous constraints that lead to the WCRT of $\tau_{ss}$, the WCRT of $\tau_{ss}$ is not always obtained when the higher priority tasks release as many jobs as possible as early as possible in each execution region of $\tau_{ss}$.

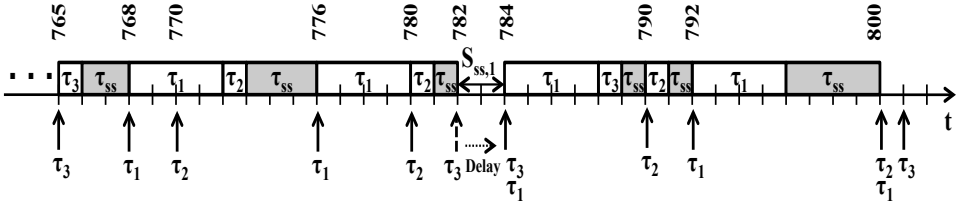
**Lemma 19.** *Let $\tau_{ss}$ be a self-suspending task and let $\mathrm{hp}(\tau_{ss})$ be the set of non-self-suspending tasks of higher priority than $\tau_{ss}$. Let $\mathrm{Synch}^{ss}$ be a set of constraints on the synchronous releases of the tasks in $\mathrm{hp}(\tau_{ss})$. Releasing the jobs of the tasks in $\mathrm{hp}(\tau_{ss})$ as often and as early as possible in each execution region of $\tau_{ss}$ while respecting the set of constraints $\mathrm{Synch}^{ss}$, will not necessarily lead to the WCRT of $\tau_{ss}$ under $\mathrm{Synch}^{ss}$.*

*Proof.* The lemma is proved by showing that the contrapositive version of the claim — that is, that releasing as many jobs as possible as early as possible in each execution region of $\tau_{ss}$ while respecting the set of constraints $\mathrm{Synch}^{ss}$ does always lead to the WCRT of $\tau_{ss}$ under $\mathrm{Synch}^{ss}$ — is false. This can be proved with the following counter-example; Consider a task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_{ss}\}$ of 4 tasks in which $\tau_1$, $\tau_2$ and $\tau_3$ are non-self-suspending tasks and $\tau_s s$ is a self-suspending task with the lowest priority. The tasks are characterized as follows: $\tau_1 = \langle (4), 8, 8 \rangle$, $\tau_2 = \langle (1), 10, 10 \rangle$, $\tau_3 = \langle (1), 17, 17 \rangle$ and $\tau_{ss} = \langle (265, 2, 6), 1000, 1000 \rangle$. The set $\mathrm{Synch}^{ss}$ imposes $\tau_1$ to release a job synchronously with the second execution region $\tau_{ss,2}$ of $\tau_{ss}$ while $\tau_2$ and $\tau_3$ must release a job synchronously with $\tau_{ss,1}$.

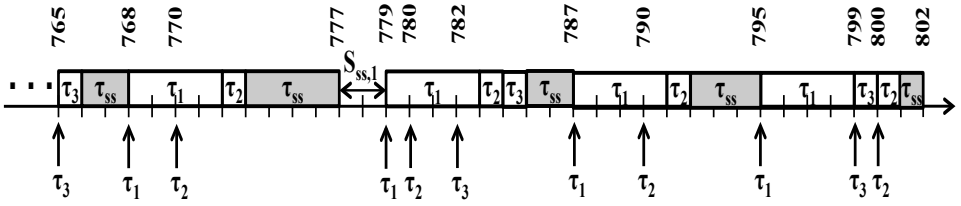
Consider two scenarios with respect to the job release pattern, always respecting the given synchronous release constraints. In Scenario 1, the jobs of the higher priority non-self-suspending tasks are released as often and as early as possible in each execution region of $\tau_{ss}$. In Scenario 2 however, one less job of task $\tau_1$ is released in and therefore interfere with $\tau_{ss,1}$. Showing that the WCRT of the self-suspending task in Scenario 2 is higher than that of Scenario 1 proves the claim.

Scenario 1 is depicted in Fig. A.4a and Fig. A.4b shows Scenario 2. For clarity of presentation, the first 765 time units are omitted in both figures. Furthermore, in both scenarios the release and schedule of the jobs is identical in this time window. A first job of $\tau_1$, $\tau_2$ and $\tau_3$ is released synchronously with the arrival of the first execution region of $\tau_{ss}$ at time 0. The subsequent jobs of these three tasks are released as often as possible respecting the minimum inter-arrival times of the respective tasks. That is, they are released periodically with periods $T1$, $T_2$ and $T_3$, respectively. With this release pattern, it is easy to compute that the $97^{\text{th}}$ job of $\tau_1$ is released at time 768, the $78^{\text{th}}$ job of $\tau_2$ at time 770 and the $46^{\text{th}}$ job of $\tau_3$ at time 765. As a consequence, at time 765, $\tau_{ss}$ has finished executing 259 time units of its first execution region out of 265 (indeed, we have $765 - 96 \times 4 - 77 \times 1 - 45 \times 1 = 259$) in both scenarios. From time 765 onwards, we separately consider Scenario 1 and 2.

**Scenario 1.** Continuing the release of jobs of the non-self-suspending tasks as often as possible without violating their minimum inter-arrival times, the first execution region $\tau_{ss,1}$ of $\tau_{ss}$ finishes its execution at time 782 as shown in Fig. A.4a. After completion of its first execution region, $\tau_{ss}$ self-suspends for two time units until time 784. As $\tau_3$ would have released a job just after the completion of $\tau_{ss,1}$, we delay the release of that job from time 782 to 784 in order to maximize the




(a) Scenario 1. Jobs are released as often as possible while respecting all the constraints on the synchronous releases.




(b) Scenario 2. Jobs are not released as often as possible.

Figure A.4: Example showing that releasing higher priority jobs as often and as early as possible while respecting a set of constraints $\text{Synch}^{\text{ss}}$ on the synchronous releases of the tasks in $\text{hp}(\tau_{ss})$ may not cause the maximum interference to a self-suspending task $\tau_{\text{ss}}$.

interference caused by $\tau_3$ to the second execution region of $\tau_{\text{ss}}$ as shown in Fig. A.4a. Note that, in order to respect its minimum inter-arrival time, $\tau_2$ has an offset of 6 time units with the arrival of the second execution region of $\tau_{\text{ss}}$. Upon following of the rest of the execution, it can easily be seen that the job of $\tau_{ss}$ finishes its execution at time 800.

**Scenario 2.** As shown on Fig. A.4b, the release of a job of task $\tau_1$ is skipped at time 776 in comparison to Scenario 1. As a result, the execution of $\tau_{\text{ss},1}$ is completed at time 777, thereby causing one job of $\tau_2$ that was released at time 780 in Scenario 1, to *not* be released during the execution of the first execution region of $\tau_{ss}$ in Scenario 2. The response time of $\tau_{\text{ss},1}$ is thus reduced by $C_1 + C_2 = 5$ time units in comparison to Scenario 1 (see Fig. A.4). Note that this deviation from Scenario 1 still allows us to respect the synchronous release constraints imposed by $\text{Synch}^{\text{ss}}$ as we can release the next job of $\tau_1$ synchronously with the second execution region of $\tau_{ss}$ without violating the minimum inter-arrival time of $\tau_1$ as can be seen in Figure A.4b. The next job of $\tau_3$ however, is not released in the self-suspending region anymore but 3 time units after the arrival of $\tau_{\text{ss},2}$. Moreover, the offset of $\tau_2$ with the second execution region is reduced by $C_1 + C_2 = 5$ time units. This causes an extra job of $\tau_2$ to be released in the second execution region of $\tau_{ss}$. This initiates a cascade effect, causing an extra job of $\tau_1$ to be released in $\tau_{\text{ss},2}$, which in turn causes the release of an extra job of $\tau_3$ itself causing the arrival of one more job of $\tau_2$ in the second execution region of $\tau_{ss}$. Consequently, the response time of $\tau_{\text{ss},2}$ increases by $C_2 + C_1 + C_3 + C_2 = 7$ time units. Overall, the response time of $\tau_{ss}$ increases by $7 - 5 = 2$ time units in comparison to Scenario 1. This is reflected in Figure A.4b as the job of $\tau_{ss}$ finishes its execution at time 802.

Thanks to this counter-example, we have proved that the response time of a self-suspending task can be larger when the tasks in $\text{hp}(\tau_{ss})$ do not release jobs as often and as early as possible,

thereby proving the lemma. □

**Theorem 13.** *Let* $\tau_{ss}$ *be a self-suspending task and let* $\mathrm{hp}(\tau_{ss})$ *be the set of non-self-suspending tasks of higher priority than* $\tau_{ss}$. *The WCRT of* $\tau_{ss}$ *is not always obtained when the tasks in* $\mathrm{hp}(\tau_{ss})$ *release their jobs as often and as early as possible in the execution regions of* $\tau_{ss}$ *under any set of constraints* $\mathrm{Synch}^{ss}$ *on their synchronous releases.*

*Proof.* Using the system of task $\tau$ of the counter-example provided in Lemma 19, one can check that the response time obtained for $\tau_{ss}$ when releasing jobs as often and as early as possible while respecting any combination of constraints on the synchronous releases of the tasks in $\mathrm{hp}(\tau_{ss})$, never exceeds 800 (note that 4 of the 8 possible combinations are already covered by Scenario 1 of the Fig. A.4 since $\tau_1$ and $\tau_3$ have a synchronous release with both execution regions of $\tau_{ss}$). However, it was shown in the proof of Lemma 19 that a response time of 802 can be experienced by $\tau_{ss}$ when the release of one job of $\tau_1$ is delayed. This proves the theorem. □

## A.5 Exact WCRT for a self-suspending task with one suspension region

In this section, we restrict our analysis to the special case of a self-suspending task $\tau_{ss}$ composed of only two execution regions and one suspension region. We propose an algorithm to compute the exact worst-case response time of such a task. Self-suspending tasks with multiple suspension regions will be considered in the next section.

As proven in Lemma 18, the critical instant for $\tau_{ss}$ happens when at least one job of every higher priority task is released synchronously with the release of the first and/or second execution region of $\tau_{ss}$. However, since it cannot be known a priori which combination of synchronous releases corresponds to the critical instant, there is no other solution for an exact WCRT analysis than consider all the possible combinations of synchronous releases. The exact WCRT for $\tau_{ss}$ is then given by the maximum response time obtained with any of these combinations. Consequently, the WCRT analysis problem boils down to the non-trivial subproblem of computing the WCRT of $\tau_{ss}$ when the higher priority tasks in $\mathrm{hp}(\tau_{ss})$ are constrained to have a synchronous release with a specific execution region of $\tau_{ss}$. We will later refer to that subproblem as the *"constrained releases subproblem"*.

### A.5.1 On the non-triviality of the constrained releases problem

Let us consider a self-suspending task $\tau_{ss}$, a set of higher priority tasks $\mathrm{hp}(\tau_{ss})$ and a subset $\mathrm{Synch}_2^{ss}$ of $\mathrm{hp}(\tau_{ss})$ containing tasks constrained to have a synchronous release with the second execution region of $\tau_{ss}$. The WCRT of the first execution region $\tau_{ss,1}$ of $\tau_{ss}$ under those circumstances can be computed as follows

$$R_{ss,1} = C_{ss,1} + \sum_{\forall k \in \mathrm{hp}(\tau_{ss})} \mathrm{NI}_k \times C_k \tag{A.1}$$

where $\mathrm{NI}_k$ is the maximum number of jobs of $\tau_k$ interfering with $\tau_{\mathrm{ss}}$. According to the usual response time analysis for sporadic tasks with fixed-priorities, $\mathrm{NI}_k$ is subject to the following constraint

$$\mathrm{NI}_k \leq \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil \tag{A.2}$$

Furthermore, for the higher priority tasks that are constrained to have a synchronous release with $\tau_{\mathrm{ss},2}$, one must ensure that

$$\forall \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}},\ \mathrm{NI}_k \times T_k \leq R_{ss,1} + S_{ss,1} \tag{A.3}$$

in order to respect the minimum inter-arrival time $T_k$ of $\tau_k$. That is, for every higher priority task $\tau_k$ that has a synchronous release with $\tau_{\mathrm{ss},2}$, the release of its last job interfering with $\tau_{\mathrm{ss},1}$, happening at time $(\mathrm{NI}_k - 1) \times T_k$, and the beginning of $\tau_{\mathrm{ss},2}$, starting at time $R_{ss,1} + S_{ss,1}$, has to be separated by at least $T_k$ time units.

As a consequence of those constraints, the following equation can be used for $\mathrm{NI}_k$ and substituted in Eq. (A.1)

$$\mathrm{NI}_k = \begin{cases} \min\left( \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil, \left\lfloor \frac{R_{ss,1}+S_{ss,1}}{T_k} \right\rfloor \right) & \text{if } \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}} \\ \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil & \text{otherwise} \end{cases} \tag{A.4}$$

When combined with Eq. (A.4), Eq. (A.1) becomes recursive. This kind of equation is usually solved using a fixed-point iteration on $R_{ss,1}$. However, as shown in the example below, contrary to the traditional WCRT analysis of non-self-suspending sporadic tasks, the first solution found to this equation by increasing the value of $R_{ss,1}$ until it converges may yield to an optimistic (and unsafe) value for the WCRT of $\tau_{\mathrm{ss},1}$.

**Example 18.** *Consider a self-suspending task $\tau_{\mathrm{ss}}$ such that $C_{ss,1} = 3$ and $S_{ss,1} = 1$, and two higher priority tasks $\tau_1 \stackrel{\text{def}}{=} \langle (1), 5, 5 \rangle$ and $\tau_2 \stackrel{\text{def}}{=} \langle (2), 6, 6 \rangle$. Let us assume that both $\tau_1$ and $\tau_2$ are constrained to have a synchronous release with $\tau_{\mathrm{ss},2}$. It can easily be seen that the WCRT of $\tau_{\mathrm{ss},1}$ under those constraints is given when both $\tau_1$ and $\tau_2$ releases one job in $\tau_{\mathrm{ss},1}$, that is, $R_{ss,1} = 6$. However, using a fixed-point iteration with Eq. (A.1) and (A.4), initiating $R_{ss,1}$ to $C_{ss,1} = 3$, the process immediately converges to $R_{ss,1} = 3$, thereby assuming no job released by the higher priority tasks.* □

This example is already sufficient to prove the non-triviality of the constrained releases subproblem. Yet, solving Eq. (A.1) with $\mathrm{NI}_k = \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil$ for all tasks in $\mathrm{hp}(\tau_{ss})$ — that is, when there is no constraint on the task releases — is known to be an upper-bound on the WCRT of $\tau_{\mathrm{ss},1}$ (Tindell and Clark, 1994). Let $\mathrm{UB}_{ss,1}$ be the value of that upper-bound. Based on the observation that increasing $R_{ss,1}$ until its convergence might be optimistic, one might propose to start the fixed-point iteration over $R_{ss,1}$ by initiating $R_{ss,1}$ to $\mathrm{UB}_{ss,1}$. The value of $R_{ss,1}$ should decrease over the

iterations thanks to the constraint $\mathrm{NI}_k \leq \left\lfloor \frac{R_{ss,1}+S_{ss,1}}{T_k} \right\rfloor$. However, as proven in the example provided below, the new solution output by this second method can over-estimate the WCRT of $\tau_{\mathrm{ss},1}$.

**Example 19.** *Consider the same set of tasks as in Example 18, but let us assume that $\tau_1$ must have a synchronous release with $\tau_{\mathrm{ss},2}$ and $\tau_2$ with $\tau_{\mathrm{ss},1}$. It can be computed that the WCRT of $\tau_{\mathrm{ss},1}$ under those constraints is given when both $\tau_1$ and $\tau_2$ release one job in $\tau_{\mathrm{ss},1}$, that is, $R_{ss,1} = 6$. Moreover, the WCRT of $\tau_{\mathrm{ss},1}$ is known to be upper-bounded by $\mathrm{UB}_{ss,1} = 9$ (this can be obtained by solving Eq. (A.1) assuming no constraints on the synchronous releases). However, if we initiate the fixed-point iteration on $R_{ss,1}$ in Eq. (A.1) with $\mathrm{UB}_{ss,1}$, we obtain $R_{ss,1} = 8$. This is impossible since it would mean that $\tau_2$ releases two jobs in $\tau_{\mathrm{ss},1}$. Its second job would be released at time 6 when $\tau_{\mathrm{ss},1}$ just completed its execution.* □

These examples show that, in the general case, no trivial solution exists to the constrained releases subproblem.

### A.5.2 Solution for the constrained releases subproblem

In this subsection, we propose a method to compute the exact WCRTs $R_{ss,1}$ and $R_{ss,2}$ under a set of constraints $\mathrm{Synch}_2^{\mathrm{ss}}$.

The proposed method to compute $R_{ss,1}$ is based on a combination of the two straightforward but inexact solutions investigated in Section A.5.1. That is, we simultaneously increase and decrease the value of $R_{ss,1}$ in two different but interdependent iterative processes until they converge to the same value. To do so, Eq. (A.4) is rewritten as follows

$$\mathrm{NI}_k = \begin{cases} \min\left(\left\lceil \frac{R_{ss,1}}{T_k} \right\rceil, \mathrm{NI}_{\mathrm{k}}^{\max}\right) & \text{if } \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}} \\ \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil & \text{otherwise} \end{cases} \tag{A.5}$$

where $\mathrm{NI}_{\mathrm{k}}^{\max}$ is an upper-bound on the number of jobs of $\tau_k$ interfering with the first execution region of $\tau_{\mathrm{ss}}$. This new formulation of Eq. A.4 removes the recursiveness in the new term that was added to the equation of $\mathrm{NI}_k$ in order to enforce compliance with the constraints imposed by $\mathrm{Synch}_2^{\mathrm{ss}}$. The WCRT can therefore be computed with the usual fixed-point iteration on $R_{ss,1}$ where $R_{ss,1}$ is initialized to $C_{ss,1}$ for the first iteration. During that process, $\mathrm{NI}_{\mathrm{k}}^{\max}$ is assigned a known upper-bound on $\mathrm{NI}_k$. Because $\mathrm{NI}_{\mathrm{k}}^{\max}$ is an upper-bound, the result obtained for $R_{ss,1}$ after convergence of Eq. (A.5) is also an upper-bound on the actual WCRT of $\tau_{\mathrm{ss},1}$. However, with this value, the constraint expressed by Eq. (A.3) may not be respected. Therefore, the constraint imposed by Eq. (A.3) is checked and if violated the value of $\mathrm{NI}_{\mathrm{k}}^{\max}$ is decreased and Eq. (A.1) is solved again. Otherwise, an exact WCRT for $\tau_{\mathrm{ss},1}$ has been found.

Lines 2 to 14 of Algorithm 7 present a pseudo-code of that method. Starting with the upper bound $\mathrm{UB}_{ss,1}$ on $R_{ss,1}$ (line 4), it iteratively removes jobs of higher priority tasks interfering with $\tau_{\mathrm{ss},1}$ (lines 7–11) when the condition expressed in Eq. (A.3) is violated, i.e., $\mathrm{NI}_k > \frac{R_{ss,1}+S_{ss,1}}{T_k}$. With this updated value, the response time $R_{ss,1}$ is recomputed at line 12 of Algorithm 7. This pro-

cess iterates until the value computed for $R_{ss,1}$ converges to the exact WCRT of $\tau_{\text{ss},1}$ under the constraints imposed by $\text{Synch}_2^{\text{ss}}$.

Once the response time of $\tau_{\text{ss},1}$ has been computed, the offset $O_{k,2}$ between the earliest instant at which each task $\tau_k$ can release a job in $\tau_{\text{ss},2}$ while respecting its minimum inter-arrival time $T_k$ can be obtained with Eq. (A.6).

$$O_{k,2} = \begin{cases} 0 & \text{if } \tau_k \in \text{Synch}_2^{\text{ss}} \\ \max(0,\ \text{NI}_k \times T_k - R_{i,1} - S_{i,1}) & \text{otherwise} \end{cases} \tag{A.6}$$

As expressed by Eq. (A.6), any release of a job of $\tau_k$ that should happen within the suspension region of $\tau_{\text{ss}}$ is delayed until the beginning of $\tau_{\text{ss},2}$, thereby imposing $O_{k,2} = 0$. This allows us to maximize the interference caused by $\tau_k$ to $\tau_{\text{ss},2}$.

$R_{ss,2}$ is given by Eq. (A.7) and is computed as the traditional response time for non-self-suspending tasks. That is, we seek a minimum response time that satisfies the fixed-point iteration by starting with $R_{ss,2} = C_{ss,2}$.

$$R_{ss,2} = C_{ss,2} + \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_{ss,2} - O_{i,2}}{T_k} \right\rceil \times C_k \tag{A.7}$$

In Algorithm 7, this is reflected in lines 15 to 18. Line 19 computes the overall response time of $\tau_{\text{ss}}$. However, as proven in Section A.4.2, it might happen that releasing one less job in $\tau_{\text{ss},1}$ allows to increase the response time of $\tau_{\text{ss},2}$ and in turn increase the overall response time of $\tau_{\text{ss}}$. Therefore, lines 20 to 29 have been added to consider that case. Those lines call recursively the function RespTime, imposing the upper-bound on the number of interfering jobs with the first execution region to be one less than in the computed solution. Of course, this recursion must not be activated if the overall response time $R_{ss}$ found for $\tau_{\text{ss}}$ is already equal to a known upper-bound obtained with simple approximation techniques likes those proposed in (Bletsas, 2007). Similarly, there is no interest in trying to increase the response time of $\tau_{\text{ss},2}$ by reducing the response time of $\tau_{\text{ss},1}$ if $R_{ss,2}$ is already equal to an upper-bound.

It can easily be seen by looking at Algorithm 7, that computing the exact WCRT of $\tau_{\text{ss}}$ becomes rapidly intractable. This fact has been confirmed by the experiments reported in Section A.8. Therefore, in the next section, we propose a second method, using a MILP formulation, to compute an approximation over the WCRT of $\tau_{\text{ss}}$.

**Algorithm 7:** Recursive function computing the WCRT of $\tau_{ss}$ assuming a set of constraints on higher priority tasks releases

**Function** *RespTime (* $\text{hp}(\tau_{ss})$, $\text{Synch}_2^{ss}$, $NI^{up}$ *)* **is**
  **Inputs** : $\text{hp}(\tau_{ss})$ - set of higher priority tasks w.r.t. $\tau_{ss}$
    $\text{Synch}_2^{ss}$ - the set of tasks in $\text{hp}(\tau_{ss})$ with a synchronous release with $\tau_{ss,2}$
    $NI^{up}$ - vector of upper bounds on the number of jobs of each task $\tau_k$, that can interfere with $\tau_{ss,1}$
  **Output**: $R_{ss}$ - The exact WCRT for $\tau_{ss}$ when respecting the constraints given by $\text{Synch}_2^{ss}$ and $NI^{up}$
  $R_{ss,1}^{bwd} \leftarrow 0$ ;
  $NI \leftarrow NI^{up}$;
  $R_{ss,1} \leftarrow C_{ss,1} + \sum_{\forall k \in \text{hp}(\tau_{ss})} NI_k^{up} \times C_k$ ;
  **while** $R_{ss,1}^{bwd} \neq R_{ss,1}$ **do**
    $R_{ss,1}^{bwd} \leftarrow R_{ss,1}$ ;
    /* Update the number of interfering jobs for the tasks synchronous with $\tau_{ss,2}$ */
    **forall the** $\tau_k \in \text{Synch}_2^{ss}$ **do**
      **if** $NI_k > \frac{R_{ss,1}+S_{ss,1}}{T_k}$ **then**
        $NI_k \leftarrow NI_k - 1$ ;
      **end**
    **end**
    // Compute the response time of $\tau_{ss,1}$
    $R_{ss,1} \leftarrow C_{ss,1} + \sum_{\forall k \in \text{hp}(\tau_{ss})} \min(NI_k; \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil) \times C_k$;
    **forall the** $\tau_k \in \text{hp}(\tau_{ss})$ **do** $NI_k \leftarrow \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil$;
  **end**
  // Compute the offsets with $\tau_{ss,2}$
  **forall the** $\tau_k \in \text{hp}(\tau_{ss})$ **do**
    $O_{k,2} \leftarrow \max(0; NI_k \times T_k - R_{ss,1} - S_{ss,1})$;
  **end**
  // Compute the response time of $\tau_{ss,2}$
  $R_{ss,2} \leftarrow C_{ss,2} + \sum_{\forall k \in \text{hp}(\tau_{ss})} \left\lceil \frac{R_{ss,2}-O_{k,2}}{T_k} \right\rceil \times C_k$;
  $R_{ss} \leftarrow R_{ss,1} + S_{ss,1} + R_{ss,2}$; ;
  /* Check if there is not a release pattern with one less job in $\tau_{ss,1}$ that increases the overall WCRT */
  **if** $R_{ss} < \text{UB}_{ss,}$ *and* $R_{ss,2} < \text{UB}_{ss,2}$ **then**
    **forall the** $\tau_k \in \text{hp}(\tau_{ss})$ **do**
      **if** $NI_k > 0$ **then**
        $NI' \leftarrow NI$ ;
        $NI'_k \leftarrow NI_k - 1$ ;
        $R \leftarrow \text{RespTime}(\text{hp}(\tau_{ss}), \text{Synch}_2^{ss}, NI')$;
        **If** $R > R_{ss}$ **then** $R_{ss} \leftarrow R$;
      **end**
    **end**
  **end**
  return $R_{ss}$;
**end**

## A.6 Upper-bound on the WCRT of a self-suspending task with multiple suspension regions

As shown in the previous section and contrary to what was commonly believed, the timing analysis of a set of sporadic non-self-suspending tasks interfering with a sporadic self-suspending task is challenging, as it the case in the analysis of a set of periodic tasks interfering with a self-suspending task. The exact test, even when only one suspension region is considered, rapidly becomes intractable. In this section, we therefore propose an MILP formulation for computing an upper-bound on the WCRT of a self-suspending task with *multiple* suspension regions when all the interfering tasks are non-self-suspending. This formulation will be extended in the next section to consider the case where multiple self-suspending tasks interfere with each other.

The optimization problem, defined by Expressions (A.8) to (A.16), has the objective to maximize the sum of the response times of every execution region of $\tau_{\text{ss}}$. Its constraints (A.9)–(A.16) can all be easily linearized. In the proposed problem formulation, the number of jobs $\text{NI}_{k,j}$ of each task $\tau_k \in \text{hp}(\tau_{ss})$ interfering with each execution region of $\tau_{\text{ss}}$ are integer variables while the response time $R_{ss,j}$ of each execution region $\tau_{\text{ss},j}$ of $\tau_{\text{ss}}$ and the offsets $O_{k,j}$ of each task $\tau_k$ with each execution region $\tau_{\text{ss},j}$ are real variables. This MILP formulation is quite simple in comparison to the exact test described in Algorithm 7. As demonstrated in Section A.8, this permits a state-of-the-art MILP solver to output results in an acceptable amount of time for reasonable system sizes.

$$\textbf{Maximize:} \quad \sum_{j=1}^{m_{ss}} R_{ss,j} \tag{A.8}$$

**Subject to:**

$$\sum_{j=1}^{m_{ss}} (R_{ss,j} + S_{ss,j}) \leq \text{UB}_{ss,} \tag{A.9}$$

$$\forall \tau_{\text{ss},j} \in \tau_{\text{ss}} \ : \ R_{ss,j} = C_{ss,j} + \sum_{\tau_p \in \text{hp}(\tau_{ss})} \text{NI}_{p,j} \times C_p \tag{A.10}$$

$$R_{ss,j} \leq \text{UB}_{ss,j} \tag{A.11}$$

$$\forall \tau_k \in \text{hp}(\tau_{ss}) : \ O_{k,1} \geq 0 \tag{A.12}$$

$$\forall \tau_k \in \text{hp}(\tau_{ss}), \forall \tau_{\text{ss},j} \in \tau_{\text{ss}} :$$

$$O_{k,j+1} \geq \max\left(0, \ O_{k,j} + \text{NI}_{k,j} \times T_k - R_{ss,j} - S_{ss,j}\right) \tag{A.13}$$

$$\text{NI}_{k,j} \geq 0 \tag{A.14}$$

$$\text{NI}_{k,j} \leq \left\lceil \frac{R_{ss,j} - O_{k,j}}{T_k} \right\rceil \tag{A.15}$$

$$R_{ss,j} > O_{k,j} + (\text{NI}_{k,j} - 1) \times T_k + \sum_{\tau_p \in \text{hp}(\tau_{ss})} \left\lfloor \text{NI}_{p,j} + \frac{O_{p,j} - O_{k,j} - (\text{NI}_{k,j} - 1) T_k}{T_p} \right\rfloor C_p \tag{A.16}$$

The constraints (A.9)–(A.15) of the optimization problem are a direct translation of the con-

straints already discussed in Section A.5. That is, Constraint (A.10) is equivalent to Eq. (A.1); Constraints (A.12) and (A.13) are a generalization of Eq. (A.6) computing the offsets of the higher priority tasks with each execution region; and Constraints (A.14) and (A.15) impose the traditional lower- and upper-bound on the number of interfering jobs of each task $\tau_k$ with each execution region $\tau_{ss,j}$ as already discussed for Eq. (A.7). Constraints (A.9) and (A.11) reduce the research space of the problem by stating that the overall response time of $\tau_{ss}$ and the response time of each of its execution region, respectively, cannot be larger than known upper-bounds computed with simple methods such as the *Joint* and *Split* methods presented in (Bletsas, 2007). Therefore, Constraint (A.16) is the only new expression requiring some explanations. Constraint (A.16) can be understood as follows; by replacing $O_{k,j} + (\text{NI}_{k,j} - 1) \times T_k$ and $O_{p,j} + \text{NI}_{p,j} \times T_p$ by $\text{rel}_{k,j}$ and $d_{p,j}$, respectively, we get

$$R_{ss,j} > \text{rel}_{k,j} + \sum_{\tau_p \in \text{hp}(\tau_{ss})} \left\lfloor \frac{d_{p,j} - \text{rel}_{k,j}}{T_p} \right\rfloor C_p$$

The value of $\text{rel}_{k,j}$ gives the release instant of the last job of $\tau_k$ released in the execution region $\tau_{ss,j}$, while $d_{p,j}$ gives the deadline of the last job of $\tau_p$ released in $\tau_{ss,j}$. Therefore, the floor value in Constraint (A.16) provides the number of jobs released by $\tau_p$ after $\text{rel}_{k,j}$ and the sum thus gives the total workload released by higher priority tasks after $\text{rel}_{k,j}$. Since $\tau_{ss}$ cannot execute when higher priority workload is available and because $\text{rel}_{k,j}$ is an instant in the response time of the execution region, the response time of $\tau_{ss,j}$ cannot be smaller than $\text{rel}_{k,j}$ plus the higher priority workload remaining to execute after $\text{rel}_{k,j}$. This is what Constraint (A.16) enforces.

Note that because the optimization problem tests all the possible values for the offsets $O_{k,j}$ of each task $\tau_k$ with every execution region of $\tau_{ss}$, it also tests all the possible synchronous release combinations. Therefore, there is no need to impose additional constraints on the synchronous release patterns, as it was the case in Algorithm 7.

## A.7 Multiple self-suspending tasks

In this section, we propose a solution to analyze multiple self-suspending tasks interfering together. We show below that each higher priority task $\tau_k$ can safely be replaced by a set of non-self-suspending tasks in the response time analysis. In particular, each execution region $\tau_{k,j}$ of $\tau_k$ is modelled by a different non-self-suspending task $\tau'_{k,j}$ with jitter $J_{k,j}$. Such solution was already proposed in (Palencia and Gonzalez Harbour, 1998). In (Palencia and Gonzalez Harbour, 1998), the jitter $J_{k,j}$ is given by the difference between the WCRT and BCRT of the partial self-suspending task composed of the $j-1$ first execution and suspension regions of $\tau_k$. Formally,

**Lemma 20.** *Let $\tau_{k,j}$ be the $j^{th}$ execution region of $\tau_k$, and let $\tau_k^j$ be a self-suspending task composed of the $j-1$ first execution and suspension regions of $\tau_k$, that is, $\tau_k^j \stackrel{\text{def}}{=} \langle (C_{k,1}, S_{k,1}, \ldots, C_{k,j-1}, S_{k,j-1}), D_k, T_k \rangle$. The release jitter of $\tau_{k,j}$ is upper bounded by $J_{k,j} \stackrel{\text{def}}{=} \text{WCRT}_k^j - \text{BCRT}_k^j$, where $\text{WCRT}_k^j$ and $\text{BCRT}_k^j$ are the worst-case and best-case response time of $\tau_k^j$, respectively.*

*Proof.* The minimum inter-arrival time of the execution region $\tau_{k,j}$ of task $\tau_k$ is inherited from the minimum inter-arrival time of $\tau_k$. However, the execution region $\tau_{k,j}$ can start to execute only when the $(j-1)^{\text{th}}$ suspension region of $\tau_k$ completes, that is, when the partial self-suspending task $\tau_k^j$ completes its execution. Since the response time of $\tau_k^j$ may vary between different jobs released by $\tau_k$, the release of $\tau_{k,j}$ experiences a jitter. This jitter is upper-bounded by the difference between the longest and the shortest response time of $\tau_k^j$, i.e., it is upper-bounded by the difference between $\text{WCRT}_k^j$ and $\text{BCRT}_k^j$. □

Let $\text{hp}(\tau_{ss})$ be a set of self-suspending tasks with higher priorities than $\tau_{ss}$. And let $\text{hp}(\tau_{ss})'$ be a set of non-self-supending tasks where for each task $\tau_k \in \text{hp}(\tau_{ss})$, the set $\text{hp}(\tau_{ss})'$ contains $m_k$ non-self-suspending tasks $\tau'_{k,j} \stackrel{\text{def}}{=} \langle (C_{k,j}), D_k, T_k, J_{k,j} \rangle$ with $1 \leq j \leq m_k$, where $J_{k,j}$ is defined as in Lemma 20 and each task $\tau'_{k,j}$ $(1 \leq j \leq m_k)$ has the same priority than $\tau_k$. We prove below that replacing $\text{hp}(\tau_{ss})$ with $\text{hp}(\tau_{ss})'$ in the WCRT analysis of $\tau_{ss}$ provides a response time upper-bound which is at least as large as the WCRT when using $\text{hp}(\tau_{ss})$. Therefore, replacing $\text{hp}(\tau_{ss})$ with $\text{hp}(\tau_{ss})'$ is safe.

We first define what is a legal release pattern for a task set.

**Definition 20** (Legal release pattern for a task set $\tau$)**.** *A release pattern $\mathcal{R}$ defines all the instants at which each execution region of the tasks in $\tau$ releases jobs. A release pattern $\mathcal{R}$ is legal if all the constraints defined by the tasks in $\tau$ (i.e., minimum inter-arrival time, precedence constraints and release jitter) are respected in $\mathcal{R}$.*

Now, we prove that the release pattern of the task set $\text{hp}(\tau_{ss})$ that generates the WCRT of $\tau_{ss}$ can be transformed in a legal release pattern for the tasks in $\text{hp}(\tau_{ss})'$.

**Lemma 21.** *Let $\overline{\mathcal{R}}$ be any legal release pattern of the execution regions of the tasks in $\text{hp}(\tau_{ss})$ such that the tasks in $\text{hp}(\tau_{ss})$ generate the worst-case interference on $\tau_{ss}$. Let $\overline{\mathcal{R}}'$ be a release pattern for the tasks in $\text{hp}(\tau_{ss})'$ such that whenever an execution region $\tau_{k,j} \in \text{hp}(\tau_{ss})$ releases a job in $\overline{\mathcal{R}}$, the corresponding task $\tau'_{k,j}$ releases a job at the same instant in $\overline{\mathcal{R}}'$. The release pattern $\overline{\mathcal{R}}'$ is a legal release pattern for the tasks in $\text{hp}(\tau_{ss})'$.*

*Proof.* We have to prove that the minimum inter-arrival times, release jitters and precedence constraints defined for the tasks in $\text{hp}(\tau_{ss})'$ are all respected in $\overline{\mathcal{R}}'$.

1. The minimum inter-arrival time of $\tau_{k,j}$ is $T_k$ and its release jitter is smaller than or equal to $J_{k,j}$ (from Lemma 20). Let $\tau_{k,j}^{\ell}$ be the $\ell^{\text{th}}$ instance (job) released by $\tau_{k,j}$. Since $\overline{\mathcal{R}}$ is legal, the time between any two jobs $\tau_{k,j}^{\ell}$ and $\tau_{k,j}^{\ell+p}$ released by $\tau_{k,j}$ is at least $(p \times T_k) - J_{k,j}$. Therefore, the time between any two jobs $\tau_{k,j}^{\ell'}$ and $\tau_{k,j}^{\ell+p'}$ released by $\tau'_{k,j}$ is at least $(p \times T_k) - J_{k,j}$ in the release pattern $\overline{\mathcal{R}}'$. Since by definition, the minimum inter-arrival time and the release jitter of $\tau'_{k,j}$ are $T_k$ and $J_{k,j}$ respectively, the release pattern $\overline{\mathcal{R}}'$ respects the minimum inter-arrival time and the release jitter constraints on $\tau'_{k,j}$.

2. Since the tasks in $\text{hp}(\tau_{ss})'$ do not have any precedence constraints, the release pattern $\overline{\mathcal{R}}'$ trivially respects those constraints.

By 1. and 2., the release pattern $\overline{\mathcal{R}}'$ is legal for $\text{hp}(\tau_{ss})'$. □

We finally prove that replacing $\text{hp}(\tau_{ss})$ by $\text{hp}(\tau_{ss})'$ in the WCRT analysis of $\tau_{ss}$ is safe.

**Theorem 14.** *The worst-case interference generated by the tasks in* $\text{hp}(\tau_{ss})'$ *is lower-bounded by the worst-case interference generated by the tasks in* $\text{hp}(\tau_{ss})$.

*Proof.* The proof is based on the following facts:

F1. If a job of $\tau_{k,j}$ or $\tau'_{k,j}$ interferes with the execution region $\tau_{ss,p}$ of $\tau_{ss}$ than they do not interfere with any other execution region of $\tau_{ss}$. This statement is true because (i) both $\tau_{k,j}$ and $\tau'_{k,j}$ have a higher priority than $\tau_{ss}$, and (ii) they do not self-suspend. Therefore, when they start to interfere with one execution region of $\tau_{ss}$, that execution region cannot resume its execution before $\tau_{k,j}$ or $\tau'_{k,j}$ complete their own execution.

F2. When they execute for their WCET, one job of $\tau_{k,j}$ generates as much interference as one job of $\tau'_{k,j}$. It is simply due to the fact that $\tau_{k,j}$ and $\tau'_{k,j}$ have the same WCET.

Let $\overline{\mathcal{R}}$ be any legal release pattern of the execution regions of the tasks in $\text{hp}(\tau_{ss})$ such that the tasks in $\text{hp}(\tau_{ss})$ generate the worst-case interference on $\tau_{ss}$. And let $\overline{\mathcal{R}}'$ be the corresponding release pattern for the tasks in $\text{hp}(\tau_{ss})'$ such that whenever an execution region $\tau_{k,j}$ of a task $\tau_k$ in $\text{hp}(\tau_{ss})$ releases a job in $\overline{\mathcal{R}}$, the corresponding task $\tau'_{k,j}$ releases a job at the same instant in $\overline{\mathcal{R}}'$. By Lemma 21, $\overline{\mathcal{R}}'$ is a legal release pattern for the tasks in $\text{hp}(\tau_{ss})'$. Since by Fact F2., each job released by each task $\tau'_{k,j}$ generates as much interference than each job released by the corresponding execution region $\tau_{k,j}$, and because by Fact F1., this interference is generated in the same execution region of $\tau_{ss}$, the total interference generated by the set of tasks in $\text{hp}(\tau_{ss})'$ under the release pattern $\overline{\mathcal{R}}'$ is equal to the worst-case interference generated by the corresponding self-suspending tasks in $\text{hp}(\tau_{ss})$ under $\overline{\mathcal{R}}$.

Therefore, because we proved that there exists at least one legal release pattern of the tasks in $\text{hp}(\tau_{ss})'$ generating as much interference as the worst-case interference generated by $\text{hp}(\tau_{ss})$, the worst-case interference generated by the tasks in $\text{hp}(\tau_{ss})'$ is lower-bounded by the worst-case interference generated by the tasks in $\text{hp}(\tau_{ss})$. □

**Theorem 15.** *The WCRT of* $\tau_{ss}$ *running concurrently with* $\text{hp}(\tau_{ss})'$ *is no smaller than its WCRT when it runs concurrently with* $\text{hp}(\tau_{ss})$.

*Proof.* Theorem 14 proves that $\text{hp}(\tau_{ss})'$ generates at least as much interference on $\tau_{ss}$ than $\text{hp}(\tau_{ss})$. Therefore, the WCRT of $\tau_{ss}$ when its runs concurrently with $\text{hp}(\tau_{ss})'$ is no smaller than its WCRT when it runs concurrently with $\text{hp}(\tau_{ss})$. □

### A.7.1 Upper-bounding $J_{k,j}$

The solution presented above requires an upper-bound on the jitter $J_{k,j}$ experienced by each execution region $\tau_{k,j}$. In this subsection, we provide three different upper-bounds (stated in Lemmas 22, 23 and 24) on the jitter $J_{k,j}$.

**Lemma 22.** *The release jitter $J_{k,j}$ of $\tau_{k,j}$ is upper bounded by* $\mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$.

*Proof.* Let $a_k$ and $f_k$ be the release time and the completion time of any job of $\tau_k$, and let $a_{k,j}$ be the release time of the execution region $\tau_{k,j}$ in that job. Instant $a_{k,j}$ also corresponds to the completion time of the partial self-suspending task $\tau_k^j$. We prove that $a_{k,j}$ is no later than $a_k + \mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$.

The proof is by contradiction. Let us assume that the completion of $\tau_k^j$, and hence the release of $\tau_{k,j}$, happens after $a_k + \mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$, that is,

$$a_{k,j} > a_k + \mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p} \tag{A.17}$$

If every execution region executes for its worst-case execution time and every suspension region suspends for its worst-case suspension time, then $\tau_k$ must still execute for $\sum_{p=j}^{m_k} C_{k,p}$ time units and suspend for $\sum_{p=j}^{m_k-1} S_{k,p}$ time units after $a_{k,j}$. Therefore, even without interference from higher priority tasks, task $\tau_k$ completes its execution at time

$$f_k \geq a_{k,j} + \sum_{p=j}^{m_k} C_{k,p} + \sum_{p=j}^{m_k-1} S_{k,p}$$

Replacing $a_{k,j}$ with Eq. (A.17), we get

$$f_k > a_k + \mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p} + \sum_{p=j}^{m_k} C_{k,p} + \sum_{p=j}^{m_k-1} S_{k,p}$$

Simplifying and passing $a_k$ from the right-hand side to the left-hand side, we obtain

$$f_k - a_k > \mathrm{WCRT}_k$$

which is a clear contradiction with the fact that $\mathrm{WCRT}_k$ is an upper-bound on the response time of $\tau_k$. It results that for any job of $\tau_k$, the partial self-suspending task $\tau_k^j$ completes at time $a_{k,j} \leq a_k + \mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$. The worst-case response time $\mathrm{WCRT}_k^j$ of $\tau_k^j$ is therefore upper-bounded by $\mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$.

Since the best-case response time $\mathrm{BCRT}_k^j$ of $\tau_k^j$ is trivially lower-bounded by 0, the jitter $J_{k,j}$, which by definition is equal to $\mathrm{WCRT}_k^j - \mathrm{BCRT}_k^j$, is upper-bounded by $\mathrm{WCRT}_k - \sum_{p=j}^{m_k} C_{k,p} - \sum_{p=j}^{m_k-1} S_{k,p}$. □

**Lemma 23.** *The release jitter $J_{k,j}$ of $\tau_{k,j}$ is upper bounded by* $\sum_{p=1}^{j-1} \left(\mathrm{UB}_{k,p} + S_{k,p}\right)$ *where* $\mathrm{UB}_{k,p}$ *is an upper bound on the WCRT of each execution region $\tau_{k,p}$ given by the smallest positive $t$ such that*

$$t = C_{k,p} + \sum_{\tau_\ell \in \mathrm{hp}(\tau_k)} \left\lceil \frac{t + J_\ell}{T_\ell} \right\rceil C_\ell$$

*Proof.* It was proven in (Bletsas, 2007) that the WCRT of a self-suspending task $\tau_k^j$ is upper-bounded by $\sum_{p=1}^{j-1}\left(\mathrm{UB}_{k,p}+S_{k,p}\right)$. Since $J_{k,j} \stackrel{\text{def}}{=} \mathrm{WCRT}_k^j - \mathrm{BCRT}_k^j$, and because $\mathrm{BCRT}_k^j$ is lower-bounded by 0, we get that $J_{k,j} \leq \sum_{p=1}^{j-1}\left(\mathrm{UB}_{k,p}+S_{k,p}\right)$. □

**Lemma 24.** *The release jitter $J_{k,j}$ of $\tau_{k,j}$ is upper bounded by $\mathrm{UB}_k^j + S_{k,j-1}$ where $\mathrm{UB}_k^j$ is given by the smallest positive $t$ such that*

$$t = \sum_{p=1}^{j-1} C_{k,p} + \sum_{p=1}^{j-2} S_{k,p} + \sum_{\tau_\ell \in \mathrm{hp}(\tau_k)} \left\lceil \frac{t+J_\ell}{T_\ell} \right\rceil C_\ell$$

*Proof.* It was proven in (Bletsas, 2007) that the WCRT of a self-suspending task $\langle (C_{k,1}, S_{k,1}, \ldots, C_{k,j-1}), D_k, T_k \rangle$ is upper-bounded by $\mathrm{UB}_k^j$. Because the last suspension region $S_{k,j-1}$ of $\tau_k^j$ cannot be preempted, the WCRT of $\tau_k^j$ is given by $\mathrm{UB}_k^j + S_{k,j-1}$. Since $J_{k,j} \stackrel{\text{def}}{=} \mathrm{WCRT}_k^j - \mathrm{BCRT}_k^j$, and because $\mathrm{BCRT}_k^j$ is lower-bounded by 0, we get that $J_{k,j}$ is upper-bounded by $\mathrm{UB}_k^j + S_{k,j-1}$. □

### A.7.2 Extending the optimization problem

Using Theorem 15, each higher priority self-suspending task can be transformed in a set of non-self-suspending tasks with jitter. Without loss of generality, let $\tau_k \in \mathrm{hp}(\tau_{ss})$ correspond to any non-self-suspending $\tau'_{i,h} \in \mathrm{hp}(\tau_{ss})'$, where $C_k = C_{i,h}$ and $J_k = J_{i,h}$. This new model can easily be integrated in the MILP formulation presented in Section A.6, which computes an upper bound on the WCRT a self-suspending task $\tau_{ss}$ running concurrently with a set of non-self-suspending tasks.

Let $J_{k,j}$ represent the jitter experienced by the jobs of $\tau_k$ released in the $j^{\text{th}}$ execution region of $\tau_{\text{ss}}$. In the traditional response time analysis, the jitter can be accounted by subtracting it from the offset of the interfering task (Liu, 2000). That is, Constraint (A.15) for instance would become

$$\mathrm{NI}_{k,j} \leq \left\lceil \frac{R_{ss,j} - (O_{k,j} - J_{k,j})}{T_k} \right\rceil$$

. However, instead of introducing a new set of variables in the optimization problem and hence increase its complexity, one can simply replace $O_{k,j}$ by $O'_{k,j}$ in Constraints (A.15) and (A.16), where $O'_{k,j}$ is defined as $O'_{k,j} \stackrel{\text{def}}{=} O_{k,j} - J_{k,j}$. Because $J_{k,j}$ is upper-bounded by $J_k$, this variable replacement has for consequence that the bound imposed on the offsets of the tasks in $\mathrm{hp}(\tau_{ss})$ must be modified. Therefore, Constraints (A.12) and (A.13) must be replaced by:

$$\begin{aligned}
&\forall \tau_k \in \mathrm{hp}(\tau_{ss}) : \; O'_{k,1} \geq -J_k \\
&\forall \tau_k \in \mathrm{hp}(\tau_{ss}), \forall \tau_{\mathrm{ss},j} \in \tau_{\mathrm{ss}} : \\
&\qquad O'_{k,j+1} \geq O'_{k,j} + \mathrm{NI}_{k,j} \times T_k - R_{ss,j} - S_{ss,j} - J_k
\end{aligned}$$

## A.8 Experiments

In this section, we describe experiments conducted through randomly generated task sets to evaluate (i) the applicability of our exact WCRT computation algorithm, (ii) the performance of the MILP method, and (iii) the respective gain in comparison with the state-of-the-art analysis for sporadic self-suspending tasks. We used Gurobi (Gurobi Optimization Inc., 2015), a state-of-the-art MILP solver, to solve our optimization problem.

All the task sets were generated using the `randfixedsum` algorithm (Emberson et al., 2010), allowing us to choose a constant total task set utilization for a given number of tasks and bounded per-task utilization. Accordingly, the total utilization ($U_{\text{tot}}$) was varied from 0.1 to 1 by 0.1 increments. The per-task utilization ranged from $[0.05, \frac{U_{\text{tot}}}{2}]$, while periods were uniformly distributed over $[10, 100]$. The task execution requirements were calculated from the respective periods and utilizations. Individual values for each of the multiple suspension regions and computing stages were assigned as fraction of the overall suspension length $S_{ss}$, using again the `randfixedsum` algorithm for a constant total of 1 and a minimum fraction of 0.1. We generated 100 task sets per combination of parameters, while ensuring that task $\tau_{n-1}$ was always schedulable.

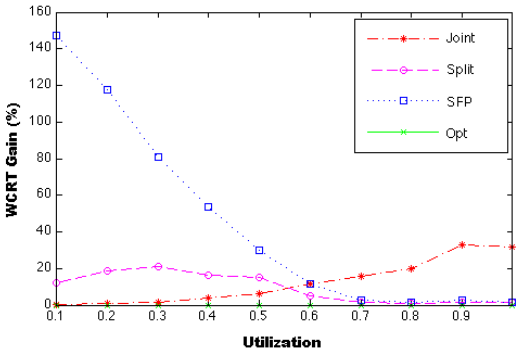
We evaluated our techniques for computing the WCRT of sporadic self-suspending tasks under fixed-priority scheduling by comparing them to the analysis presented in (Kim et al., 2013b) (denoted "SFP") and the 'Split" and "Joint" analysis presented in (Bletsas, 2007). For *SFP*, we provided the task priorities using RM as an input to their optimization problem but let it find the optimal phase assignment. We also removed the constraint $R_i \leq D_i$ in the solver described in (Kim et al., 2013b) since the goal of our experiments is not to check the schedulabity of the system *per se* but comparing the actual worst-case response time computed with each analysis. It is worth mentioning that *SFP* deemed a few task sets infeasible, in which case, in order to maintain a fair comparison, they were discarded in the evaluation. For the task model evaluated in this chapter, *Split* boils down to force all higher priority tasks to have synchronous releases with each of the execution regions of the self-suspending task. *Joint* is the traditional suspension-oblivious analysis for fixed-priority scheduling, that is, assuming that the suspension regions are part of the worst-case execution time of the task. Both *Split* and *Joint* are simple response time tests that yield well-known upper bounds and that can be computed straightforwardly. The analysis from (Liu and Anderson, 2013) was not considered as the authors acknowledged the existence of a flaw. All the plots of Fig. A.5 represent the inaccuracy of the previous timing analysis techniques when compared to our optimal method (for Fig. A.5a) or our optimization problem (for Fig. A.5b–f).

Herein we restrict our attentions to problem instances that are representative in size of many real-time systems, in order to study the applicability and trade-offs of the different analysis towards specific parameter intervals. For the first set of experiments presented on Fig. A.5a–c, we fixed the number of execution regions of $\tau_{\text{ss}}$ to 2 and we varied the number of tasks from 4 to 12. The suspension length of $\tau_{\text{ss}}$ was set at a ratio $\frac{S_i}{T_i}$ with values 0.1, 0.3, and 0.5. Fig. A.5a–c show the average gain achieved by our analysis with respect to the WCRT when varying the utilization, the length of the suspension and the number of interfering tasks, respectively. As expected, our MILP

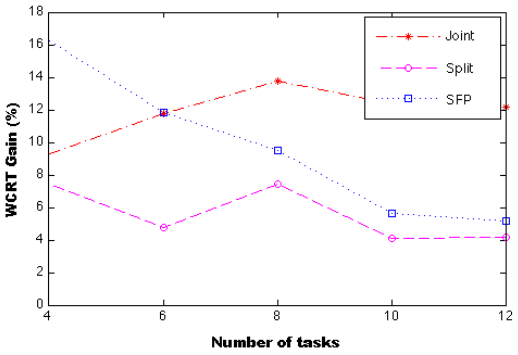
formulation (denoted "Opt") and our exact analysis always outperform the other approaches with average gains varying from 1 to 30% relatively to *Joint* and *Split*, and considerably more for *SFP*. The difference with *SFP* can be explained by the fact that the optimization problem of (Kim et al., 2013b) was formulated to find schedulable solutions and not necessarily reduce the response time of the self-suspending tasks. Maximum gains (which is not represented on the plots) observed during our experiments showed that our method can reduce the pessimism on the WCRT over 120% in some cases but is around 70% for utilizations around 0.7, i.e., close to the schedulability bound of RM for non-self-suspending tasks. However, the gains are highly dependent on the utilization of the system and the length of the suspension region as can be seen on Fig. A.5a and A.5c.

*Joint* performs relatively poorly when the suspensions become longer and the utilization increases, but is competitive in the presence of short suspensions and low utilizations. *Split* exhibits the opposite behavior as it is unlikely that the higher priority tasks happen to be released synchronously with both execution regions when the suspension is short and the system is not highly loaded. Although our exact algorithm becomes intractable for $n > 8$ (it is only part of Fig. A.5a), *Opt* is able to also provide the exact WCRT for the majority of the task sets as illustrated in Fig. A.5f. In order to verify if a given WCRT is the exact solution, one must check if the response time of each execution region converges to the number of interfering job assumed to be released inside their window.

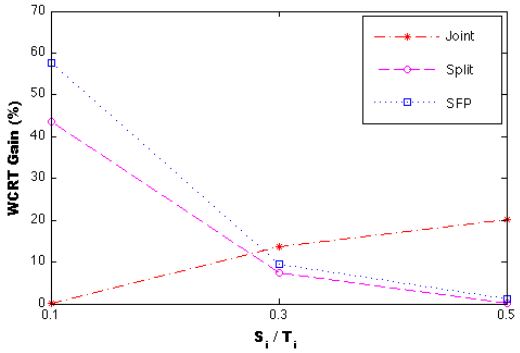
We then study the importance of different suspension ratios and how it relates with multiple suspension regions. Accordingly, the second set of experiments had the number of tasks fixed to 6 and the number of execution regions varying from 2 to 5. The suspension length was instead set as a ratio $\frac{S_i}{C_i}$ with values 1 and 2. The results are depicted in Fig. A.5d and A.5e. *SFP* could not find significant feasible solutions and thus was excluded in the evaluation. It can be observed that an increase in the number of suspension regions, when not accompanied by a substantial increase in the ratio $\frac{S_i}{C_i}$, leads to a severe degradation for the *Split*'s performance, just attenuated for high utilizations which typically result in smaller offsets due to the increase in response time. *Joint* remains very competitive even when the suspension is twice the length of the execution time of the task. Throughout the experiments, it becomes clear that *Joint* and *Split* are not comparable, but also that for such low run-time complexity both approaches may yield tight upper-bounds when applied over this model of tasks. As a last remark, we note that our optimization problem takes in average a few seconds to find its best solution and that the computational time remain reasonable (below 10s) for reasonable dimension of the problem (12 tasks for 2 execution regions, or 6 tasks for 5 execution regions), although under highly loaded circumstances the solver may struggle to improve over the initial solutions of some specific problem instances, in which case a timer may limit the research time.



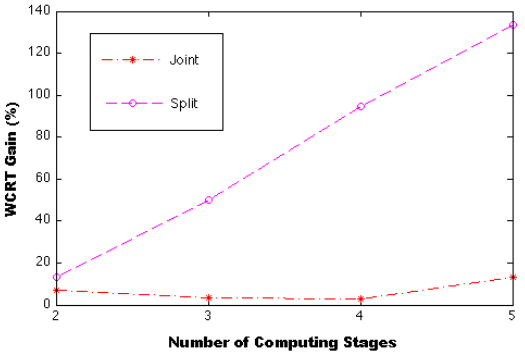

(a) Varying total utilization when $n = 6$ and $\frac{S_{ss}}{T_{ss}} = 0.3$.



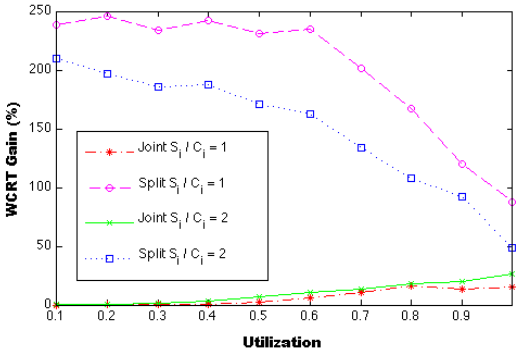

(b) Varying number of tasks when $U_{\text{tot}} = 0.6$ and $\frac{S_{ss}}{T_{ss}} = 0.3$.



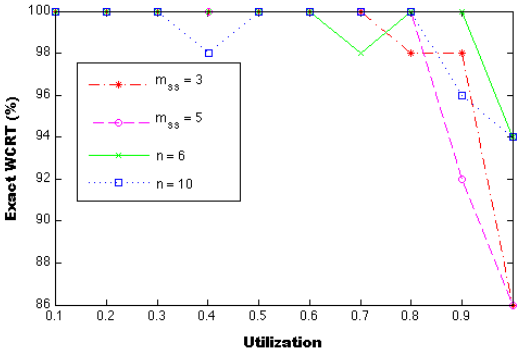

(c) Varying ratio $\frac{S_{ss}}{T_{ss}}$ when $U_{\text{tot}} = 0.6$ and $n = 6$.




(d) Varying number of execution regions when $U_{\text{tot}} = 0.7$, $n = 6$ and $\frac{S_{ss}}{C_{ss}} = 2$.




(e) Varying utlization when $n = 6$, $m_{ss} = 5$ and $\frac{S_{ss}}{C_{ss}}$ is equal to 1 and 2.




(f) Percentage of exact solutions found by the optimization problem.

Figure A.5: Experimental results.

## A.9 Summary

In this chapter, we considered the fixed-priority scheduling of a set of self-suspending tasks onto uniprocessors. We have demonstrated that it is not simple to characterize a critical instant for sporadic tasks with self-suspensions, thereby invalidating a claim made in an earlier work. We highlighted the complexity of the problem and presented an algorithm to compute the exact WCRT of a lower priority self-suspending task when scheduled together with sporadic non-self-suspending tasks. As the algorithm rapidly becomes intractable for a large number of higher priority tasks due to the exponential number of scenarios that need to be considered, we formulated a response time test for multiple suspension regions as an optimization problem that can be solved by a MILP tool in reasonable time. The optimization problem was then extended to accommodate multiple sporadic self-suspending tasks interfering with each other. Experiment results showed that the proposed response time tests dominate state-of-the-art techniques, although the WCRT gains highly depend on the peculiarities of the task sets. Experiments also pointed out that the optimization problem finds the exact WCRT solution in the majority of the cases.

# Appendix B

# Counter Examples

In this appendix, we present two counter-examples to the fixed-priority response time analysis for fork-join tasks scheduled in a partitioned fashion on a multiprocessor system presented in Axer et al. (2013), thereby proving its incorrectness.

## B.1 Counter-examples to the schedulability analysis for partitioned fork-join tasks presented in Axer et al. (2013)

In Axer et al. (2013), a sporadic fork-join $\tau$ is represented by a set of subtasks, where each subtask $\tau^{s,\sigma}$ belongs to the $\sigma^{th}$ segment of the $s^{th}$ stage. Subtasks on stage $s+1$ cannot start before all subtask on stage $s$ finish their executions. A sequential task is then $\tau^{1,\sigma}$. The WCET of a subtask is given by $C^{s,\sigma}$, while $\sigma$ also denotes the core to which the subtask is mapped.

Following Theorem 3 and Equation 11 as defined in Axer et al. (2013), the worst-case response time of a sequential task $\tau_i$ assigned to core $\sigma = 1$, when suffering interference from a higher priority fork-join task $\tau_j$, is given by

$$R_i = C_i^{1,1} + N_j \times \sum_{\forall \tau_j^{s,\sigma}} C_j^{s,1} \tag{B.1}$$

where $N_j$ is the number of jobs released by $\tau_j$ in the scheduling window $[0, R_i)$. That is, all stages are assumed to be activated with the activation of a job of $\tau_j$, and the first job of $\tau_j$ is released synchronously with the sequential task $\tau_i$.

Now, we provide a counter-example to Equation B.1.

**Example 20.** *Consider a task set with three tasks and a platform with two cores. Task $\tau_1$ is sequential with $C_1^{1,2} = 5$, period $T_1 = 100$, and has high priority. Task $\tau_2$ is fork-join with $C_2^{1,1} = 0$, $C_2^{2,1} = 1$, $C_2^{2,2} = 1$, $C_2^{3,1} = 2$, period $T_2 = 8$, and has medium priority. Task $\tau_3$ is sequential with $C_3^{1,1} = 1$, and has low priority. Using Equation B.1, the worst-case response time of the sequential task $\tau_3$ is $R_3 = 4$, as illustrated in Fig. B.1. Fig. B.2 shows a different schedule for the same task*

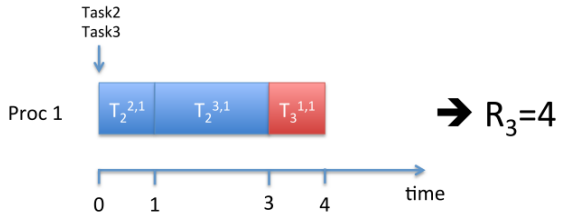



Figure B.1: Worst-case schedule according to Equation B.1 for Example 20.

*set but with a different arrival pattern, while also considering the interference happening on core 2. Clearly, $R_3$ is actually 6.*

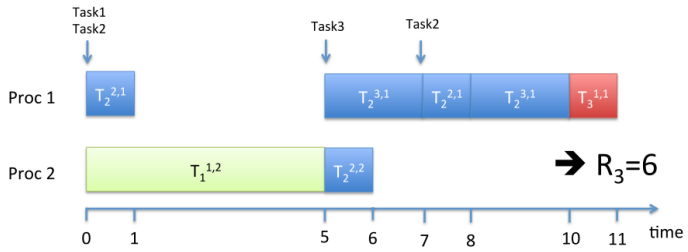



Figure B.2: Counter-example to Equation B.1 for Example 20.

Following Theorems 3 and 4 in Axer et al. (2013), the worst-case arrival pattern for a sequential task $\tau_j$ interfering with a fork-join task $\tau_i$ is achieved when $\tau_j$ (1) releases the first job synchronously with the first stage of $\tau_i$ which contains a segment $\sigma = p$ and $\tau_j$ is assigned to $p$; (2) subsequent releases are separated by $T_j$ time units unless no subtask $\tau_i^{s,p}$ is active at that time, in which case $\tau_j$ is delayed to be released synchronously with the activation of the next subtask $\tau_i^{s,p}$.

Now, we provide a counter-example to Theorems 3 and 4 in Axer et al. (2013).

**Example 21.** *Consider a task set with three tasks and a platform with two cores. Task $\tau_1$ is sequential with $C_1^{1,1} = 1$, period $T_1 = 100$, and has high priority. Task $\tau_2$ is sequential with $C_2^{1,1} = 1$, period $T_2 = 4$, and has medium priority. Task $\tau_3$ is fork-join with $C_3^{1,1} = 1$, $C_3^{2,1} = 1$, $C_3^{2,2} = 2$, $C_3^{3,1} = 3$, and has low priority. By applying Theorems 3 and 4 as defined in Axer et al. (2013), we obtain that the worst-case response time of the fork-join task $\tau_3$ is $R_3 = 9$, as illustrated in Fig. B.3. Fig. B.4 shows a different schedule for the same task set but with a different arrival pattern. Clearly, $R_3$ is actually 10.*

The examples provided above prove that the analysis presented in Axer et al. (2013) is incorrect, both when a fork-join task interferes with other tasks and when it is the task under analysis.

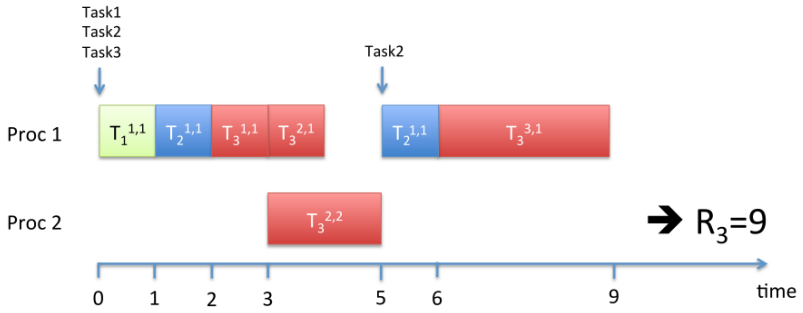



Figure B.3: Worst-case schedule according to Theorems 3 and 4 in Axer et al. (2013) for Example 21.

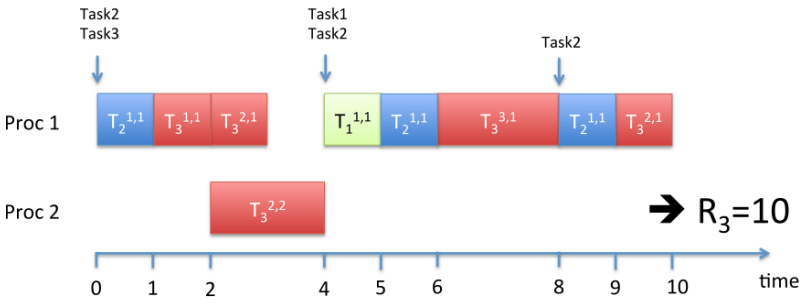



Figure B.4: Counter-example to Theorems 3 and 4 in Axer et al. (2013) for Example 21.

# References


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 *ECRTS*, July 2013, pp. 215–224.

A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *ECRTS*, July 2015.

J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.

F. Cerqueira, G. Nelissen, and B. B. Brandenburg, "On Strong and Weak Sustainability, with an Application to Self-Suspending Real-Time Tasks," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, vol. 106, 2018, pp. 26:1–26:21.

C. L. Liu, "Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment," *JPL Space Programs Summary 37-60*, vol. II, pp. 28–31, 1969.

A. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronisation," *IEEE Transaction on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. Softw. Eng.*, vol. 19, no. 9, pp. 920–934, Sep. 1993.

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

M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

S. Baruah and A. Burns, "Sustainable scheduling analysis," in *RTSS*, Dec 2006, pp. 159–168.

K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–151, Mar 1994.

M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," in *Proceedings of IFIP Congress (IFIP'74)*, 1974, pp. 807–813.

K. Albers and F. Slomka, "Efficient feasibility analysis for real-time systems with edf scheduling," in *Design, Automation and Test in Europe*, March 2005, pp. 492–497 Vol. 1.

L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Syst.*, vol. 28, no. 2-3, pp. 101–155, Nov. 2004.

J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*, 2004.

S. K. Dhall, "Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems." Ph.D. dissertation, 1977.

C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 140–149.

R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Survey*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *RTSS*, Dec 2007, pp. 149–160.

T. P. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *RTSS*, Dec 2003, pp. 120–129.

J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems Journal*, vol. 25, pp. 187–205, September 2003.

M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of edf on multiprocessor platforms," in *ECRTS*, July 2005, pp. 209–218.

T. P. Baker and S. K. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," in *Handbook of Realtime and Embedded Systems*, 2007.

N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Syst.*, vol. 45, no. 1-2, pp. 26–71, Jun. 2010.

A. Srinivasan and J. H. Anderson, "Optimal rate-based scheduling on multiprocessors," *J. Comput. Syst. Sci.*, vol. 72, no. 6, pp. 1094–1117, Sep. 2006.

T. Ungerer, F. J. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa®-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654 – 1663, 2013.

Intel, "Intel many integrated core architecture," february 2017. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.

Intel, "Intel parallel building blocks," september 2010. [Online]. Available: http://software.intel.com/en-us/articles/intel-parallel-building-blocks/

OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, 2000, pp. 36–43.

K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010, pp. 259–268.

S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012, pp. 63–72.

J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Analysis of global EDF for parallel tasks," in *ECRTS*, 2013, pp. 3–13.

A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *RTSS*, 2011, pp. 217–226.

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 *ECRTS*, July 2013, pp. 25–34.

B. Andersson and D. de Niz, "Analyzing global-edf for multiprocessor scheduling of parallel tasks," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, 2012, vol. 7702, pp. 16–30.

parMESARA Project, "Multi-core execution of parallelised hard real-time applications supporting analysability," 2014. [Online]. Available: http://www.parmerasa.eu/

P-SOCRATES Project, "Parallel software framework for time-critical many-core systems," 2016. [Online]. Available: http://www.p-socrates.eu/

J. Fonseca, G. Nelissen, and V. Nelis, "Improved response time analysis of sporadic dag tasks for global fp scheduling," in *RTNS'17*, 2017.

A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 339–353, Feb 2017.

J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic dag tasks under partitioned scheduling," in *SIES'16*, 2016.

J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel application with conditional execution," in *SAC*, Mar 2015.

S. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *RTAS*, 2002, pp. 154–163.

G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis, "Timing analysis of fixed-priority self-suspending sporadic tasks," in *ECRTS'15*, July 2015.

Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *J. Parallel Distrib. Comput.*, vol. 59, no. 3, pp. 381–422, Dec. 1999.

M. Drozdowski, *Scheduling for Parallel Processing*, 1st ed. Springer Publishing Company, Incorporated, 2009.

G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *J. ACM*, vol. 46, no. 2, pp. 281–321, 1999.

Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.

T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 9, pp. 951–967, Sep. 1994.

I. Ahmad and Y.-K. Kwok, "On exploiting task duplication in parallel program scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872–892, Sep 1998.

C.-C. Han and K.-J. Lin, "Scheduling parallelizable jobs on multiprocessors," in *RTSS*, dec 1989, pp. 59 –67.

Q. Wang and K. H. Cheng, "A heuristic of scheduling parallel tasks and its analysis," *SIAM J. Comput.*, vol. 21, no. 2, pp. 281–294, Apr. 1992.

J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," in *RTSS*, 2006, pp. 89–100.

J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *ECRTS*, 2009, pp. 194–204.

J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," *CoRR*, vol. abs/1006.2617, 2010.

G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks inreal-time multiprocessor systems," *Real-Time Systems Journal*, vol. 15, pp. 39–60, July 1998.

S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS*, 2009, pp. 459–468.

W. Y. Lee and H. Lee, "Optimal scheduling for real-time parallel tasks," *Transactions on Information and Systems*, vol. E89-D, pp. 1962–1966, June 2006.

S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Inf. Process. Lett.*, vol. 106, no. 5, pp. 180–187, May 2008.

K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, pp. 117–134, Apr. 1994.

J. Palencia, J. G. Garcia, and M. Harbour, "On the schedulability analysis for distributed hard real-time systems," in *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, Jun 1997, pp. 136–143.

J. Palencia and M. Gonzalez Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *RTSS*, Dec 1998, pp. 26–37.

J. Palencia and M. Harbour, "Exploiting precedence relations in the schedulability analysis of distributed real-time systems," in *RTSS'99*, 1999.

J. C. Palencia and M. G. Harbour, "Offset-based response time analysis of distributed systems scheduled under edf," in *ECRTS'03*, 2003, pp. 3–12.

M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *RTAS*, April 2011, pp. 71–80.

J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *ICCPS*, April 2013, pp. 31–40.

C. Maia, P. M. Yomsi, L. Nogueira, and L. M. Pinho, "Real-time semi-partitioned scheduling of fork-join tasks using work-stealing," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, p. 31, Sep 2017.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, July 2012, pp. 321–330.

B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *J. ACM*, vol. 47, no. 4, pp. 617–643, Jul. 2000.

C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *RTNS*, 2014, pp. 3–12.

V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *ECRTS*, July 2013, pp. 225–233.

S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," in *ECRTS*, July 2014, pp. 97–105.

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*, vol. 25, no. 12, pp. 3242–3252, Dec 2014.

M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," in *RTNS*, 2013, pp. 287–296.

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*, ser. RTNS '14, 2014, pp. 13:13–13:22.

X. Jiang, X. Long, N. Guan, and H. Wan, "On the decomposition-based global edf scheduling of parallel real-time tasks," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016, pp. 237–246.

M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, "An analysis of lazy and eager limited preemption approaches under dag-based global fixed priority scheduling," in *IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, 2017, pp. 193–202.

A. Parri, A. Biondi, and M. Marinoni, "Response time analysis for g-edf and g-dm scheduling of sporadic dag-tasks with arbitrary deadline," in *RTNS'15*, 2015, pp. 205–214.

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 *ECRTS*, July 2014.

S. Baruah, "The federated scheduling of constrained-deadline sporadic dag task systems," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15, 2015, pp. 1323–1328.

——, "Federated scheduling of sporadic dag task systems," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15, 2015, pp. 179–186.

J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu, "Mixed-criticality federated scheduling for parallel real-time tasks," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," in *RTSS'17*, 2017.

L. Nogueira and L. M. Pinho, "Server-based scheduling of parallel real-time tasks," in *Proceedings of the tenth ACM international conference on Embedded software*, 2012, pp. 73–82.

J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu, "Randomized work stealing for large scale soft real-time systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016, pp. 203–214.

K. Yang, M. Yang, and J. H. Anderson, "Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16, 2016, pp. 349–358.

S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "The global edf scheduling of systems of conditional sporadic dag tasks," in *ECRTS'15*, 2015.

S. Baruah, "The federated scheduling of systems of conditional sporadic dag tasks," in *Proceedings of the 12th International Conference on Embedded Software*, ser. EMSOFT '15, 2015, pp. 1–10.

X. He and Y. Yesha, "Parallel recognition and decomposition of two terminal series parallel graphs," *Information and Computation*, vol. 75, no. 1, pp. 15–38, 1987.

J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *STOC'79*, 1979, pp. 1–12.

A. González-Escribano, A. J. C. Van Gemund, and V. Cardeñoso Payo, "Mapping unstructured applications into nested parallelism," in *VECPAR'02*, 2002, pp. 407–420.

N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *30th IEEE Real-Time Systems Symposium*, Dec 2009, pp. 387–397.

E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, May 2005.

S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS*, 2015, pp. 129–138.

K. Bletsas, "Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism," Ph.D. dissertation, University of York, Department of Computer Science, 2007.

D. Bozdağ, F. Özgüner, and U. V. Catalyurek, "Compaction of schedules and a two-stage approach for duplication-based dag scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 6, pp. 857–871, Jun. 2009.

J. M. López, J. L. Díaz, and D. F. García, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Systems*, vol. 28, no. 1, pp. 39–68, Oct 2004.

S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, ser. RTSS '05, 2005, pp. 321–329.

D. S. Johnson, "Fast algorithms for bin packing," *J. Comput. Syst. Sci.*, vol. 8, no. 3, pp. 272–314, Jun. 1974.

S. K. Baruah and N. W. Fisher, "The partitioned dynamic-priority scheduling of sporadic task systems," *Real-Time Systems*, vol. 36, no. 3, pp. 199–226, Aug 2007.

Universidad de Cantabria, SPAIN, "Modeling and analysis suite for real-time applications MAST 1.5.0," 2014. [Online]. Available: http://mast.unican.es

P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS Workshop*, 2010, pp. 6–11.

Y. Sun and M. Di Natale, "Assessing the pessimism of current multicore global fixed-priority schedulability analysis," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, 2018, pp. 575–583.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

S. K. Baruah, "A general model for recurring real-time tasks," in *RTSS'98*, Dec 1998, pp. 114–122.

M. Anand, A. Easwaran, S. Fischmeister, and I. Lee, "Compositional feasibility analysis of conditional real-time task models," in *ISORC'08*, May 2008, pp. 391–398.

L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, Jul. 2004.

C. Lin, T. Kaldewey, A. Povzner, and S. Brandt, "Diverse soft real-time processing in an integrated system," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Dec 2006, pp. 369–378.

K. Lakshmanan and R. Rajkumar, "Scheduling self-suspending real-time tasks with rate-monotonic priorities," in *RTAS*, 2010, pp. 3–12.

J. J. Chen, G. Nelissen, W. H. Huang, M. Yang, B. Brandenburg, B. K., C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and D. de Niz, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Technical Report 854, Faculty of Informatik, TU Dortmund*, 2016.

J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

J. Kim, B. Andersson, D. de Niz, and R. Rajkumar, "Segment-fixed priority scheduling for self-suspending real-time tasks," in *RTSS*, Dec 2013, pp. 246–257.

C. Liu and J. H. Anderson, "Task scheduling with self-suspensions in soft real-time multiprocessor systems," in *RTSS*, Dec 2009, pp. 425–436.

C. Liu and J. Anderson, "An o(m) analysis technique for supporting real-time self-suspending task systems," in *RTSS*, Dec 2012, pp. 373–382.

C. Liu and J. H. Anderson, "Suspension-aware analysis for hard real-time multiprocessor scheduling," in *ECRTS*, 2013, pp. 271–281.

F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *RTSS*, Dec 2004, pp. 47–56.

F. Ridouard, P. Richard, F. Cottet, and K. Traoré, "Some results on scheduling tasks with self-suspensions," *Journal of Embedded Computing*, vol. 2, no. 3,4, pp. 301–312, Dec. 2006.

I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, and M.-P. Hong, "Real-time scheduling of tasks that contain the external blocking intervals," in *RTCSA*, Oct 1995, pp. 54–59.

C. Liu, J. J. Chen, L. He, and Y. Gu, "Analysis techniques for supporting harmonic real-time tasks with suspensions," in *ECRTS*, July 2014, pp. 201–210.

N. Audsley and K. Bletsas, "Fixed priority timing analysis of real-time systems with limited parallelism," in *ECRTS*, June 2004, pp. 231–238.

K. Bletsas and N. Audsley, "Extended analysis with reduced pessimism for systems with limited parallelism," in *RTCSA*, Aug 2005, pp. 525–531.

Gurobi Optimization Inc., "Gurobi optimizer reference manual," 2015. [Online]. Available: http://www.gurobi.com