PhD Thesis

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

Ricardo Garibay-Martínez

CISTER-TR-160407

2016/04/25

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

Ricardo Garibay-Martínez

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: rgmaz@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Real-time embedded systems are part of our everyday life. These systems range from the traditional areas of military and mission critical to domestic and entertainment applications. The aim of real-time systems is to execute applications in a way that those applications met their temporal constraints. Most of the results for real-time systems are based on the sequential real-time execution model, where intra-task parallelism is forbidden. In the last decade real-time researchers have started to focus their attention on the case of multi-threaded parallel real-time task models.In these models, concurrent real-time activities are allowed to become parallel and to be processed in more than one processor at the same time.

Despite the fact that parallel computations in multi-processors can offer an increased processing capacity, the multi-threaded parallel real-time task models have not considered the cases in which a distributed execution also exists. Such execution pattern can be used to provide even more capabilities and processing power. Furthermore, in some applications the use of parallel distributed computations is the only possibility in which the applications can comply with their time constraints. An example of such type of applications are modern automotive applications, which need to execute computational intensive applications (e.g., infotainment or driver assistance applications). Consequently, design frameworks that allow the workload to be distributed in peak situations by both parallel and distributed processors are required. Such scenarios require the integration of distributed computations with parallel real-time models. In a distributed system, the transmission delay of messages cannot be considered negligible as in the case of multi-processors systems, therefore, their impact on the schedulability of the system has to be considered.

The fork-join Parallel/Distributed real-time model (P/D task) proposed in this thesis is designed to consider such execution pattern. The P/D task model is derived from observing the execution of parallel and distributed programs (e.g. OpenMP, MPI). The thesis proposes a scheduling algorithm for P/D tasks, the parallel/Distributed Deadline Monotonic Scheduling (P/D-DMS). The thesis also proposes two heuristics for task partitioning and priority assignment for the linear transactional model and the P/D tasks model, the Distributed using Optimal Priority Assignment (DOPA) heuristic and the Parallel-DOPA (P-DOPA) heuristic, respectively.

A holistic analysis technique for P/D tasks is also proposed. The holistic approach allows to consider the systems as a whole, and an improved analysis is proposed based on that holistic view of the system. The allocation of P/D tasks is later extended by considering distributed multi-core nodes. The extension is based on the constraint programming approach.

The analysis and proposals presented in this thesis are validated through simulations and an experimental evaluation.

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

Ricardo Garibay Martínez

Programa Doutoral em Engenharia Electrotécnica e de Computadores

Supervisor: Prof. Dr. Luis Miguel Moreira Lino Ferreira

Co-supervisor: Prof. Dr. Luís Miguel Rosário da Silva Pinho

April 25, 2016

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

Ricardo Garibay Martínez

Programa Doutoral em Engenharia Electrotécnica e de Computadores

April 25, 2016

Resumo

Os sistemas embebidos e de tempo real são parte de nossa vida cotidiana. Estes sistemas abrangem desde as áreas tradicionais dos sistemas militares e sistemas críticos, até às aplicações domésticas e de entretenimento. Estes sistemas têm como um dos requisitos principais executar as suas aplicações respeitando prazos temporais, para além das características funcionais.

A maioria das análises temporais para sistemas de tempo real é baseada no modelo de execução sequencial, onde não é permitida a execução paralela de uma tarefa. Na última década, esta restrição tem sido levantada, e os trabalhos na área começaram a prestar mais atenção aos modelos paralelos. Nestes trabalhos as atividades paralelas e de tempo real são permitidas e podem ser executadas nos processadores disponíveis do sistema.

Apesar do facto de que a computação paralela em sistemas multi-processador pode oferecer uma maior capacidade de processamento, os modelos paralelos propostos para sistemas de tempo real não têm considerado os casos em que também é possível a execução distribuída. Este padrão de execução pode ser usado para proporcionar maior capacidade de processamento, sem necessidade de cada nó computacional por si ser escalado. Além disso, em alguns sistemas a utilização conjunta de processamento paralelo e distribuído é a única possibilidade de garantir os requisitos temporais do sistema.

Um exemplo deste tipo de aplicações são os sistemas computacionais utilizados nos automóveis modernos, em que se dispõe de redes de sistemas computacionais embebidos, com nós que incluem pequenos processadores até sistemas multi-processador, e que podem ter que executar aplicações computacionalmente pesadas (por exemplo, aplicações de *infotainment* ou de assistência ao condutor).

Tais cenários exigem a integração de computação distribuída com modelos paralelos para sistemas de tempo real. Num sistema distribuído, o atraso na transmissão de mensagens não pode ser considerados como desprezável, tal como é normal fazer no caso de sistemas multi-processador. Consequentemente, o seu impacto no desenho do sistema tem que ser considerado.

O modelo de tempo real *fork/join* paralelo/distribuído, proposto nesta tese, tem em conta assim o padrão de execução descrito. Este modelo é derivado da análise de programas paralelos e distribuídos com base em arquiteturas globalmente utilizadas como sejam o OpenMP e o MPI.

A dissertação propõe também um algoritmo de escalonamento para tarefas paralelas/distribuídas – o algoritmo *Parallel/Distributed Deadline Monotonic Scheduling* (P/D-DMS). São também propostas duas heurísticas para o particionamento de tarefas e atribuição de prioridades para o modelo de execução linear e para o modelo paralelo/distribuído

– as heurísticas *Distributed using Optimal Priority Assignment* (DOPA) e *Parallel*-DOPA (P-DOPA).

A partição de tarefas paralelas/distribuídas é também estendida, considerando os nós distribuídos com múltiplos cores e numa abordagem de programação por restrições.

Finalmente, é proposta uma técnica de análise holística para tarefas paralelas/distribuídas. A abordagem holística permite considerar os sistemas como um todo, e permite obter resultados menos pessimistas.

Finalmente, as propostas e análises apresentadas nesta tese são validadas por meio de simulações e uma implementação experimental.

Abstract

Real-time embedded systems are part of our everyday life. These systems range from the traditional areas of military and mission critical to domestic and entertainment applications. The aim of real-time systems is to execute applications in a way that those applications met their temporal constraints.

Most of the results for real-time systems are based on the sequential real-time execution model, where intra-task parallelism is forbidden. In the last decade real-time researchers have started to focus their attention on the case of multi-threaded parallel real-time task models. In these models, concurrent real-time activities are allowed to become parallel and to be processed in more than one processor at the same time.

Despite the fact that parallel computations in multi-processors can offer an increased processing capacity, the multi-threaded parallel real-time task models have not considered the cases in which a distributed execution also exists. Such execution pattern can be used to provide even more capabilities and processing power. Furthermore, in some applications the use of parallel distributed computations is the only possibility in which the applications can comply with their time constraints. An example of such type of applications is modern automotive applications, which need to execute computational intensive applications (e.g., infotainment or driver assistance applications).

Consequently, design frameworks that allow the workload to be distributed in peak situations by both parallel and distributed processors are required. Such scenarios require the integration of distributed computations with parallel real-time models. In a distributed system, the transmission delay of messages cannot be considered negligible as in the case of multi-processors systems, therefore, their impact on the schedulability of the system has to be considered.

The fork-join Parallel/Distributed real-time (P/D task) model proposed in this thesis is designed to consider such execution pattern. P/D task model is derived from observing the execution of parallel and distributed programs (e.g. OpenMP, MPI).

The thesis proposes a scheduling algorithm for P/D tasks, the Parallel/Distributed Deadline Monotonic Scheduling (P/D-DMS). The thesis also proposes two heuristics for task partitioning and priority assignment for the linear transactional model and the P/D tasks model, the Distributed using Optimal Priority Assignment (DOPA) heuristic and the Parallel-DOPA (P-DOPA) heuristic, respectively.

A holistic analysis technique for P/D tasks is also proposed. The holistic approach allows to consider the systems as a whole, and an improved analysis is proposed based on that holistic view of the system.

The allocation of P/D tasks is later extended by considering distributed multi-core nodes. The extension is based on the constraint programming approach. The analysis and proposals presented in this thesis are validated through simulations and an experimental evaluation.

Acknowledgments

A doctoral research work and writing a Ph.D. thesis requires long commitment, hard work and perseverance. This is an exciting challenge that I had enjoyed and embraced most of the time, however, sometimes it becomes a difficult path in which without the support of key persons, it would be simply impossible to complete. I would like to express my appreciation to those special persons that have supported me during these years of Ph.D. work.

First of all, I would like to express my great appreciation and gratitude to my supervisors, Luis Lino Ferreira and Luís Miguel Pinho for giving me the opportunity and helping me to complete this challenging task. For sharing their experience and for the innumerable constructive discussions, useful feedback and guidance. I am grateful with Luis Lino Ferreira, he always found time to provide me with consistent, accurate and valuable feedback about my work. I highly appreciate his big patience and efforts in all the matters related to this work. I am grateful with Luis Miguel Pinho, for his problem solving skills and his capacity of finding accurate solutions whenever I felt in a dead-end path. Thank you both for always providing me with funding, that made me feel secure and permitted me to focus on my work.

I am extremely thankful to my colleague and friend Geoffrey Nelissen. I would like to thank him for being a great source of support, both academically and personally. Geoffrey possesses great technical knowledge and always has an interesting and pragmatic approach towards the solution to very challenging problems. It is certain that without his help the quality of this work would not have been the same.

I would also like to thank the board of directors of CISTER, Eduardo Tovar, Luis Miguel Pinho and Filipe Pacheco for providing me with an ideal research environment here at CISTER. I am grateful to Ines Almeida, Sandra Almeida and Cristiana Barros for being always available to help me with administrative processes, especially the visa-related ones.

I wish to acknowledge the constructive technical discussions with Artem Burmyakov, Vincent Nelis, Patrick Yomsi and Michele Albano, on different topics and at different stages of this work.

I am grateful with all my colleagues at CISTER for sharing experiences, thoughts and knowledge with me through all these years. Thanks to Maryam, Hossein, Andree Pedro, Joao, Claudio, Patrick, Ramiro, Damien, Maria Angeles, Ali, Antonio, Artem, Shashank, Kostiantyn, Jose Fonseca, Hazem, Michele, Konstantinos, Vincent, Geoffrey, David, Nuno, Jose Marinho, Dakshina, Guru and Vikram.

I would like to thank the B.S. students with whom I had the chance to work, Tomas

Sotomaior, Fábio Oliveira and Roberto Duarte. Thank you for your efforts and contribution to this work, as well as for reminding me that teaching is one of the things I enjoy doing the most in life.

Apart of the professional side I would like to thank my family and friends who made this stage of my life a memorable and enjoyable experience. I wish to thank them and express my appreciation for all the laughs and tears, I hope to keep you with me for all my life. Many thanks to my dear friends: Vincent Nelis, Anna Zieba, Iulia Ilca, Maria Angeles Serna, Artem Burmyakov, Joao Loureiro, Nuno Pereira, Geoffrey Nelissen, Vikram Gupta, Shashank Gaur, Patrick Yomsi, Vanessa Merolla, Ramiro Robles, Andrea Baldovin and Damien Masson. *Muchas gracias a mi mamá Elia Martínez Rosas y a mi papá Ricardo Garibay Bonales por su apoyo incondicional y por siempre confiar en mí. Los quiero. Muchas gracias a toda mi familia por todo.* I would like to thank you all for bearing with me in what is definitely the biggest challenge of my life until now. Thank you for everything.

This work was partially supported by FCT/MEC and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71562/-2010.

Sincerely,
Ricardo Garibay Martínez

“To go too far is as bad as to fall short”

Confucius

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Program Interface
BSF	Breadth Search First
CAN	Controller Area Network
DM	Deadline Monotonic
DBF	Demand Bound Function
DEP	Distributed Execution Path
D-Fork	Distributed-Fork
D-Join	Distributed-Join
DAG	Directed Acyclic Graph
DOPA	Distributed using Optimal Priority Assignment
DST	Distributed Stretch Transformation
EC	Elementary Cycle
ECU	Electronic Control Unit
EDF	Earliest Deadline First
FBB-FFD	Fisher-Baruah-Baker First-Fit-Decreasing
F-J	Fork-Join
FTT	Flexible Time Triggered
FTT-SE	Flexible Time Triggered Switched Ethernet
HOPA	Heuristic Optimized Priority Assignment
LIN	Local Interconnect Network
MPI	Message Passing Interface
NR-T	Non-Real-Time
OPA	Optimal Priority Assignment
OpenMP	Open Multi-Processing
P/D	Parallel/Distributed
P/D-DMS	Parallel/Distributed Deadline Monotonic Scheduling
P-DOPA	Parallel-Distributed using Optimal Priority Assignment
RM	Rate Monotonic
RMA	Rate Monotonic Analysis
R-T	Real-Time
RTE	Real-Time Ethernet
RSE	Rear Seat Entertainment
SST	Segment Stretch Transformation
SWaP	Size, Weight and Power
TM	Trigger Message
TST	Task Stretch Transformation
WCET	Worst-Case Execution Time
WCML	Worst-Case Message Length

WCRT	Worst-Case Response Time

List of Symbols

f_i	Capacity of a task τ_i
$b_{i,j,k}$	Code block k in a segment $\sigma_{i,j}$
$GCB(V,E)$	Code block DAG
$\mu^{cd}_{i,j,k}$	Constrained deadline message $\mu_{i,j,k}$
$\theta^{cd}_{i,j,k}$	Constrained deadline thread $\theta_{i,j,k}$
$d^{Path}_{i,j}$	Deadline of DEP i,j
D^{master}_i	Deadline of a master thread τ^{master}_i
$d_{i,j}$	Deadline of a segment $\sigma_{i,j}$
$d_{i,2j}$	Deadline of a P/D segment $\sigma_{i,2j}$
$d_{i,2j+1}$	Deadline of a sequential segment $\sigma_{i,2j+1}$
$DBF(\tau_i,t)$	Demand bound function of task τ_i respect to time t
δ_i	Density of a task τ_i
$\delta_{i,j,k}$	Density of a thread $\theta_{i,j,k}$
$\delta^{msg}_{i,j,k}$	Density of a message $\mu_{i,j,k}$
$DEP_{i,2j,k}$	Distributed Execution Path
$l^d_{i,j}$	Download link connecting the nodes π_i to the switch SW_x
D_i	End-to-end deadline of task τ_i
e_i	External event
$z(t)$	First elements in $G^{sort}_{i,j,k}(t)[l]$ in the interval of time $[0,t]$
$J_{\mu_{i,j,k}}$	Jitter of a thread $\mu_{i,j,k}$
$J_{\theta_{i,j,k}}$	Jitter of a thread $\theta_{i,j,k}$
LW	Length of the specific transmission window
τ^{master}_i	Master thread of task τ_i
C_i	Maximum execution length of task τ_i
C^v_i	Maximum execution length of task τ_i executed on a processor v times faster
$\delta^{\max}_i$	Maximum density of a task τ_i
$\delta^{\max}_{i,j,k}$	Maximum density of a thread $\theta_{i,j,k}$
I	Maximum inserted iddle time in such window
δ_{max}	Maximum load ratio of a task τ_i
$nb^{\max}_{i,j,k}$	Maximum number of code blocks per thread $\theta_{i,j,k}$
$C^{\max}_{i,j,k}$	Maximum WCET of a code block $b_{i,j,k}$
$C^{\max}_{\theta_{i,j,k}}$	Maximum WCET of a thread $\theta_{i,j,k}$ in a P/D region $\sigma_{i,2j}$
$\mu_{i,j,k}$	Message k of a P/D segment $\sigma_{i,2j}$ of a task τ_i
$\delta^{\min}_i$	Minimum density of a task τ_i
$\delta^{\min}_{i,j,k}$	Minimum density of a thread $\theta_{i,j,k}$
η_i	Minimum execution length of task τ_i
η^v_i	Minimum execution time of a task τ_i executed on a processor v times faster

$G_{i,j,k}(t)[l]$	Multi-dimensional array containing the switch delays SD_x when a message crosses a switch in the network
π	Node set
π_i	Node i
$\pi_{i,s}$	Processor s in node i
$OvJ_{\pi_i}^{l^U_{SW_x}}$	Non-interference during the D-Join operation
$l_{i,j}$	Number of code blocks in a segment $\sigma_{i,j}$
m	Number of nodes π_l
m_i	Number k of threads $\theta_{i,j,k}$ in each segment $\sigma_{i,j}$
n_i	Number j of segments $\sigma_{i,j}$ composing task τ_i
$sn_{i,j,k}$	Number switches a message $\mu_{i,j,k}$ crosses from origin to destination
$n_{i,j}$	Number k of threads of segment $\sigma_{i,j}$ composing task τ_i
$i_{i,2j}$	Number of P/D threads $\theta_{i,2j,k}$, that each P/D segment $\sigma_{i,2j}$ can insert into the master thread τ_i^{master}, without causing task τ_i to miss its deadline D_i
$q_{i,2j}$	Number of remaining P/D threads $\theta_{i,2j,k}$, that that have not been coalesced into the master thread τ_i^{master}
$\phi_{i,j,k}$	Offset of a thread $\theta_{i,j,k}$
$\phi_{i,j}^{Path}$	Offset of DEP i,j
$\phi_{i,2j}$	Offset of a P/D segment $\sigma_{i,2j}$
$\phi_{i,2j+1}$	Offset of a sequential segment $\sigma_{i,2j+1}$
$\phi_{i,j,k}$	Offset of a thread $\theta_{i,j,k}$
$\phi_{i,j,k}^{msg}$	Offset of a message $\mu_{i,j,k}$
$OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$	Overlap on a download link connecting the last switch SW_x in the path of a message $\mu_{i,j-1,k}$ and a remote processor node π_i executing threads $\theta_{i,j,k}$
$OvF^*_{\pi_i,l^d_{SW_x}}(\theta_{i,j,k})$	Lower bound on the overlap $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$
P_i	Parallel execution length of a task τ_i
P_i^v	Parallel execution length of a task τ_i executed on a processor v times faster
T_i	Period or *minimum inter-arrival time* of task τ_i
$OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$	Pipeline effect during the D-Fork operation
$\sigma_{i,2j}$	P/D segment $2j$ of task τ_i
$\theta_{i,2j,k}$	P/D thread of segment $\sigma_{i,j}$ of task τ_i
$Wr_{i,j,k}(t)$	Remote link delay of a message $\mu_{i,j,k}$
$rbf_{i,j,k}(t)$	Request bound function of a messages $\mu_{i,j,k}$
t^*	Response time of a message $\mu_{i,j,k}$
S	Schedulability test
ϖ	Shared real-time network
$Wl_{i,j,k}(t)$	Shared link delay of message $\mu_{i,j,k}$
L_i	Slack time of task τ_i
$\sigma_{i,j}$	Segment j of task τ_i
$\sigma_{i,2j+1}$	Sequential segment $2j+1$ of task τ_i
$\theta_{i,2j+1,k}$	Sequential thread of segment $\sigma_{i,j}$ of task τ_i
$hp(\mu_{i,j,k})$	Set of messages $\mu_{i,j,l}$ with higher priority than task $\mu_{i,j,k}$
$lp(\mu_{i,j,k})$	Set of messages $\mu_{i,j,l}$ with lower priority than task $\mu_{i,j,k}$
$\mathscr{A}$	Set of threads that are resource constrained
Υ	Set of threads that do not have any resource constrained
$hp(\theta_{i,j,k})$	Set of threads $\theta_{i,j,l}$ with higher priority than task $\theta_{i,j,k}$
$lp(\theta_{i,j,k})$	Set of threads $\theta_{i,j,l}$ with lower priority than task $\theta_{i,j,k}$

$G^{sort}_{i,j,k}(t)[l]$	Sorted multi-dimensional array containing the switch delays SD_x when a message crosses a switch on the network
$IntT(\theta_{i,j,k})$	Subset of jobs that participated to the WCRT of a thread $\theta_{i,j,k}$, including itself a remote processor
$SLD_{i,j,k}$	Subset of messages that compose the shared link delay $Wl_{i,j,k}(t)$
$RLD_{i,j,k}$	Subset of messages that compose the remote link delay $Wr_{i,j,k}(t)$
$IntM(\theta_{i,j,k})$	Subset of messages that participated to the WCRT of a message $\mu_{i,j-1,k}$, and triggered the execution of the jobs in $IntT(\theta_{i,j,k})$
$WT(\mu_{i,j,k})$	Subset of messages that are scheduled in the same transmission window type compose as $\mu_{i,j,k}$
$sbf_{i,j,k}$	Supply bound function of a messages $\mu_{i,j,k}$
SFD_x	Store-and-Forward of switch SW_x
Δ	Switching-delay-effect of a message $\mu_{i,j,k}$
SW_x	Switch x of the network ϖ
SD_x	Switching delay when crossing a switch SW_x
δ_{sum}	System load of task set τ
δ^{v}_{sum}	System load of task set τ executed on a processor v times faster
u_{sum}	System utilisation of task set τ
τ	Task set
τ_i	Task i
$GT(E^*,V^*)$	Threads DAG
θ	Thread set
$\theta_{i,j,k}$	Thread k of segment $\sigma_{i,j}$ of task τ_i
t	Time instant t
$CP_{rbf_{i,j,k}}$	Time instants in which the $rbf_{i,j,k}$ is modified due to the interference of messages in $hp(\mu_{i,j,k})$
δ_{tot}	Total density of the tasks τ_i in the set τ
δ^{msg}_{tot}	Total density of the messages μ_i in the set τ
m_{tot}	Total number of cores in the distributed multi-core platform
U_{tot}	Total utilisation of the tasks τ_i in the set τ
$l^{u}_{i,j}$	Upload link connecting the nodes π_i to the switch SW_x
$l^{u}_{i,j}$	Vertex v_i of the graph $G=(V,E)$
$C^{b}_{i,j,k}$	WCET of a code block $b_{i,j,k}$
C^{master}_{i}	WCET of a master thread τ^{master}_{i}
$C_{i,j,k}$	WCET of thread $\theta_{i,j,k}$
$P_{i,j,k}$	WCET of a P/D thread $\theta_{i,j,k}$
$M_{i,j,k}$	WCML of message $\mu_{i,j,k}$ of P/D segment $\sigma_{i,2j}$
$WCRT^{msg}_{i,j,k}$	WCRT of message $\mu_{i,j,k}$
$WCRT_{i,j,k}$	WCRT of thread $\theta_{i,j}$
$r_{\mu_{i,j,k}}$	WCRT of a message $\mu_{i,j,k}$
$r^{asyn}_{\mu_{i,j,k}}$	WCRT of a asynchronous message $\mu_{i,j,k}$ transmitted with the FTT-SE Protocol
$r^{syn}_{\mu_{i,j,k}}$	WCRT of a synchronous message $\mu_{i,j,k}$ transmitted with the FTT-SE Protocol
$r^{v}_{\mu_{i,j,k}}$	WCRT of a message $\mu_{i,j,k}$ transmitted on a network v times faster
r_{τ_i}	WCRT of a task τ_i
$r_{\sigma_{i,2j}}$	WCRT of a Distributed Execution Path
$r_{\theta_{i,j,k}}$	WCRT of a thread $\theta_{i,j,k}$

Constrained Programming Decision Variables

$\mathsf{d_{i,j}}$	Artificial intermediate deadlines of a segment $\sigma_{i,j}$
$\mathsf{NV}(\theta_{\mathsf{i,j,k}})$	Function denoting to which node v_q a thread $\theta_{i,j,k}$ is mapped
$\mathsf{p^{a,b,c}_{i,j,k}}$	Higher priority relation of a thread $\theta_{a,b,c}$ on a thread $\theta_{i,j,k}$
$\Pi_{\theta_{\mathsf{i,j,k}}}$	Identifier of the core on which the thread $\theta_{i,j,k}$ is mapped
$\mathsf{IHP^{a,b,c}_{i,j,k}}$	Interference caused by a thread $\theta_{a,b,c}$ on a thread $\theta_{i,j,k}$
$\mathsf{ls_{i,j,k}}$	Maximum bounded switching delay of a message $\mu_{i,j,k}$
$\mathsf{z(t)}$	Number of ECs in an interval $[0,t]$
$\mathsf{I^{a,b,c}_{i,j,k}}$	Number of pre-emptions caused by a thread $\theta_{a,b,c}$ on a thread $\theta_{i,j,k}$
$\mathsf{p_{i,j,k}}$	Priority of a thread $\theta_{i,j,k}$
$\mathsf{Wr_{i,j,k}(t)}$	Remote link delay of a message $\mu_{i,j,k}$ at time t
$\mathsf{rbf_{i,j,k}(t)}$	Request bound function of a message $\mu_{i,j,k}$ at time t
$\mathsf{Wl_{i,j,k}(t)}$	Shared link delay of a message $\mu_{i,j,k}$ at time t
$\mathsf{PN}_{\mu_{\mathsf{i,j,k}}}$	Set of switches that a message $\mu_{i,j,k}$ traverses during a D-fork or D-Join
$\mathsf{sbf_{i,j,k}(t)}$	Supply bound function of a message $\mu_{i,j,k}$ at time t
$\mathsf{SD_{i,j,k}}$	Switching delay of a message $\mu_{i,j,k}$
$\mathsf{SFD_{i,j,k}}$	Store-and-forward delay of a message $\mu_{i,j,k}$
$\mathsf{t_{i,j,k}}$	Time associated to a message $\mu_{i,j,k}$
$\mathsf{r_{DP_{i,j,k}}}$	WCRT of a distributed execution path $DP_{i,j,k}$
$\mathsf{r^{msg}_{i,j,k}}$	WCRT of a message $\mu_{i,j,k}$
$\mathsf{r_{i,j,k}}$	WCRT of a thread $\theta_{i,j,k}$

Chapter 1

Introduction

1.1 Research Motivation

1.1.1 Background

Real-time systems are defined as "*those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced*" (Stankovic, 1988). Therefore, the aim of real-time systems is to execute applications in a way that those applications met their temporal constraints, named deadlines.

Real-time embedded systems are part of our everyday life. These systems range from the traditional areas of military and mission critical to domestic and entertainment applications. Real-time applications span a wide range of domains, such as industrial automation, automotive applications, flight control systems, multimedia applications, telecommunications and space missions. Real-time applications are mainly classified according to the consequences that a failure in the system can cause, as: *hard real-time* applications and *soft real-time* applications. In a hard real-time system the missing of a single deadline leads to catastrophic consequences, meanwhile in soft real-time systems it implies the degradation of the service provided by the system but without jeopardizing its correct behaviour.

Real-time applications are commonly implemented using high level programming languages (e.g., C\C++, Java and Ada) and sometimes implemented with lower level languages and functionalities such as assembly code, timers, and low-level drivers for manipulating tasks and interrupts. Unfortunately, the analysis of those programs is a complex task and in order to simplify the analysis of such programs, a higher level of abstraction is used to verify the properties of the system. In the time-critical domain, the timing behaviour of the software parts composing a real-time systems are modelled as *real-time tasks*.

The main aspects to consider when designing a real-time system are:

i. the characteristics related to the tasks in the system such as if they execute *sequentially* or in *parallel*, their *Worst-Case Execution Time* (WCET), their *periodicity*, their *deadlines*, their *dependencies*, etc. ; and,

ii. the processor platforms and their respective processing capabilities.

During the first decades of real-time research, it mainly focused on issues related to *sequential real-time models* executed on uni-processor platforms. Scheduling real-time tasks in uni-processor systems is considered a well-understood area. In uni-processor systems, for a tasks to respect its time constraints, its WCRT must be less than or equal to its deadline D_i, where the WCRT of a task τ_i, is the maximum time that a job of a task requires for completing execution. Nowadays, it is considered that uni-processor scheduling is a mature area (some interesting results on uni-processor scheduling are summarised in (Sha et al., 2004)).

Scheduling on multiprocessor systems has been on the interest of real-time researchers for several years. But special attention to multiprocessors was paid with the advent of commercial platforms from the main multiprocessor vendors in the first decade of 2000. Multiprocessor platforms were introduced as a solution for the physical limitations of the traditional approach of increasing the processor clock speed for obtaining more processing power. These limitations are mainly related to problems with high power consumption and heat dissipation. An interesting survey on relevant techniques for multi-core (a.k.a. multi-processor) executing sequential real-time tasks is presented in (Davis and Burns, 2011).

Due to advent of multiprocessor architectures, parallel programs became a highly relevant tool for exploiting the multiprocessor platform resources. Some of the more relevant programming languages are the ones based on C and C++. For example OpenMP (OpenMP-Arch-Rev-Board, 2012) and MPI (MPI-Forum, 2012), for shared memory and distributed memory, respectively. Parallel execution models are widely used in areas such as High Performance Computing (HPC) for several decades and utilised on several computational demanding applications. Parallel processing can increase the performance of applications by executing them on multiple processors at the same time.

Most of the results on multiprocessor scheduling are based on the sequential execution model, in which intra-task parallelism is forbidden. Whenever task parallelism is forbidden, sub-tasks (i.e. threads) belonging to a task must execute only in one processor at a time. On the other hand, task models in which the execution of several sub-tasks (threads) of the same task are allowed to execute in different processors at the time are called *multi-threaded parallel task models*. In the multi-threaded parallel real-time models each task τ_i starts execution as a sequential thread followed by the execution of a set of parallel threads; this sequential sections and parallel sections are called *sequential segments* and *parallel segments*, respectively. These segments are commonly alternated between sequential and parallel segments.

Regardless uni-processor scheduling is considered a well-understood area, the problem of scheduling sequential real-time tasks in distributed systems composed by a set of uni-processor nodes is still an open problem. Similarly, the case of scheduling sequential real-time tasks in distributed systems composed by a set of multi-processor nodes is also an open problem.

Furthermore, despite the fact that parallel computations in multi-processors can offer an increased processing capacity, the *multi-threaded parallel real-time task* models have not considered the cases in which a *distributed execution* also exists. Such execution pattern can be used to provide even more capabilities and processing power. Even more, in some applications the use of parallel and distributed computations is the only possibility in which the applications can comply with their timing constraints. An example of such type of applications are modern automotive applications, which need to execute computational intensive applications (e.g., infotainment or driver assistance applications).

A distributed real-time system is the one in which the system objectives involve real-time activities which must be carried on with some specific time bounds. In a distributed real-time system, applications are required to interact with the environment under control, by reading sensors and consequently responding accordingly through actuators. In fact, in a distributed real-time system, both the processing and communication phase need to be guaranteed to occur on time.

Real-time network scheduling considers the time needed to transmit messages through a network and guarantee that they will be transmitted within a certain time bound - this time is referred as the end-to-end delay. According to (Tindell et al., 1995), the end-to-end delay is defined as the time that takes to transmit a message between a task that generates it and the task that receives it. Thus, a key objective in real-time network scheduling it to be able to accurately bound such end-to-end delays. But, computing such a delay is not an easy task, because it heavily depends on the underlying network technology to be used. This is because the mechanisms to arbitrate the access for transmission between sending tasks may vary depending on the type of transmission network.

The integration of distributed real-time applications requires that the schedulability analysis is extended to consider the computations in the nodes and the transmission delay in the network. A well accepted technique for verifying the time correctness of real-time distributed applications is the *holistic analysis*. The main objective of the holistic analysis is to bound the *end-to-end* delay. The key aspect on holistic analysis is the possibility of accurately computing the WCRTs of tasks and messages. Furthermore, by using a holistic approach it may be possible to achieve higher system resource utilization due to the consideration of the system as a whole (see Chapter 6).

1.1.2 Problem statement

Given the increased availability of multiprocessor platforms and their natural applicability towards parallel execution models, real-time researchers have started to pay attention to the case of *multi-threaded parallel real-time task models*. The use of parallel models can reduce the time required for processing computational intensive applications, and it is currently the general trend to increase processing in many areas requiring high computing power, and real-time systems are not an exception (e.g., (Lakshmanan et al., 2010; Saifullah et al., 2011; Fauberteau et al., 2011; Qamhieh et al., 2011; Saifullah et al., 2013), etc.). In these models, concurrent real-time activities are allowed to become parallel and the processing of a single task may occur in more than one processor at the same time. By using parallel computations, the time required for processing computational-intensive applications can be reduced, thereby allowing them to comply with more stringent deadlines. Commonly, parallel applications are based on the fork-join execution model. Such kind of applications start by executing sequentially and then forks to be executed in parallel, when the parallel execution has completed, the results are aggregated by performing a join operation.

However, despite the fact that parallel computations can offer an increased processing capacity, multi-threaded parallel real-time task models have not considered the cases in which a distributed execution also exists and which can be used to provide even more capabilities and processing power. Furthermore, in some applications the use of parallel computations is the only possibility in which the applications can comply with their time constraints. Figure 1.1 shows an example of a parallel distributed task τ_1 which is composed of 4 threads, two are executed on the *local* node (i.e., *local execution*) and the other two ones are executed on a *remote* node (i.e., *remote execution*). In Figure 1.1, the execution of τ_1 starts with the thread θ_1, θ_1 is split into 4 threads θ_1, θ_2, θ_3 and θ_4. Threads θ_1 and θ_2, execute locally and threads θ_3 and θ_4 are executed in a remote node. In Figure 1.1 it is also possible to see the messages for sending and receiving data. In this dissertation it is assumed that the code is already available and being executed on distributed nodes. This a valid assumption since there exist frameworks that allow the mobility of code/data meanwhile providing Quality-of-Service guarantees (Goncalves et al., 2010).

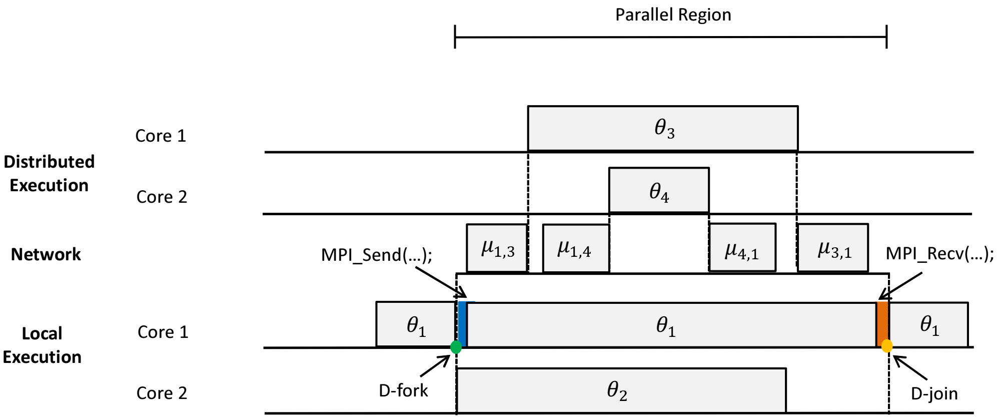

Figure 1.1: Execution of a task following the parallel and distributed model.

An example in which this kind of computing model could offer large benefits are modern cars (Lim et al., 2011). Modern cars are a good example of time-constrained distributed systems, since they are composed of tens of computing nodes interconnected by various types of communication networks. Figure 1.2 shows an example of such a system, which provides an ideal platform for the execution of computational intensive applications such as infotainment or a driver assistance applications. It is also possible to observe that Figure 1.2 shows the execution of the task τ_1 and its respective threads θ_{1-4} shown in the example of Figure 1.1. It is assumed that the Electronic Control Units (ECUs) have processors (a.k.a. cores). The local execution is carried on the Head-Unit (HU) ECU and the remote execution is being carried on the Rear Seat Entertainment (RSE) ECU. Thus, threads θ_3 and θ_4 have to cross switches SW-1 and SW-2 during the Distributed-fork (D-fork) operation and they cross SW-2 and SW-1 in the Distributed-join (D-join) operation. Furthermore, other type of applications such as avionic applications, industrial environments, smart city applications, etc., can take advantage of such execution patterns.

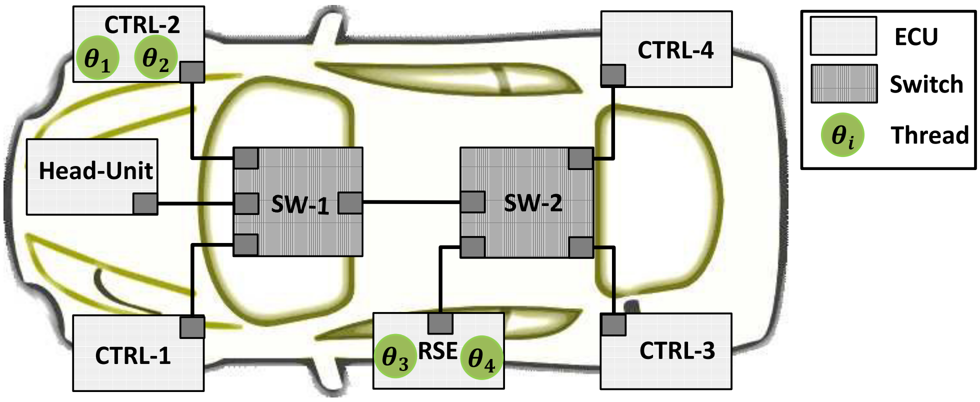

Figure 1.2: Parallel and distributed automotive application example.

Consequently, frameworks are thus required that allow to manage the resources of the system globally, enabling high demanding and time-constrained software to be distributed cooperatively by both parallel and distributed processors. Such scenarios require the integration of distributed computations with parallel real-time models. The fork-join Parallel/Distributed real-time (P/D task) model, is designed to consider such execution pattern.

The integration of distributed computations within real-time applications requires that scheduling algorithms consider the messages interchanged on the network. In a distributed system, the transmission delay of messages cannot be considered negligible as in the case of multi-core systems, therefore, their impact on the schedulability of the system has to be considered.

Therefore, existing real-time analysis tools must also be extended to consider both the processing and the network overhead. The communication infrastructure to interchange information needs to be analysed in detail, therefore real-time network scheduling plays an important role in the design of such distributed architectures (e.g., (Davis et al., 2007; Bauer et al., 2010; Kopetz et al., 2005; Pedreiras et al., 2005)). On the other hand, the combined analysis of network and the processing elements has to be considered (e.g., (Tindell and Clark, 1994; Spuri, 1996; Palencia and Harbour, 2003)).

1.2 Goal and Objectives

Based on the motivation presented in Section 1.1, the main goal of this dissertation is to provide a design framework that allows for the integration of parallel distributed models in real-time embedded applications.

This goal can be divided into three objectives:

i. Provide a framework for designing parallel and distributed real-time embedded systems that considers an adequate execution model;

ii. Finding efficient and predictable methods for the allocation of parallel and distributed applications (P/D tasks) onto the processing and communication resources of the underlying computing platform;

iii. Provide analytical tools that verify the correctness of the proposed methods.

In relation to (i), several programming models have addressed the problem of programmability for parallel and distributed systems, in both shared memory and distributed memory (see Section 2.6). But none of these approaches integrates distribution and parallelism in a transparent and efficient way.

Therefore, the objective of providing an adequate execution framework can be stated as:

*Propose an **execution model** for **parallel** and **distributed real-time embedded** platforms and introduce an **easy to use programming framework** for the development of such systems.*

In relation to (ii), real-time scheduling theory has extensively provided tools and methods for addressing the problem of scheduling task sets within the traditional real-time task models for both uni-processor (Sha et al., 2004) and multi-core platforms (Davis and Burns, 2011). Recently these tools and methods have been extended to consider multi-threaded parallel task models on multi-core platforms. In parallel, research related to real-time network scheduling has been discovering methods to efficiently schedule messages on different network architectures. However, there exist a gap for the convergence of these two important domains.

Therefore, the scheduling objective can be stated as:

*Propose a set of **scheduling algorithms and heuristics** for the allocation of parallel and distributed applications (P/D tasks) onto a distributed system composed of **identical embedded nodes**.*

In relation to (iii), the *holistic response time analysis* theory has considered traditional real-time models, but it has not explored multi-threaded parallel task models executing in distributed platforms.

Therefore, the holistic analysis objective can be stated as:

*Propose a **holistic response time analysis** technique that validates the allocations produced by the scheduling algorithms and heuristics proposed in (ii).*

As noted, current parallel models being addressed in real-time embedded systems do not address distributed computations. Nevertheless, it is clear that parallelism and distribution are two dimensions of existent systems, which are more and more simultaneously found.

By correctly addressing the problems stated in Section 1.2, it will be possible to develop reliable and predictable parallel and distributed real-time applications.

1.3 Structure of the Dissertation

This chapter introduced the research context of this dissertation. The dissertation is focused on the development of parallel and distributed real-time embedded systems. In this context, this dissertation presents a set of design tools that consider both the execution tasks (or threads) on distributed processors and the transmission of messages on a real-time network.

Chapter 2 introduces the background and a brief survey of the relevant works on topics addressed in this dissertation. Some concepts on real-time models and real-time scheduling on centralised, and distributed real-time systems are presented in Section 2.2 and Section 2.3, respectively. Section 2.4 studies works related to task partition and priority assignment in real-time distributed systems. Section 2.5 presents the related works on holistic analysis for real-time distributed systems. Section 2.6 surveys some parallel programming models.

Chapter 3 introduces the Fork-Join Parallel/Distributed Real-Time Task Model (P/D task model). Section 3.2 shows a case study of the possible implementation of the P/D task model with a combination of OpenMP and MPI programming models. A timing model for OpenMP/MPI programs is derived by individually studying the behavior of typical OpenMP and MPI programs (Garibay-Martínez et al., 2012). Section 3.4 presents the P/D task model which is derived from the observations presented in Section 3.2.

Chapter 4 presents the Partitioned/Distributed-Deadline Monotonic Scheduling (P/D-DMS) algorithm for P/D tasks (Garibay-Martínez et al., 2014b). The P/D-DMS algorithm is shown to have a resource augmentation bound of 4, which implies that any task set that is feasible on m unit-speed processors and a single shared bus real-time network, can be scheduled by this algorithm on m processors and a single shared real-time network that are 4 times faster. Section 4.2 presents the Distributed Stretch Transformation (DST) algorithm for P/D tasks, an algorithm that tries to keep as much as possible treads to be executed locally. The main objective of the DST is to reduce the number of messages in the network meanwhile complying with the task deadlines. The resource augmentation bound for the Partitioned-Distributed-DMS algorithm is explained in Section 4.3. The simulations that confirm the analytical results are provided in Section 4.4, and finally a summary of the chapter is given in Section 4.5.

Chapter 5 present the Distributed using Optimal Priority Assignment (DOPA) and the Parallel-DOPA (P-DOPA) heuristics, which partitions a set of sequential and P/D tasks, respectively, and find their priority by using the Optimal Priority Assignment Algorithm (OPA) (Audsley, 1991). Section 5.2 describes the DOPA heuristic for the linear transactional model. The P-DOPA heuristic for the P/D tasks model is described in Section

5.3.

Chapter 6 presents a holistic timing analysis for the computation of the Worst-Case Response Time (WCRT) for P/D tasks when transformed by the DST algorithm (Garibay-Martínez et al., 2014a, 2015b). Both synchronous and asynchronous communication patterns are considered. Section 6.2 briefly describes the Flexible Time Triggered - Switched Ethernet (FTT-SE) protocol and a technique for computing the WCRT of messages scheduled with the FFT-SE protocol (Ashjaei et al., 2013). Section 6.3 presents the proposed holistic analysis for synchronous and asynchronous systems. Section 6.4 shows how the WCRT computation presented in Section 6.3 can be improved by considering a pipeline effect that occurs on those systems.

Chapter 7 introduces a set of formulations for modelling the allocation of P/D tasks in a distributed multi-core (a.k.a. multi-processor) architectures by using a constraint programming approach (Garibay-Martínez et al., 2015). A constraint programming approach expresses the relations between variables in the form of constraints. The constraint programming formulation is guaranteed to find a feasible allocation, if one exists, in contrast to other approaches based on heuristic techniques. Section 7.1.1 introduces the system model. The system model presented in Section 7.1.1 differs from the one presented in Chapter 3 that the processing nodes are multi-core nodes.

Chapter 8 presents some simulations and an experimental evaluation of the concepts presented through the dissertation. Section 8.3 introduces a simulator for P/D tasks that considers an FTT-SE network (Oliveira et al., 2015). Section 8.5 presents the Parallel/Distributed Real-Time (PDRT) library. Finally, the results from the experimental evaluation are compared with the results obtained by simulation.

Finally, Chapter 9 presents the research contributions, conclusions and the future work.

Chapter 2

Background and Previous Relevant Work

2.1 Introduction

This chapter presents a brief survey of the works that are relevant for this dissertation and it reinforces the research context of the dissertation. This dissertation focuses on designing distributed applications in which the use of parallel computations is the only possibility in which the applications can comply with their time constraints.

This chapter starts by presenting a broad classification of real-time task models and real-time scheduling algorithms for different computing platforms (Section 2.2). In this dissertation those models are extended for considering parallel and distributed execution (Chapter 3). The consideration of a distributed architecture implies the coordination of different elements in the systems given their transmission patterns (e.g., time-triggered or event-triggered) and the use of a real-time network (Section 2.3). In order to execute real-time applications by respecting their deadlines, a correct allocation of tasks to processors and messages to the real-time networks must exists. Works related to task partition and priority assignment in real-time distributed systems are studied in Section 2.4. Works related to holistic analysis for real-time distributed systems are studied in Section 2.5. It is important to notice that the works related to holistic analysis and partitioning and priority assignment had only consider the sequential real-time task models (in this dissertation it is also considered the P/D task model). Section 2.6 surveys some parallel programming models. The studied parallel programming models are mainly C/C++ based (commonly used for the development of real-time embedded systems). Finally, a summary of the chapter is given in Section 2.7.

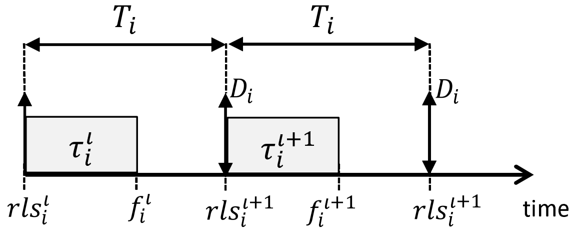

Figure 2.1: Jobs $\tau_i^\iota$of a task τ_i.

2.2 Real-Time Models and Scheduling Algorithms

Real-time applications are commonly implemented using C\C++, Java and Ada programming languages and sometimes with lower level languages and functionalities like assembly code, timers, and low-level drivers for manipulating tasks and interrupts. Unfortunately, the analysis of those programs is a complex task, and in order to simplify the analysis of such programs, higher levels of abstraction are used to verify the properties of the system. In the time-critical domain, the timing behaviour of the software parts composing a real-time systems are modelled as real-time tasks.

A real-time system is composed by a set τ of n tasks $(\tau_1, \ldots, \tau_n)$ (also known as applications). A task τ_i becomes *ready* to execute at a *release time* (denoted as rls_i) and continues execution until completion or finishing time (denoted as f_i). A task τ_i is mainly characterized by:

i. its Worst-Case Execution Time (WCET) (denoted as C_i);

ii. its minimum inter-arrival time or period, for the sporadic and the periodic task models, respectively:

   a. ***Periodic task model.*** Tasks arrive in a strict periodic fashion (every T_i time units). In the periodic task model, instances (i.e., jobs) of a task τ_i arrive in a strictly periodic manner (every T_i time units).

   Figure 2.1 shows an example of the jobs τ_i^ι of a task τ_i. Job τ_i^ι arrives at time rls_i^ι and completes its execution at time f_l^ι. A new job $\tau_i^{\iota+1}$ (task instance) arrives at time $rls_i^{\iota+1}$ and completes its execution at time $f_l^{\iota+1}$. Thus, different jobs refer to different activation times of the same task. For brevity, in this dissertation the super index ι is omitted since it is considered that only one job τ_i^ι can be active at every time instant. This is because in this dissertation only systems with deadlines smaller than the periods are considered.

   Figure 2.2a shows the periodic task model, the upside arrows denote the release (rls_i) of a task τ_i. It is possible to see that tasks arrive every T_i time units.

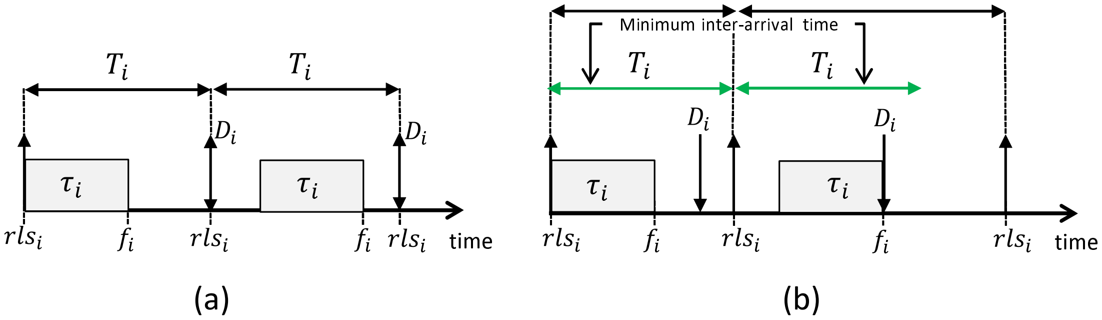

Figure 2.2: (a) Periodic task model with implicit deadlines and (b) sporadic task model with constrained deadlines.

In the periodic task model, task sets are classified depending on the time in which tasks are released as:

i. ***Synchronous.*** All tasks arrive simultaneously to the system;

ii. ***Asynchronous***. Tasks may not arrive simultaneously to the system, and may be separated by fixed time slots called *offsets*;

b. ***Sporadic task model.*** Tasks arrive to the system with a minimal inter-arrival time (denoted as T_i) between successive arrivals. I. e., the time elapsed between arrivals is at least T_i time units. In Figure 2.2b the elapsed time after the arrival of a task instance of task τ_i is larger than its minimum inter-arrival time;

The results in this dissertation apply to both the periodic and sporadic task model.

iii. its relative deadline (denoted as D_i), there are three categories related to the constraints imposed by deadlines D_i:

a. ***Implicit deadlines***. All tasks have deadlines equal to their periods (i.e. $D_i = T_i$). This can be observed in Figure 2.2a. Downside arrows denote the deadline of such task. It can be seen that the deadlines D_i coincide with the release rls_i of a taks τ_i;

b. ***Constrained deadlines***. All tasks have deadlines less than or equal to their periods (i.e. $D_i \leq T_i$). It can be observed in Figure 2.2b that the deadline D_i of a task is placed before the minimum inter-arrival time;

c. ***Arbitrary deadlines***. Task deadlines are arbitrary; less than, equal to or greater to their periods.

This dissertation focuses on implicit and constrained deadlines (i.e., $C_i \leq T_i$).

The response time r_{τ_i} of a task τ_i is defined as the difference between its finishing time and its release time $r_{\tau_i} \overset{\text{def}}{=} f_i - rls_i$. For a task set τ to be schedulable, it has to be guaranteed that for all their tasks τ_i, the response time r_{τ_i} of all their jobs must be less than or equal to its deadline D_i.

Tasks are also classified according to their possible precedence constraints as:

i. ***Dependent task sets.*** A task τ_j is called dependent on task τ_i, if a task τ_i has to finish execution before a task τ_j can start execution. It is said that task τ_j, has a precedence constraint and depends on τ_i. A precedence constraint is denoted by $\prec$. For example $\tau_i \prec \tau_j$ reads as τ_i precedes τ_j;

ii. ***Independent task sets.*** Otherwise they are called independent.

This dissertation focuses on dependent task sets.

This dissertation focuses on dependent task sets, in a distributed system, tasks are communicated through messages, such messages imply a dependency between tasks. Therefore, a task that is activated by the arrival of a message is *dependent* on such arrival. More details are presented in Section 2.3.

The *utilization* u_i of a task τ_i is given by $u_i \overset{\text{def}}{=} \frac{C_i}{T_i}$. The total utilisation of a task set is given by: $u_{sum} \overset{\text{def}}{=} \sum_{i=1}^{n} \frac{C_i}{T_i}$. The *density* δ_i of a task τ_i is given by $\delta_i \overset{\text{def}}{=} \frac{C_i}{D_i}$. The total density of a task set is given by: $\delta_{sum} \overset{\text{def}}{=} \sum_{i=1}^{n} \frac{C_i}{D_i}$.

2.2.1 Classification of Real-Time Processing Platforms

From the scheduling perspective, processor platforms are classified in relation to the processing capabilities of the processors composing such platform. Those platforms are classified in three main categories according to their processing capabilities (Carpenter et al., 2004):

i. ***Identical.*** Processors composing the platform are identical; this is, the speed rate of the execution of all tasks in the system is the same on all processors;

ii. ***Uniform.*** Processors composing such a multiprocessor system differ only on the speed rate they execute tasks; hence a processor with speed rate of 2, executes all tasks in the system twice as fast as when compared to a processor with a speed rate of 1;

iii. ***Unrelated.*** Processors composing the multiprocessor systems are different; hence, the speed rate of execution of tasks depends on both processors and tasks. For example, suppose two different types of tasks a and b, and two different types of processors x and y. It may be possible that task a executes with higher speed rate on processor x than in processor y; conversely, task b may execute with higher speed rate in y than in x. In fact, not all tasks on the system may be able to execute on all processors.

This dissertation mainly focuses on identical platforms. Although, a combination of nodes with different processing capabilities is considered in Chapter 5 and Chapter 7.

The processing capabilities and number of processors that compose a computing platform, dictate the type of scheduling algorithms to be used. In the following some of those algorithms are described.

2.2.2 Real-Time Uni-processor Scheduling

Uni-processor real-time scheduling is considered a mature research area (some interesting results on uni-processor scheduling are summarised in (Sha et al., 2004)).

Uni-processor algorithms, in particular, and real-time scheduling in general can be divided in three categories (Carpenter et al., 2004):

i. ***Fixed-task priority***. Each task and its jobs have a fixed priority assigned at the beginning of the execution, and is kept unchanged until the end of the execution. Examples of this type of algorithms are: Rate Monotonic (RM) (Liu and Layland, 1973), Deadline Monotonic (DM) (Audsley et al., 1991), Optimal Priority Assignment (OPA) (Audsley, 1991), etc.;

ii. ***Fixed-job priority***. Each task may assign different priorities to its jobs, but each job with an assigned priority keeps that priority until the end of the job execution. Some examples of this type of algorithms are the ones based on the Earliest Deadline First (EDF) scheduling (Ramamritham et al., 1990);

iii. ***Dynamic priority***. Task jobs may have different priorities at any point in time which depends on the current scheduling conditions. Some examples of this type of algorithms are the ones based on the Least Laxity First (LLF) scheduling (Dertouzos and Mok, 1989).

This dissertation focuses on fixed-priorities.

Scheduling algorithms are also classified according to its pre-emption levels as:

i. ***Non-pre-emptive.*** Tasks (or task jobs) that have started execution cannot be pre-empted, and therefore, they must execute until completion;

ii. ***Pre-emptive.*** Tasks (or task jobs) can be pre-empted by another task (tasks job) with higher priority at any point in time;

iii. ***Cooperative.*** Tasks (or task jobs) can only be pre-empted on certain predefined scheduling points. This means that there are a series of pre-emptive sections where pre-emptions take place and non-pre-emptive sections where pre-emptions are forbidden.

This dissertation considers the transmission of messages over the network as non-preemptive and the in node execution of tasks as preemtive.

In uni-processor systems, the Worst-Case Response Time (WCRT) (denoted as r_{τ_i}) of a task τ_i, is the maximum time that a job of a task requires for completing execution (including possible interference and/or blocking times). For a tasks to respect its time constraints, its response time must be less than or equal to its deadline D_i. Two results for the computation of the WCRT are of special importance for this dissertation: *preemptive uni-processor systems* and *non-preemptive uni-processor systems*.

The results for preemptive uni-processor scheduling were introduced by (Joseph and Pandya, 1986). They provided the following recursive equation that can be used to calculate the response time of a task τ_i:

$$r_{\tau_i}^{n+1} = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}^{n}}{T_j} \right\rceil C_j, \tag{2.1}$$

where r_{τ_i} is the WCRT of a task τ_i and $hp(r_{\tau_i})$ is the set of all tasks τ_j with higher priority than τ_i that execute on the same processor.

The recursion ends when $r_{\tau_i}^{n+1} = r_{\tau_i}^{n} = r_{\tau_i}$, and can be solved by successive iterations starting from $r_{\tau_i}^{0} = C_i$. The series is non-decreasing, and therefore converges if $\sum_{\tau_j \in hp(\tau_i) \cup \tau_i} \frac{C_j}{T_j} \leq 1$. If the condition of convergence is not respected, task θ_i is not schedulable.

Eq. (2.1) also applies to fixed-priority networks by considering the non-premtability of messages which provokes a *blocking time*. For example, suppose a message μ_l of lower priority being transmitted over a real-time network. If a message of higher priority τ_h desires to transmit over the network, message μ_h has to wait for the lower priority message μ_l to complete its execution. This is the so called blocking time.

The calculation of the WCRT for messages μ_i needs to consider the non-preemptability of messages. For fixed-priority non-preemptive messages, the following recursive equation can be used to calculate its WCRT (George et al., 1996):

$$r_{\mu_i}^{n+1} = C_i + \sum_{\mu_j \in hp(\mu_i)} \left\lceil \frac{r_{\mu_i}^{n}}{T_j} \right\rceil C_j + \max_{\mu_j \in lp(\mu_i)} \{\mu_j\}, \tag{2.2}$$

where, the third term on the right hand side of Eq. (2.2), accounts for the maximum possible blocking time of a higher priority messages μ_i, caused by lower priority message μ_j, contained in the set of lower priority messages $lp(\mu_j)$.

2.2.3 Real-Time Multi-processor Scheduling

Scheduling on multiprocessor systems has been on the interest of real-time researchers for several years. But special attention to multiprocessors was paid with the advent of commercial platforms from the main multiprocessor vendors in the first decade of 2000. Multiprocessor platforms were introduced as a solution for the physical limitations of the traditional approach of increasing the processor clock speed for obtaining more processing power. These limitations are mainly related to problems with high power consumption and heat dissipation.

The aim of multiprocessor scheduling algorithms is to guarantee that the tasks belonging to an application meet their deadlines when executed in a multiprocessor platform. According to (Carpenter et al., 2004), there are two main aspects to consider when scheduling real-time tasks on multiprocessors:

i. ***The priority assignment problem.*** The priority assignment problem is related of assigning priorities to tasks in order to give real-time guarantees;

ii. ***The allocation problem.*** The allocation problem tries to find the best allocation of tasks to processors.

In fact, in (Carpenter et al., 2004), multiprocessor algorithms are classified according to the level of priority changes they allow to perform over the task set, and on the way allocations to processors are done (i.e. by allowing or restricting migration). Therefore, the main classification of scheduling algorithms for multiprocessors is referred as *priority-based* and *migration-based* algorithms. For the case of distributed systems, a similar classification can be considered.

The priority-based classification is the same as the one presented in Section 2.2.2. Related to Migration-based algorithms, they are classified in three categories:

i. ***Non-migration.*** Whenever a task is allocated to a processor, all task jobs are kept on that processor without possibility of migration to another processor;

ii. ***Task-level migration.*** Task jobs can be executed on different processors, but each job already assigned to a processor can only be executed on that processor;

iii. ***Job-level migration.*** Task jobs can migrate and execute on different processors without restrictions.

Based on the level of allowed migrations, real-time researchers commonly divide scheduling algorithms in two groups; *partitioned* and *global*. If no migration of tasks into other processors is allowed, the scheduling algorithms are referred as partitioned. Otherwise they are referred as global.

This dissertation focuses on non-migration or static allocation systems.

Scheduling algorithms are also classified according to the way they process workloads as:

i. ***Work-conserving.*** An algorithm is said to be work-conserving if it does not allow a processor in the system to be idle (i.e. without executing a job) if there exist a task ready to execute;

ii. ***Non-work-conserving.*** Otherwise, is referred as non-work conserving.

This dissertation focuses on work-conserving algorithms.

2.2.4 Real-Time Multi-threaded Parallel Task Models for Multiprocessor Systems

Most of the results on multiprocessor scheduling are based on the sequential execution model, in which intra-task parallelism is forbidden (Davis and Burns, 2011). Whenever task parallelism is forbidden, sub-tasks (i.e. threads) belonging to a task must execute only in one processor at a time. On the other hand, task models in which the execution of several sub-tasks (threads) of the same task are allowed to execute in different processors at the time are called multi-threaded parallel task models.

Recently real-time researchers have considered task models based on multi-threaded parallel task models. In this section, we present some of the works on multi-threaded parallel task models. We focus our attention to the case of fixed-priority systems.

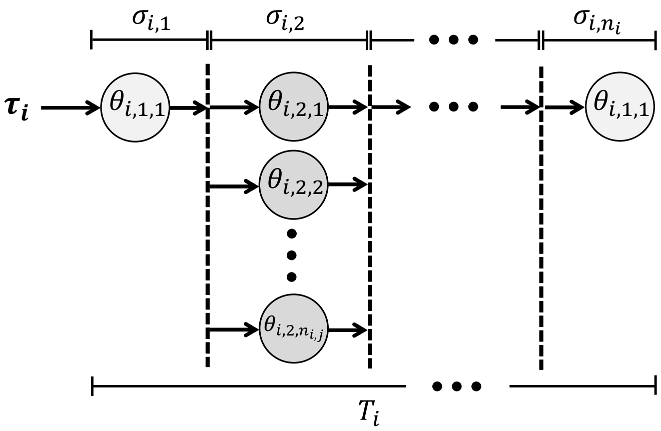

Figure 2.3: Multi-threaded parallel real-time task.

Figure 2.3 illustrates a multi-threaded parallel real-time task. In the multi-threaded parallel real-time models each task τ_i is composed of a set of n_i segments (denoted as $\sigma_{i,j}$). Each task starts execution as a sequential thread followed by the execution of set of parallel threads; this sequential sections and parallel sections are called *sequential segments* and *parallel segments*, respectively. These segments are commonly alternated between sequential and parallel segments.

This dissertation focus on multi-threaded parallel fixed-priority real-time tasks, in particular to fork-join task model.

Research related to multi-threaded parallel fixed-priority real-time tasks has targeted mostly multi-core architectures. For example, in (Lakshmanan et al., 2010), the authors introduced the Task Stretch Transformation (TST) model for fork-join parallel synchronous tasks. The TST considers fork-join preemptive fixed-priority periodic tasks with implicit deadlines. The fork-join structure is transformed into a sequential structure and the set of sequential fixed-priority tasks remaining after the transformation is partitioned according to the Fisher-Baruah-Baker First-Fit-Decreasing (FBB-FFD) (Fisher et al., 2006) partitioning algorithm. The authors proved that the TST has a resource augmentation bound of 3.42. That implies that any task set τ that is feasible on m unit-speed processors, can be scheduled by this algorithm on m processors that are 3.42 times faster.

The Segment Stretch Transformation (SST) model was proposed by (Fauberteau et al., 2011). The authors also transform the fork-join structure of a task into sequential one by creating a master thread, but with the difference (when compared to (Lakshmanan et al., 2010)) that no thread is ever allowed to migrate between cores. They showed through simulations that the TST and SST algorithms obtain similar results, and that none of them dominates the other. Later, (Qamhieh et al., 2011) proved that the SST has the same resource augmentation bound than TST, i.e., 3.42.

A generalization of this problem was introduced in (Saifullah et al., 2013). Two main extensions to previous works (Lakshmanan et al., 2010; Fauberteau et al., 2011; Qamhieh et al., 2011) were made. First, the limitation of having the same number of threads in all parallel segments within a task was lifted by allowing an arbitrary number of threads to be executed on each parallel segment. And second, the authors considered the analysis of DM and EDF scheduling. They provided a resource augmentation bound of 4 and 5, when global EDF and partitioned DM are used to schedule tasks, respectively.

An effort towards the integration of the parallel real-time task models and distributed systems is presented in this dissertation. Contrarily to multi-core systems (Lakshmanan et al., 2010; Fauberteau et al., 2011; Saifullah et al., 2011), the transmission delay of messages sent between nodes of the distributed system have to be considered and cannot be deemed negligible.

This dissertation focuses on multi-threaded parallel models, specifically in the fork-join model.

However, this dissertation focuses on multi-threaded parallel distributed applications consideration of a distributed architecture implies the coordination of different elements in the systems given their transmission patterns (e.g., time-triggered or event-triggered) and the use of a real-time network, therefore Section 2.3 discusses those issues. In order to execute real-time applications by respecting their deadlines, a correct allocation of task to processors and messages to the real-time networks must exists.

2.3 Distributed Real-Time Systems

(Tanenbaum, 1995) defined a distributed system as *"a collection of independent computers that appears to the users of the system as a single computer"*. Distributed systems are composed by a set of processing units cooperating to achieve a common objective. In order to achieve their goal, distributed systems exchange information in the form of messages which are sent through an interconnection network.

A distributed real-time system is the one in which the system objectives involve real-time activities which must be carried on with some specific time bounds. Thus, the main difference of distributed real-time systems and traditional distributed systems is the way the time-constrained activities are performed.

In a distributed real-time system, applications are required to interact with the environment under control, by reading sensors and consequently responding accordingly through actuators. In fact, in a distributed real-time system, both the processing and communication phase need to be guaranteed to occur on time.

Real-time network scheduling considers the time needed to transmit messages through a network and guarantee that they will be transmitted within a certain time bound - this time is referred as the end-to-end delay. According to (Tindell et al., 1995), the end-to-end delay is defined as the time that takes to transmit a message between a task that generates it and the task that receives it. Thus, a key objective in real-time network scheduling it to be able to accurately bound such end-to-end delays.

But, computing such a delay is not an easy task, because it heavily depends on the underlying network technology to be used. This is because the mechanisms to arbitrate the access for transmission between sending tasks may vary depending on the type of transmission network.

Real-time network researchers had debated for several years about the better approach for handling message transmissions with real-time constraints. In (Kopetz, 1997), methods for handling real-time messages are divided in three main approaches:

i. ***Event-triggered.*** Messages are generated in a sporadic fashion and the generation of messages depends on the events generated at the sender side; therefore, it is difficult to guarantee a correct time behaviour at the receiver side. To solve such a problem, event-triggered systems implement acknowledgement mechanisms in order to control the sending rate of messages. Event-triggered systems present high flexibility and are preferred for the development of systems in which the messages rate is highly variable;

ii. ***Rate-constrained.*** Messages are generated in a sporadic fashion; however there exist an agreement between the sender and the receiver of generating messages without exceeding certain message rate. This approach allows the calculation of bounded times in which message transmissions can occur. The Rate-constrained approach is considered as a middle term when compared to the event-triggered approach and the time-triggered approach;

iii. ***Time-triggered.*** Messages are generated based on an a priori agreement between sender and receiver on the exact time in which messages are transmitted. This approach presents high degree of predictability when compared to other approaches, but it lacks flexibility when handling sporadic task.

This dissertation focuses on event-triggered and time-triggered systems.

Ethernet is so far the most widely used intercommunication protocol in general purpose networks. But audio and video streaming applications requiring real-time guarantees are already laying upon Ethernet as a non-costly solution for such real-time applications. Furthermore, it is currently in use on real-time systems applications, in some cases replacing other real-time architectures due to practical and monetary reasons. In (Loeser and Haertig, 2004) the foundations of real-time scheduling for switched Ethernet are presented. Such approach presents a viable way to implement real-time communications based on traffic shaping techniques that are commonly available over switched Ethernet.

Another widespread protocol is the CAN bus protocol, and it is probably the most widely used protocol in more stringent event-triggered environments (e.g. automotive applications). A response time analysis has been proposed in (Tindell and Burns, 1994) and later revised by (Davis et al., 2007) which introduced a revisited schedulability analysis that proves incorrect previous established bounds for CAN networks, even more, that work developed an optimal technique for the schedulability analysis of those architectures.

The Avionics Full-Duplex (AFDX) protocol is within the class of rate-constrained approaches. Such a protocol is one of the most successful standards for commercial safety-critical application. The 802.3 Ethernet standard utilises a technique of dedicated (reserved) bandwidth to guarantee Quality-of-Service (QoS) for the transmission of messages with real-time constraints. The AFDX protocol is a successful standard for avionic applications, its analysis is based on network calculus. However, recently in (Bauer et al., 2010), a method based on a combination of networks calculus and the trajectory method has outperformed previous calculated bounds.

Perhaps the more predictable technologies are the ones based on time-triggered protocol (Kopetz, 1997), in which a predefined set of time slots are used for message transmission. The most prominent implementation derived from the time-triggered protocol is the time-triggered Ethernet (Kopetz et al., 2005) which offers as its main feature high predictability and accurate response time bounds. However, protocols belonging to the time-triggered approach are also characterised to be rigid when dealing with sporadic messages.

To overcome such limitations, the authors of (Pedreiras et al., 2005) introduced the Flexible Transmission Triggered (FTT) Ethernet protocol. The FTT approach is based on a master-slave mechanism that coordinates the cohabitation of both types of real-time messages; the ones based on event-triggered and the ones based on time-triggered approaches, thus providing higher flexibility to a broader type of applications.

Related to Flexible Time Triggered Switched Ethernet (FTT-SE) networks, (Pedreiras et al., 2005; Marau et al., 2006) introduced the foundations for the FTT-SE protocol which

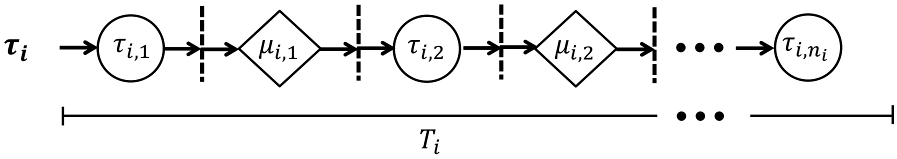

Figure 2.4: Linear transactional model for distributed systems.

adds time determinism to common switched Ethernet switches. In (Marau et al., 2012) the authors considered the computation of the WCRT for the single-master architecture. Further developments of the FTT-SE paradigm have been proposed by the authors of (Ashjaei et al., 2013) presented the multi-master architecture of the FTT-SE protocol and presented a technique to compute the WCRT in such a network. In (Ashjaei et al., 2014), an FTT-SE protocol is presented, in which the functionality of a master node is embedded on a specialized switch; the Hard Real-Time Ethernet Switching architecture (HaRTES) Ethernet switch.

2.4 Task Partition and Priority Assignment in Real-Time Distributed Systems

The problem of allocating real-time task in distributed systems is usually divided in two sub-problems:

i. finding the partitioning of tasks and messages onto the elements of the distributed system (processors and networks, respectively);

ii. finding the priority assignment for that partition.

In this section we present works related to the allocation of the linear transactional model for distributed systems (or simply sequential transactional model). Figure 2.4 show the linear transactional model. It is possible to observe a that a task $\tau_{i,j}$ is followed by a message $\mu_{i,j}$, the reason is because messages $\mu_{i,j}$ are sent over the network to communicate with task $\tau_{i,j}$ in a transaction τ_i.

In (Tindell et al., 1992) the allocation of tasks and messages is addressed as an optimization problem, solving it with the general purpose Simulated Annealing algorithm. The Simulated Annealing algorithm is used for iterating in a random manner over a given allocation, and perform an evaluation based on an “energy function” that measures the quality of the encountered solution (allocation). Tindell used the Deadline Monotonic (DM) scheduling algorithm (Leung and Whitehead, 1982) to assign priorities to tasks.

In (García and Harbour, 1995), the authors proposed an optimization technique that assumes a set of tasks and messages that are statically allocated to processors and networks (therefore, no partitioning phase is considered). Therefore, the authors focused their attention on the problem of assigning the priorities to the allocated tasks and messages. Their method is based on imposing artificial intermediate deadlines to the tasks and messages and assigning priorities to tasks by using DM.

(Richard et al., 2003) proposed a solution based on branch-and-bound; enumerating the possible paths that can lead to an allocation, and cutting the path whenever a feasible schedule cannot be reached by following such a task assignment. Again, DM is used to assign the priorities assuming that each task is defined by its own deadline and period. The bounding step is performed by checking the schedulability of each branch, based on the schedulability analysis derived by (Tindell and Clark, 1994).

In (Metzner and Herde, 2006), the authors model the task partitioning problem as an optimisation problem. However, this work assumes that each task has its own period and deadline, and it uses DM to assign priorities.

(Zheng et al., 2007), studied the case of automotive applications. The approach is based on finding the priorities for tasks and messages, in a way that no end-to-end deadline is missed. They proposed to solve the problem of priority assignment of tasks and messages by modelling it as an optimization problem. In (Zhu et al., 2010) is presented a similar problem as in (Zheng et al., 2007), but for a more detailed system model. The authors presented a sensibility analysis that is able to measure how much the execution time of tasks can be increased without missing its end-to-end deadlines. Their method is based on a combination of mixed integer linear programming for task allocation, which is optimized according to tasks utilization and deadlines. As a second stage of their method, they apply a set of heuristic steps for priority assignment of tasks and messages.

(Azketa et al., 2011) addressed this problem by using the general purpose genetic algorithms. They use a genetic algorithm with a permutational solution encoding. They initiate their genetic algorithm by assigning priorities using the HOPA heuristic (García and Harbour, 1995) which is based on DM priority assignment and iterate over different solutions by applying crossover, mutation and clustering operations. To test schedulability they use the holistic analysis presented in (Tindell and Clark, 1994) and (Palencia and Gonzalez Harbour, 1998, 1999).

None of the previous works had addressed the allocation of multi-threaded parallel tasks onto elements of a distributed system. This dissertation presents some methods for the allocation of such parallel task in distributed systems (see Chapter 4, Chapter 5 and Chapter 7).

2.5 Holistic Analysis for Real-Time Distributed Systems

The integration of distributed real-time applications requires that the schedulability analysis is extended to consider the computations in the nodes and the transmission delay in the network. A well accepted technique for verifying the time correctness of real-time distributed applications is the *holistic analysis*. The holistic analysis approach is based on the concept of *attribute inheritance*, in which the messages sent by a task inherit its temporal attributes (i.e., the period and the release jitter). The key aspect on holistic analysis is the possibility of accurately computing the WCRTs of tasks and messages. By using a holistic approach it may be possible to achieve higher system resource utilization due to the consideration of the system as a whole (e.g. the observed pipeline effect presented in Chapter 6).

The main objective of the holistic analysis is to bound the *end-to-end* delay. The end-to-end delay between a pair of tasks is composed of the time required for producing a message at the sender side, the time for transmitting the message through the communication network, and the time required for processing the reception of the message at the receiver side. The previous works related to holistic analysis have mainly considered the sequential transactional model (see Figure 2.4). Some of those works are presented in the following.

In (Tindell and Clark, 1994) the authors presented a schedulability analysis for bounding the timing behaviour of a distributed hard real-time system by including both processing times and transmission delays. The research considered static priority preemptive tasks with arbitrary deadlines to be executed in a distributed system composed of a set of uni-processor nodes. Also, a TDMA protocol for inter-tasks communications was analysed. Their approach also considers the overheads incurred when messages are buffered and processed at the destination processor. They calculated the worst-case transmission delays of inter-task messages by considering synchronous patterns.

In a similar approach, (Spuri, 1996) considers a distributed system composed of a set of uni-processor nodes in which the EDF algorithm is used for task scheduling. Tasks with dynamic priorities and preemptive capabilities are analysed. As a network communication protocol, the author proposed the use of the Timed Token MAC protocol which queues outgoing packets using EDF. The author also proposed an iterative procedure for calculating the worst-case transmission delays of messages by considering both synchronous and asynchronous communication patterns.

In (Palencia and Gonzalez Harbour, 1998), the authors studied the inclusion of static and dynamic offsets in the schedulability analysis for preemptive fixed-priorities sequential tasks, thus reducing the pessimism of calculating the WCRT for such a real-time task model.

In (Gutierrez Garcia et al., 2000), the authors presented a schedulability analysis technique for distributed hard real-time systems in which responses of different events may synchronize with each other. This is a general method for computing the WCRT of different synchronization events. This method allows the study of complex synchronization structures, in which the fork-join structure is included. The technique is based on existing Rate Monotonic Analysis (RMA) techniques.

The authors of (Palencia and Harbour, 2003) introduced an offset-based analysis for dynamic priorities by considering preemptive EDF scheduling. The offset-based model considers the transactional model. The authors based their analysis on the work presented by (Spuri, 1996), but they consider the concept of offsets to avoid the pessimistic assumption of all tasks arriving simultaneously which leads to the worst-case execution in EDF scheduled systems.

In (Gutiérrez et al., 2014), the authors presented a holistic analysis for multi-packet messages in AFDX networks. The authors considered the scheduling of *virtual links* for the transmission of messages and their interaction (contention) in the receiving processors. Their analysis considers arbitrary message periods and release jitters.

None of the previous works addressed the problem of providing a holistic analysis for multi-threaded parallel real-task models in distributed systems. In this dissertation (see Chapter 6), a holistic timing analysis technique that considers parallel real-time tasks executing in a distributed system (Garibay-Martínez et al., 2014b) is considered. The analysis is extended to consider the FTT-SE network (Pedreiras et al., 2005; Marau et al., 2006). A technique to reduce the pessimism of the WCRT analysis of such distributed systems is also proposed.

2.6 Parallel Programming Models

Parallel execution model has been known (e.g., High Performance Computing (HPC)) for several decades and used on several computational demanding applications, like weather forecast, however common modern applications demand more computing power. A solution for providing the required computing power is now offered by new commercial multiprocessor platforms.

Parallel processing can increase the performance of applications by executing them on multiple processors at the same time. However, in order to fully exploit the capacities of multiprocessor architectures, it is needed to apply correct programming techniques.

Programming parallel applications is not a straight-forward task when compared with sequential programming. Furthermore, it implies a burden on the programmer and in some cases it implies the detailed knowledge of the targeted computing platform. On that respect, parallel programming approaches are mainly divided in two categories (Kasim et al., 2008):

i. ***Auto-parallelisation.*** Sequential programs are automatically parallelised by using Instruction Level Parallelism (ILP) or parallel enabled compilers. The main advantage of this type of parallelisation is the simplicity of obtaining parallel programs from sequential ones. On the other hand, the achieved performance is poor when compared with the parallel programming approach;

ii. ***Parallel programming.*** The parallel programming approach implies programming: (i) the splitting mechanism of a problem into a set of tasks; and (ii) the development of a distribution mechanism that map those tasks onto processors in an efficient manner. Therefore, it requires more attention from the programmer, making it more difficult to code when compared with the auto-parallelisation approach. Consequently, it achieves higher execution performance.

This dissertation focuses on parallel programming models.

According to (Mattson et al., 2004), a parallel programming model is an abstraction of a general computing platform. Therefore, a programming model is not intended to be tight or specifically designed for a particular platform. However, in practice, programming models are closely associated with the computing architectures (e.g., shared memory, distributed memory, type of processor, etc.) they are designed for. Furthermore, combinations of such parallel technologies have lead towards the development of hybrid programming models. The aim of these hybrid approaches is to take advantage of the strong aspects of each technology, and thus, better exploit the available resources.

Parallel architectures and parallel models can be divided in two main categories:

i. ***Shared memory platforms.*** Shared memory platforms have a single memory access space which is accessible to all the processors in the platform. This is also referred as global memory space. Examples of these architectures are modern CPUs containing more than one processor (core) in the same chip;

ii. ***Distributed memory platforms.*** Conversely to shared memory platforms, on distributed memory platforms the concept of a global memory space does not exists, therefore, each processor (or set of processors) has its own private memory. Access to other processor memories need to be communicated through the network.

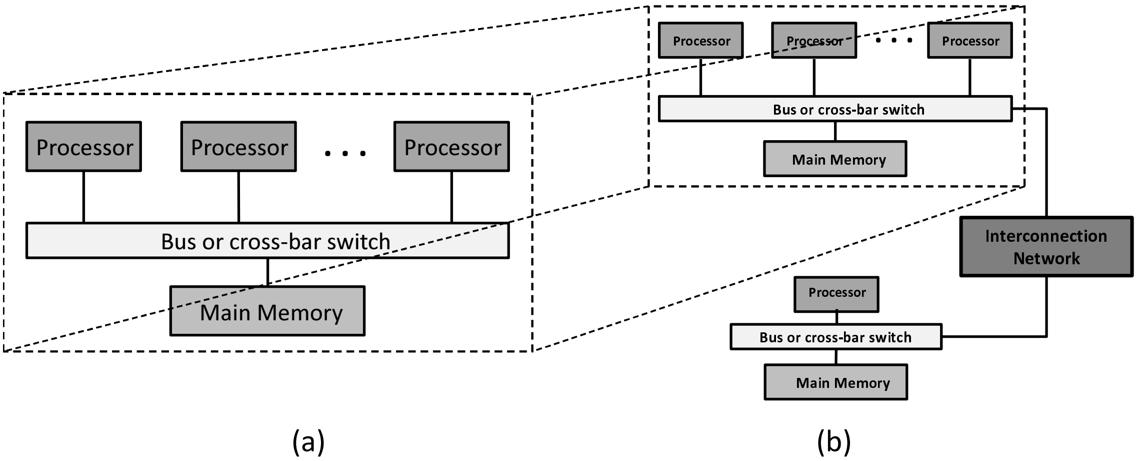

Figure 2.5: Example of (a) shared memory and (b) distributed memory platforms.

This dissertation focuses on distributed memory platforms.

Figure 2.5a, depicts an example of a shared memory architecture, in where all processors have access to the main memory through an internal bus or a cross bar switch. In contrast, in Figure 2.5b a distributed memory platforms is shown, where the access to other processors memories must be done through an interconnection network.

2.6.1 Programming Models for Shared Memory Platforms

Threads are usually associated with the implementation of applications for shared memory platforms and operating systems. In the following it is described some of the most relevant programming models for shared memory platforms with a special focus on those used for implementing embedded applications:

- **Portable Operating Systems Interface (POSIX) Threads** or **Pthreads**. Pthreads is a standard library (Standard, 2012) for the creation and manipulation of threads. The Pthreads library implements a series of functions based on the C language (and its variations), for the creation, coordination, memory access primitives and destruction of threads. Programmers of Pthreads need to put large effort on managing access to shared variables through control access mechanisms (e.g. *mutex* (mutual exclusion) mechanisms and semaphores). That complicates the programmability of large scale applications. However, Pthreads is valuable as concurrency model and its efficiency is high when used by experienced programmers.

- **OpenMP.** OpenMP is an API for the development of parallel programs based on the threading model of Pthreads. OpenMP is a compiler enabled programming model for C/C++ and FORTRAN (OpenMP-Arch-Rev-Board, 2012). Similarly to Pthreds, OpenMP is conceived for the development of applications with shared memory platforms. Its main objective is to facilitate the development of parallel applications by applying the *auto-parallelisation* approach (a.k.a. incremental parallelisation approach). This is achieved through the use of the OpenMP library (libgomp.h), which implements a combination of compiler directives (*#pragmas*) and a set of runtime routines that provide the tools for managing the generated threads.

In contrast to Pthreds, the use of threads in OpenMP is highly structured because it was conceived for the developing of large scale parallel applications. OpenMP programming model is structured in blocks and follows the *fork-join* execution model. In OpenMP a thread called the *master* thread is in charge of starting the execution; afterwards this thread creates a *team* of threads and executes a set of instructions (a code block) in parallel. A brief description of the OpenMP programming model is given in Section 3.2.1.

- **Cilk.** Cilk is a multi-threaded programming model (similar to OpenMP). It is also based on the C programming language and it is intended for the development of parallel programs on shared memory platforms. It was initially developed by the Michigan Institute of Technology (MIT) (Cilk, 2013) and after adopted by Intel (Intel Cilk Plus) (Intel, 2013). Programmers must identify possible parallel sections by introducing Cilk's *spawn* keywords (in a similar way to *#pragmas* in OpenMP), with the objective of exploiting locality and leaving the run-time being responsible for executing computations in an efficient way.

- **Intel's Threading Building Block (TBB).** TBB is based on C++ templates containing a collection of data structures and parallel algorithms that allow programmers to avoid complications when programming with low-level multi-threaded programs such as Pthreads. TBB's main philosophy is to provide to programmers with a high-level of abstraction by applying concepts of tasks and algorithm skeletons. Therefore, programmers can use predefined algorithmic patterns, leaving the responsibility of matching the machine architecture to the TBB library. The main advantage of TBB over other programming models for shared memory is that it is based on algorithmic templates which provides high throughput. But unfortunately, programmers need to put more effort in producing parallel programs when compared to the auto-parallelisable approaches.

2.6.2 Programming Models for Distributed Memory Platforms

Processes are usually the common execution unit when implementing programming models for distributed platforms. Two special types of processes are used for the implementation of distributed systems: the *network socket* and a special type of inter-process communication processes (IPCs) the *Remote Procedure Call* (RPC).

Internet sockets are the endpoints used by the RPCs across a computer network. The main objective of internet sockets is to provide a mechanism for the delivery of data packets to the appropriate application processes. Internet socket APIs are based on the *Berkeley sockets* standard (Vessey and Skinner, 1990). Berkeley sockets is a library with an API for network sockets and UNIX domain sockets.

An RPC is a special type of IPC that allows a program to call a procedure to be executed in another address space (usually in another node). The programmer does not need to explicitly code the details for such remote call. One of the most used IPC methods are the *message passing methods*. Message passing methods are widely used in parallel computing models and supported by sending and receiving messages (e.g., complex data structures, segments of code, etc.) to/from other processes. For example:

- **MPI.** The MPI specification is based on the message passing paradigm. The MPI specification has become the *de facto* standard for developing parallel distributed programs using the message passing paradigm (MPI-Forum, 2012). Among other parallel programming approaches, it can implement the fork-join parallel programming model.

  MPI is not a language, but a standard library that can be used to build C/C++ and FORTRAN programs. MPI has the advantage that it can be compiled by simple compilers (on the contrary to OpenMP which needs a special compiler enabled to support it), by linking the MPI library during the compilation process.

  The potential of MPI programs is exploited by using the *communication routines* inside the MPI library. These communication routines make the workload distribution. A brief survey with the main features of the MPI programming model is presented in Section 3.2.2.

2.7 Summary

This chapter presented a brief survey of works that are relevant for this dissertation. It also reinforced the research context of this dissertation. Based on the topics described in the chapter, it is possible to summarise the focus of the dissertation. The dissertation focus on:

- Multi-threaded parallel models, in particular the fork-join structure. The model presented in this dissertation (see Chapter 3) considers:
  - fixed-priority sporadic task sets with implicit and constrained deadlines.
  - work-conserving non-preemptive (for message transmission) and preemptive (for in-node execution) systems.
- Distributed memory platforms that executes programs generated with an auto-parallelisation approach similar to the one in OpenMP. Related to the nodes composing the distributed platform:
  - It mainly considers identical nodes, with the exception of Chapter 5 and Chapter 7, in which a combination of nodes with different processing capabilities is considered.
  - The time-triggered and event-triggered communication patterns are considered.
- The scheduling algorithms derived on the dissertation are based on non-migration systems (i.e., fully-partitioned).

Chapter 3, introduces the *Fork-Join Parallel/Distributed Real-Time Task Model (P/D task model)*, the model is derived from a case study of the possible implementation of a parallel and distributed execution pattern with a combination of OpenMP (OpenMP-Arch-Rev-Board, 2012) and MPI (MPI-Forum, 2012) programming models.

Chapter 3

The Fork-Join Parallel/Distributed Real-Time Task Model (P/D Tasks)

3.1 Introduction

This chapter presents the *Fork-Join Parallel/Distributed Real-Time Task Model (P/D task model)*. Section 3.2 shows a case study of the possible implementation of a parallel and distributed execution with a combination of OpenMP (OpenMP-Arch-Rev-Board, 2012) and MPI (MPI-Forum, 2012) programming models. A timing model for OpenMP/MPI programs is derived by individually studying the behavior of OpenMP (OpenMP-Arch-Rev-Board, 2012) and MPI (MPI-Forum, 2012) programs (Garibay-Martínez et al., 2012). Section 3.4 presents the P/D task model which is derived from the observations presented in Section 3.2. The P/D task model is the model mostly used through the dissertation (with the exception of Section 5.2). A summary of the chapter is given in Section 3.5.

3.2 OpenMP + MPI Programming Models

This section considers the fork-join paradigm with some extensions for supporting distributed execution and real-time requirements. Section 3.2.1 introduces de OpenMP programming model, Section 3.2.2 presents the MPI programming model, and Section 3.3.1 presents a timing model based on the combination of OpenMP + MPI for supporting parallel/distributed real-time execution. Based on those observations, Section 3.4 presents the P/D task model.

3.2.1 OpenMP Programming Model

The OpenMP Application Program Interfaces (API) has been developed for providing portability and a user-friendly environment for programming shared memory multipro-

Algorithm 3.1: parallel sections construct and single construct example.

```
 1 #pragma omp parallel sections num_threads(2){
 2    #pragma omp section{
 3       #pragma omp single{
 4          /* variable declaration */
 5       }
 6    }
 7    #pragma omp section{
 8       function_1();
 9    }
10 }
```

cessor platforms (OpenMP-Arch-Rev-Board, 2012). The great success of OpenMP as a programming model relies on the simplicity of generating parallel programs. Furthermore, it is very efficient for creating incremental parallelism from existing sequential code.

OpenMP programs follow the fork-join paradigm. The structure of a parallel OpenMP program contains:

i. a sequential part (e.g., some C/C++ variable initialization);

ii. some parallel constructs (e.g., parallel sections) that are inserted in the code for making the parallel execution (fork), and finally;

iii. the set of generated threads are aggregated (e.g., with the `reduction` clause) to generate the final result (join).

In OpenMP, the *parallel* construct defines a segment of the code to be executed in parallel; this segment is known as *parallel region*. The construct is defined by the `#pragma omp parallel` directive (see Algorithm 3.1). Whenever a thread encounters a *parallel* construct, this thread becomes the *master thread* and it creates a team of threads, which will run the code inside that code segment.

At the end of a *parallel region* there is an implicit synchronization mechanism (represented by the symbol }) called *barrier* (line 9, Algorithm 3.1). Each thread executes its associated code and waits at the barrier, when all threads complete their execution only the master thread continues. There is a possibility of using a `nowait` clause which inhibits the *barrier* and allows continuing the execution. The `nowait` clause is not considered in this dissertation because the dissertation focuses on the classical fork-join model. In the classical fork-join model, a synchronization point (the join operation) is needed.

In order to provide the desired functionality to *parallel* constructs, OpenMP provides a set of *work-sharing* constructs. These constructs are in charge of distributing the workload among threads in OpenMP programs.

These *work-sharing* constructs can be complemented and/or modified with a set of clauses to control the parallel execution. Of particular importance for this work is the `numthreads(n)` clause, which is used to specify the number of threads to execute on a *parallel region*. The most relevant *work-sharing* constructs follow.

The amount of parallelism achieved by a *sections* construct is a function of the number of threads and the number of individual *parallel section* clauses associated to it. Each section is defined by the `#pragma omp section` directive.

The single construct (`#pragma omp single`) specifies that the associated code block is executed by only one of the threads in the team (not necessarily the master thread). The other threads in the team, wait at an implicit barrier at the end of the single construct (line 5, Algorithm 3.1). In a similar way the `#pragma omp master` directive guarantees that only the master thread will execute a specific code block. Algorithm 3.1 shows a fragment of code related to the use of the `parallel section` and the `single` constructs.

The *for loop* construct is signalled to the compiler through the `#pragma omp for` directive and it is used for dividing the `for` cycle iteration among several threads. This directive has different behaviours depending on the scheduler type selected, which is determined by internal OpenMP variables or by the `schedule(...)` clause. There are two standard types of `schedule(...)` clauses: `static` and `dynamic`. In the `static` type, iterations are assigned to threads in a round-robin fashion. In case the `dynamic` scheduler type is chosen, *chunk* iterations are assigned to threads on request. In a similar manner as a *work sharing pool*. More details about the scheduler type are given in Section 3.3.2.

The `#pragma omp for` directive might also be associated with a `reduction` clause. Algorithm 3.2, depicts a fragment of code exemplifying the `parallel for` construct in OpenMP. In that example, the `for` loop iterations are divided over three threads (through the `num_threads(3)` clause), each one executing two iterations. OpenMP also provides functionality for nested parallelism. For instance, whenever a thread encounters another *parallel* construct, it creates a new team of threads, and the thread that created the team becomes the new master thread of that team. This allows exploiting extra parallelism in OpenMP programs. However, no nested parallelism is considered in this dissertation.

Algorithm 3.2: `parallel for` example.

```
#pragma omp parallel for num_threads(3) reduction(+:sum){
   for (i = 0; i < 6; i++){
      loopCode();
   }
}
```

3.2.2 MPI Programming model

The MPI specification has become a *de facto* standard for developing parallel distributed programs using the message passing paradigm (MPI-Forum, 2012), which among others can implement the fork-join programming model. Common fork-join programs implemented in MPI have:

i. a serial execution of a code segment (e.g., variables declaration and initialization);

ii. the MPI environment initialization (e.g., a call to `MPI_Init(...)`);

iii. an explicit *work-sharing* algorithm, for the distribution of the workload among the processing elements (the *work-sharing* algorithm is implemented by the programmer);

iv. the transfer of data using message passing functions (e.g., calls to `MPI_Send(...)`, `MPI_Recv(...)`);

v. an execution of the computations on the remote and local nodes;

vi. a reduce of the partial results from remote nodes to obtain the final one (e.g., a call to `MPI_Reduce(...)`);

vii. the finalization of the execution (e.g., a call to `MPI_Finalize(...)`).

A code fragment is presented in Algorithm 3.3.

A normal MPI program starts with a call to `MPI_Init(...)` routine to initialize the MPI environment (line 2, Algorithm 3.3). This creates a *communicator*, which groups a set of MPI processes in the local or in different nodes. All MPI messages must specify a *communicator* for the interchange of messages between the processes belonging to the same *communicator*.

`MPI_Comm_rank(...)`, returns the "rank" (the ID) of a process within the associated communicator.

Algorithm 3.3: MPI two-sided send/receive example.

```
/* variable declaration */
MPI_Init(...);
MPI_Comm_size(...);
MPI_Comm_rank(...);
if (rank == 0) {
    MPI_Send(...);
    MPI_Recv(...);
}
if (rank ≠ 0) {
    MPI_Recv(...);
   /* execution */
    MPI_Send(...);
}
MPI_Finalize();
```

The potential of MPI programs is given due to the *communication routines*. These communication routines are the ones that make the distribution of workload to processes. MPI communications can be Point-to-Point or Collective. The most used MPI communication functions are `MPI_Send(...)` and `MPI_Recv(...)`, where the first is a non-blocking call used to send a block of data to be processed and the second blocks until a message is received.

MPI also implements reduce operations, in an analogous way to OpenMP through the `MPI_Reduce(...)` function.

One important difference between OpenMP and MPI is that MPI leaves all the burden of the parallel coding to the programmer, while OpenMP supports incremental parallelism at the cost of allowing less flexibility.

3.3 Supporting Parallel and Distributed Real-Time Execution with OpenMP + MPI

Based on the OpenMP and MPI constructs introduced in Section 3.2.1 and Section 3.2.2, in this section a model for supporting parallel/distributed execution with real-time constraints is proposed.

In the framework presented in this chapter, it is assumed that the programmer only writes code using the OpenMP API with minor changes to the OpenMP specification. The last with the intention of reducing the complexity of writing parallel distributed programs. The changes to the OpenMP specification include the extension of existing OpenMP constructs for enabling them to support workload distribution (i.e., supported by

Algorithm 3.4: `distributedParallel` clause example.

```
#pragma omp distributedParallel for deadline(200) num_threads(4){
   for (i = 0; i < 4; i++){
      loopCode();
   }
}
```

MPI). Therefore, the MPI code is not seen by the programmer (the MPI code is implicitly called by the OpenMP library). The programmer only needs to specify which OpenMP code blocks to distribute by using the `#pragma omp distributedParallel` pragma, and specifying their deadlines (Garibay-Martínez et al., 2012, 2013a). This is illustrated in Algorithm 3.4, the `for` loop can be distributed among 3 threads and the computation must be completed before a deadline of 200 milliseconds. In this case, the `distributedParallel` directive, signals the compiler to enable the parallelisation of some iterations of the parallel `for` loop on distributed nodes. It is also the responsibility of the compiler to generate code that can be dynamically or statically parallelised on the destination node.

Dynamic means that the run-time decides the number of threads to split the computation on the neighbour node(s), according to the availability of resources. Static means that it is the programmer who guides the splitting procedure.

Programs based on the dynamic computation model have an execution time line similar to the one in Figure 3.1, which is related to the code in Algorithm 3.4. In Figure 3.1, the horizontal lines represent threads and the vertical lines represent forks and joins. In this case, the parallel `for` clause splits into three threads, two are executed on the local node and another is executed on a cooperative node. This type of execution is called *remote execution*.

Furthermore, it is assumed that it is possible to split the remote execution into two threads. Observing the time line in Figure 3.1, it is assumed that the execution starts with the thread θ_1, θ_1 is split into two threads θ_1 and θ_2 which execute locally one `for` loop iteration each, the distributed thread θ_3 executes the remaining two iterations. Thread θ_3 is hosted in a neighbour node and further split into two threads, by adding thread θ_4, each one of these threads is executing one iteration of the `for` loop. In Figure 3.1 it is also possible to see the messages for transmitting and receiving code or data ($\mu_{1,3}$ and $\mu_{3,1}$). It is assumed that it is the responsibility of the master thread to marshal and send the data required for remote execution, and for receiving processed data and unmarshalling it.

Similarly, Figure 3.2 shows an example of the execution of a OpenMP/MPI program

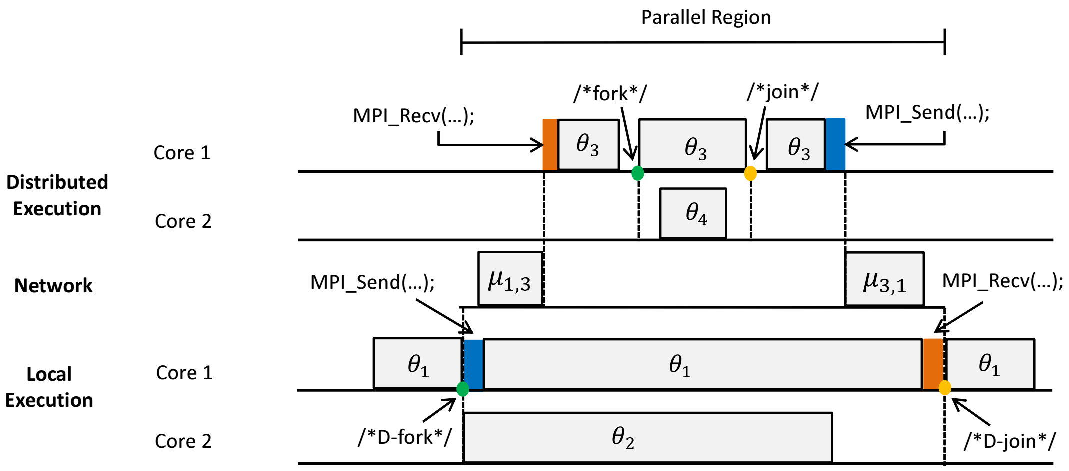

Figure 3.1: Execution of a OpenMP/MPI program based on the dynamic computation model.

based on the static computation model. Figure 3.2, shows a parallel distributed task in which 4 threads are executed, two are executed on the *local* node and another is executed on a *remote* node. The execution starts with the thread θ_1, θ_1 is split into 4 threads θ_1, θ_2, θ_3 and θ_4. Threads θ_1 and θ_2, execute locally and threads θ_3 and θ_4 are executed in a remote node. In Figure 3.2 it is also possible to see the messages for transmitting and receiving code or data. The main difference between the dynamic model (Figure 3.1) and the static model (Figure 3.2) is that in the static model the for and the join operations are realised only at the local node.

This dissertation focuses on the OpenMP/MPI programs based on distribution of parallel operations organized at the local node: static computation model.

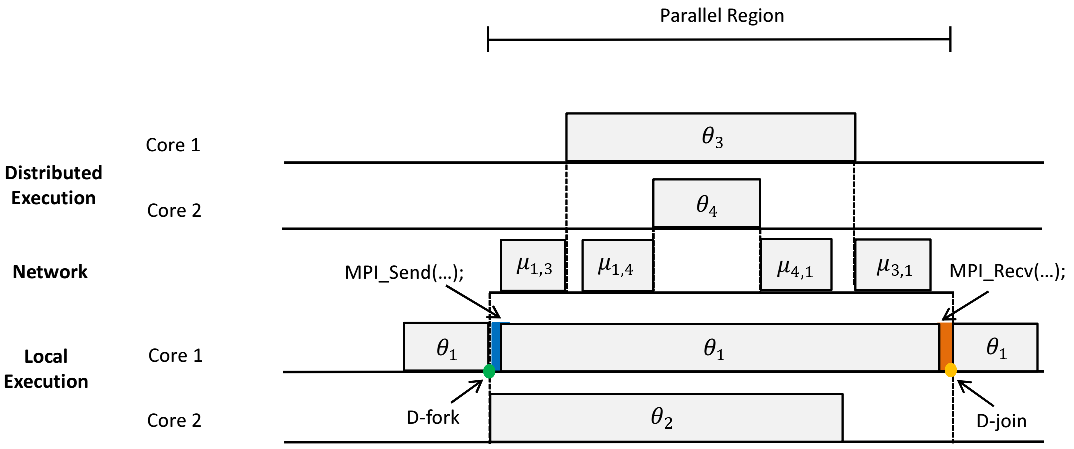

Figure 3.2: Execution of a OpenMP/MPI program based on the static computation model.

3.3.1 Timing Model for OpenMP/MPI Programs

This section provides an overview of the envisaged programming and execution model based on the OpenMP/MPI models. Section 3.4 presents an extended and formal version of the execution model used through out the dissertation.

The generic operation of the local thread that controls the remote execution (the master thread) is as follows:

i. the local thread must issue a `MPI_Init(...)` to initialize the MPI environment;

ii. it determines the data to be sent and sends it using `MPI_Send(...);`

iii. the data gets transmitted through the network;

iv. the data is received on the remote node and executed as an OpenMP program;

v. when the execution is finished, the results are sent back to the local node which should be waiting for the results, by calling `MPI_Recv(...).`

It is assumed that the neighbour node already has the code to be executed. Therefore, the costs of transmitting and installing the code are not considered. Such operation can be executed during the system set up phase. In order to combine the functionality of OpenMP and MPI in a single program, both models need to reach certain commitments to guarantee the correct execution of hybrid programs. For example, on the OpenMP side, it is required to guarantee that if a single thread is blocked by an operating systems call, all the other threads can still be runnable. This is already supported by the most recent OpenMP implementations (OpenMP-Arch-Rev-Board, 2012). On the MPI side, from the release of MPI-2 standard (MPI-Forum, 2012), the concept of level of thread support has been defined. There are four levels of support:

i. `MPI_THREAD_SINGLE`: only one thread exists in the application;

ii. `MPI_THREAD_FUNNELED`: multiple threads can exist but only the master thread can make MPI calls;

iii. `MPI_THREAD_SERIALIZED`, multiple threads can exists, each thread can make MPI calls as long as there is no other thread making a call, and;

iv. `MPI_THREAD_MULTIPLE`, multiple threads can exist and they can make MPI calls at any time.

Using MPI_THREAD_SINGLE, would make impossible the splitting of the team of threads among the neighbour nodes since only one thread per node is allowed. Any other option, allows the execution of such programs according to what has been described. MPI has some limitations for guaranteeing real-time communications (e.g., it is required a mechanism for bandwidth reservation). However, for overcoming such limitations the authors of (Kanevsky et al., 1998) introduced an extension for MPI Real-Time (MPI/RT), the MPI/RT standard is based on channel reservation and fault tolerant mechanisms to guarantee time properties.

To model an hybrid OpenMP/MPI program it is considered a set τ of n periodic tasks denoted by $\{\tau_1, \ldots, \tau_n\}$. Each task τ_i, $i \in [1,n]$ is potentially a parallel task composed of a set of n_i segments. Each parallel segment $\sigma_{i,2j}$, $i \in [1,n], j \in [1,n_i]$ may further be composed of $b_{i,j}$ potentially parallel code blocks (e.g., the code inside a parallel for loop, etc.) denoted by $b_{i,j,k}$, $i \in [1,n]$, $j \in [1,n_i]$, $k \in [1,b_{i,j}]$, it is assumed that each code block has a WCET denoted as $C^{cb}_{i,j,k}$. For example, consider line 3 in Algorithm 3.4, the parallel for loop has 4 iterations thus there are 4 different code blocks (assuming a chunk size of 1) that can be executed in parallel. Each segment can be executed by a set of $n_{i,j}$ threads denoted as $\theta_{i,j,k}$, $i \in [1,n]$, $j \in [1,n_i]$, $k \in [1,n_{i,j}]$.

3.3.2 Timing Behaviour of OpenMP programs

To correctly characterize the OpenMP timing behaviour, it is necessary to analyse the transformation process from high level #pragma directives to standard C/C++ code, which is finally compiled by the C/C++ compiler.

The process of converting OpenMP constructs to multi-thread code is known as *lowering* the code. The lowered code makes the calls to the OpenMP run-time environment. OpenMP compilers make this lowering process in two phases: (i) *the pre-lowering* and (ii) *lowering*.

The *pre-lowering* phase is in charge of transforming (simplifying) some OpenMP *work-sharing* constructs in other equivalent ones, with the objective of facilitating later processing.

This is the case of the sections construct and the single construct. In particular, the sections construct is converted into an equivalent *for* loop and each section construct corresponds to one iteration in the loop. After this transformation, each iteration is scheduled according to the scheduler type in use, which can be defined as static or dynamic. Also, the single construct is transformed to a for loop with just one iteration. If a *parallel region* uses the single construct, the schedule clause is always defined as dynamic.

The *lowering* phase takes the pre-lowered code and performs the transformation to C/C++ code. The *lowering* step realizes a transformation known as *outlining* the code. Outlining is the process of transforming lexically existing code into a new procedure and this new procedure is passed as an argument to the runtime libraries of OpenMP. OpenMP does this outlining process to code inside *parallel regions*. After the outlining phase, the compiler calls the OpenMP run-time which is in charge of mapping code to threads. Therefore, the instructions that are scheduled and processed are the *lowered* ones.

After *lowering* the code, the final mapping from code blocks to threads depends on the scheduler type that is used to assign (map) code blocks to threads.

Whenever the `schedule` clause is defined as `static`, the iterations are assigned to threads in a round-robin fashion. In this model, the *chunk_size* is the number of lowered iterations inside a `for` loop. When the parameter *chunk_size* is not specified, the *chunk_size* is approximately the same for all threads; equal to the number of iterations divided by the number of threads. However, regardless the *chunk_size*, Eq. (3.1) computes the number of lowered code blocks $b_{i,j,k}$ that are mapped into threads when the `static` scheduler type is used to schedule *lowered* code:

$$nb_{i,j,k}^{\max} = \left\lceil \frac{b_{i,j}}{n_{i,j}} \right\rceil . \tag{3.1}$$

This is the maximum number of code blocks $b_{i,j,k}$ to be assigned to each thread $\theta_{i,j,k}$. Then, if it is considered the maximum WCET of a code block $b_{i,j,k}$ (denoted as $C_{i,j,k}^{max}$), an upper bound for the WCET of a thread $\theta_{i,j,k}$ (denoted as $C_{\theta_{i,j,k}}^{max}$) can be derived as:

$$C_{\theta_{i,j,k}}^{max} = nb_{i,j,k}^{\max} \times C_{i,j,k}^{\max} . \tag{3.2}$$

In case the `dynamic` scheduler type is chosen, *chunk* iterations are assigned to threads on request. In a similar manner as a *work sharing pool*. Whenever a thread finishes processing a *chunk*, it requests another until no more *chunks* are available. Therefore, the `dynamic` scheduler type can potentially offer better performance, especially when the execution times of the respective code blocks are not uniformly distributed (i.e., irregular parallelism). However, the upper bound in Eq. (3.2) is also an upper bound whenever the `dynamic` scheduler type is used. For illustrating the reasoning of this, consider the following example.

Example 3.1. *Consider two threads $\theta_{1,1,1}$ and $\theta_{1,1,2}$ to execute three code blocks $b_{1,1,1}$, $b_{1,1,2}$ and $b_{1,1,3}$, with execution times of $C_{1,1,1}^{b}$, $C_{1,1,2}^{b} + \varepsilon$ and $C_{1,1,3}^{b} + 2 \times \varepsilon$, where ε represents a very small execution time quantity; that is, ε approaches zero. Suppose that code blocks $b_{1,1,1}$, $b_{1,1,2}$ are being executed by threads $\theta_{1,1,1}$ and $\theta_{1,1,2}$, respectively. Then, $\theta_{1,1,1}$ ends its execution and request the next code block $b_{1,1,3}$, which is the one having the maximum WCET $C_{i,j,k}^{\max}$. From Eq. (3.1) it is known that each thread has a maximum*

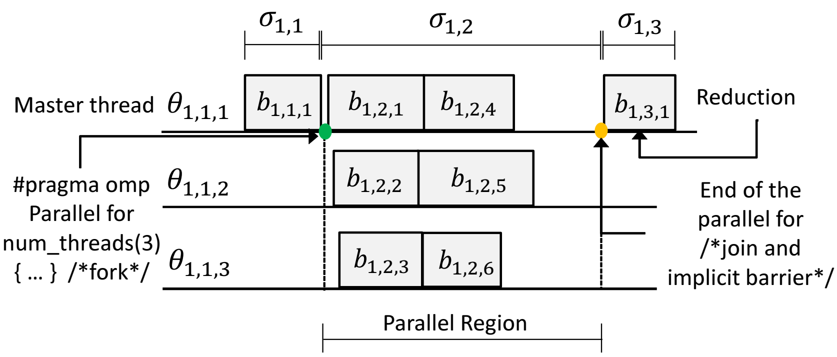

Figure 3.3: Timing execution of code block $b_{i,j,k}$ of a *parallel for* in OpenMP programs.

$nb^{\max}_{i,j,k}$ of two. Hence, it is possible to see that in this example $C^{\max}_{\theta_{i,j,k}}$ would not be bigger than $2 \times C^{\max}_{i,j,k}$. Thus Eq. (3.2) also holds as an upper bound for the `dynamic` *scheduler.*

OpenMP supports other scheduler types, such as `guided`, `runtime` and other variations. But they are implementation dependent and therefore they are not considered.

After the initialization of an OpenMP program, a task τ_i is executed sequentially, and is only composed by the *master thread*. Whenever it encounters a parallel region, the master thread forks and creates a team of $n_{i,j}$ threads belonging to task τ_i. The number of threads to be created, is explicitly expressed by the `num_threads(`$n_{i,j}$`)` clause.

An example of a typical OpenMP program is depicted in Algorithm 3.2. In line 1, a `#pragma omp parallel for` directive is encountered, which also includes a `num_threads(3)` clause and a reduction clause. In this case, three threads are to share the iterations of a *for* loop. Iterations in a `parallel for` loop are divided in *chunks* that are assigned to threads. In that specific example, the number of iterations to share is 6, and then assuming that the default scheduler typr is `static`, the threads $\theta_{i,j,k}$ with $k = 1,2,3$ share two *chunks* each in a round robin manner. A possible time line for the execution of the code presented in Algorithm 3.2, divided among three threads is depicted in Figure 3.3. This Figure shows three different code segments: $\sigma_1, \sigma_2, \sigma_3$, with its code blocks. Code block $b_{1,1,1}$ corresponds to serial code being executed prior to the parallel region, then code blocks $b_{1,2,1-6}$ represent the execution of the code in line 3, the function `loopcode()`. Code block $b_{1,3,1}$, in segment $\sigma_{1,3}$ corresponds to the execution of the reduction clause. Section 3.3.4 presents an algorithm to map the code block into a set of threads.

3.3.3 Timing Behaviour of MPI communications

In contrast to the use of threads in OpenMP, MPI uses processes as execution units for implementing two-sided communication. But for modelling purposes it is not distinguished between threads and processes, therefore the same notation is used.

Figure 3.1 shows the code depicted in Algorithm 3.3, it is possible to notice that during the transmission of data to be used by thread θ_3, there is a transmission delay that depends on the size of the data to transfer and the network protocol. The processes θ_1 and θ_3 hosted in two different nodes incur in a transmission delay for message $\mu_{1,3}$. A similar process is shown in Figure 3.2.

Note that the transmission delay is an important parameter to consider when analysing hibrid OpenMP/MPI programs and cannot be considered negligible.

3.3.4 Timing Behaviour of OpenMP + MPI

In order to consider hybrid execution, the OpenMP model is extended into a Directed Acyclic Graph (DAG) that allows to model the behaviour of the program. Please note that given the structure of programs based on the combination of OpenMP and MPI, the generated DAG is always a fork-join DAG. The goal is to provide a DAG that can be handled by a real-time schedulability test as the one presented in Chapter 6.

The execution time of a task τ_i can be represented by a fork-join DAG. A DAG $G(V,E)$ is able to capture the combination of sequential and parallel code blocks in parallel/distributed programs and the possible dependencies between them. The hybrid model is based on two different graphs the *Graph of Code Blocks* (denoted as $GCB(V,E)$) and the *Graph of Threads* (denoted as $GT(V^*,E^*)$), where $GCB(V,E)$ represents the dependencies between code blocks in a program and $GT(V^*,E^*)$ represents the mapping of such blocks to threads.

The graph $GCB(V,E)$ represents the structure of the program with the code blocks that may be executed in parallel and the code blocks that may be executed sequentially. The set of vertices in $V = v_0,\ldots,v_k$, represent the set of code blocks $b_{i,j,k}$, and the set of edges $E = \{v_0,v_1\},\ldots,\{v_{k-1},v_k\}$, represent the dependencies between them. If a vertex v_1 precedes v_2, denoted by $v_1 \prec v_2$, indicates that a vertex v_1 must complete its execution, before v_2 can start its execution. The relation $\prec$ indicates a predecessor-to-successor relation.

The dependencies in a hybrid OpenMP/MPI program can be imposed by implicit synchronization points (e.g., `single` constructs, `master` constructs, etc.), explicit barriers or memory synchronization (e.g., critical sections, flush operations, etc.); just to mention some. Those dependencies are related to OpenMP. If there is no precedence relation between nodes v_1 and v_2 they are logically parallel, and therefore, they can be executed in parallel.

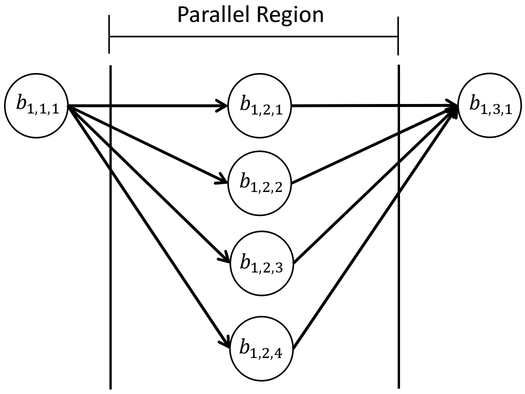

Figure 3.4: Code blocks DAG $GCB(V,E)$.

Given a graph $GCB(V,E)$, it is needed to map this code blocks graph into the graph of threads $GT(V^*,E^*)$. Where the set of vertices $V^* = v_0^*,\ldots,v_k^*$ in $GT(V^*,E^*)$, represent the set of code blocks $b_{i,j,k}$ and the set of edges $E^* = \{(v_0^*,v_1^*),\ldots,(v_{k-1}^*,v_k^*)\}$ represent the order of execution of code blocks assigned to the threads in $\theta_{i,j,k}$.

To obtain $GT(V^*,E^*)$, it is needed to traverse $GCB(V,E)$ for obtaining a tree that contains predecessor-to-successor relations indicating which code blocks precedes others. Each branch in the tree corresponds to the execution of successive code blocks belonging to a thread, that is $GT(V^*,E^*)$ has exactly the same number of branches than threads executing in the program.

The traverse mechanism is the *Breadth-First Search* (BFS) algorithm (Cormen et al., 2001). The BFS algorithm systematically discovers every vertex that is reachable from a source node s. The BFS expands the frontier between discovered and non-discovered nodes uniformly across the breadth. This means that all vertex at distance k from the source node s are discovered before discovering another vertex from distance $k+1$. Algorithm 3.5 shows the pseudo code of the BFS algorithm. For more details please refer to (Cormen et al., 2001). This is particularly useful because all discovered vertex may be executed in parallel since they do not have precedence constraints between them. In the BFS algorithm, the discovered nodes that are reachable from s are maintained in a queue before deciding to discover another level in the DAG. This queue can be assigned to threads according to the defined schedule type (`static` or `dynamic`) and respecting the maximum blocks per thread as specified in Eq. (3.1). Consider the following example:

Example 3.2. *Assume the code blocks inside a parallel region in a DAG $GCB(V,E)$ as the one shown in Figure 3.4. Also assume that there are four threads to assign the code blocks. After applying BFS algorithm to GCB according to a static schedule type, it is possible to obtain a DAG $GT(V^*,E^*)$ as shown in Figure 3.5a. That DAG is obtained by considering a computing platform similar to the one presented in Figure 3.2, where two cores are for local execution and two cores are available for remote execution. Since 4*

Algorithm 3.5: BFS(G, s) (Cormen et al., 2001)

```
for each vertex $u \in V[G] - \{s\}$ do
    $color[u] \leftarrow WHITE$;
    $d[u] \leftarrow \infty$;
    $p[u] \leftarrow NIL$;
$colors[s] \leftarrow GRAY$;
$d[s] \leftarrow 0$;
$p[s] \leftarrow NIL$;
$Q \leftarrow \emptyset$;
$ENQUEUE(Q, s)$;
while $Q$ is not empty do
    $u \leftarrow DEQUEUE(Q)$;
    for each $v \in Adj[u]$ do
        if $color[v] = WHITE$ then
            $colors[v] \leftarrow GRAY$;
            $d[v] \leftarrow d[u] + 1$;
            $p[v] \leftarrow u$;
            $ENQUEUE(Q, v)$;
    $colors[u] \leftarrow BLACK$;
```

threads are available, two are assigned to be executed locally and two are assigned to be executed remotely (code blocks $b_{1,2,4}$ and $b_{1,2,4}$ are executed remotely). In Figure 3.5b the same example is represented but in that case only three threads are available for execution, therefore, code blocks $b_{1,2,1}$, $b_{1,2,2}$ and $b_{1,2,4}$ are executed locally, and $b_{1,2,3}$ is executed remotely.

With this approach, designers can transform the applications structure into a model which can be analysed in terms of timing behaviour.

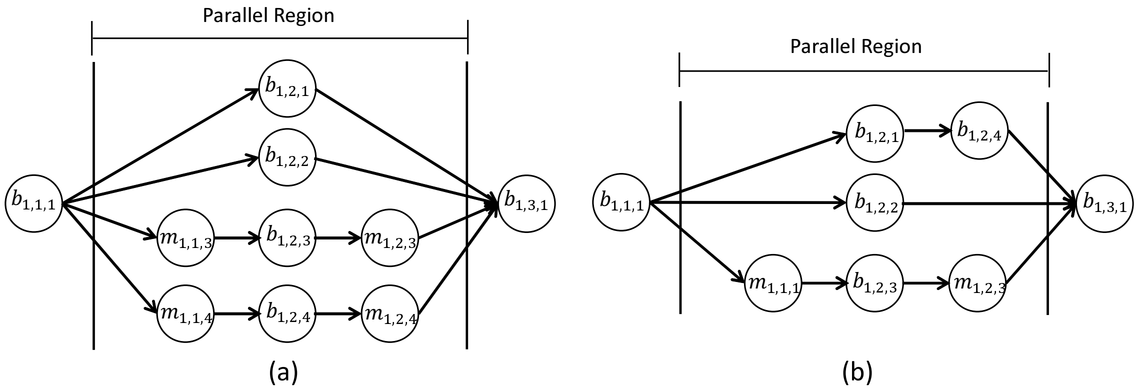

Figure 3.5: Thread Blocks DAG $GT(V^*, E^*)$.

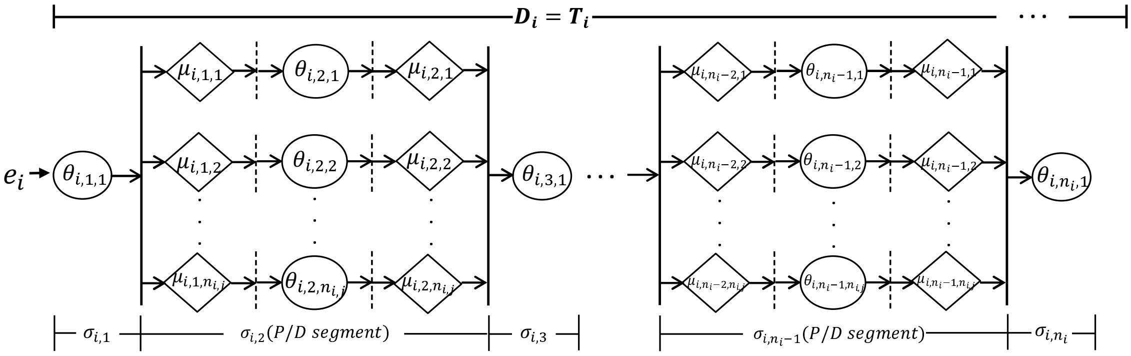

Figure 3.6: Fork-Join P/D Real-Time Task Model (P/D tasks).

3.4 Fork-Join Parallel/Distributed Real-Time (P/D) Task Model

In this section, the fork-join Parallel/Distributed real-time (P/D) task model is introduced. The P/D task model is derived from the observations presented in Section 3.2 and Section 3.3. However, the P/D task model is able to model any program in which the computations are based on the fork-join paradigm, regardless the implementation technologies used. A real-time distributed system is composed of two main elements:

i. a *distributed computing platform*, and;

ii. a *set of real-time software applications*.

This dissertation considers a distributed computing platform composed of a set of m identical nodes $\pi = \{\pi_1, \dots, \pi_m\}$ (uni-processor nodes and multi-processor nodes) interconnected with a fixed-priority real-time network ϖ (e.g., FTT Ethernet (Pedreiras and Luis, 2003)). The real-time network ϖ is composed of a set $\{SW_1, \dots, SW_r\}$ of r switches. The switches SW_x ($x \in \{1, \dots, r\}$), and their respective links, interconnect all the distributed nodes in the network. The number of nodes m is defined by the architecture.

A set of real-time software applications is represented as a set τ of fork-join Parallel/Distributed real-time (P/D) tasks. A task τ_i ($i \in \{1, \dots, n\}$) is composed of a sequence of sequential and Parallel/Distributed (P/D) segments $\sigma_{i,j}$ with $j \in \{1, \dots, n_i\}$. Figure 3.6 shows an example of a P/D task τ_i. Where, n_i represents the number of segments composing τ_i, n_i is assumed to be an odd integer, as a P/D task should always start and finish with a sequential segment. Therefore, odd segments $\sigma_{i,2j+1}$ identify sequential segments and even segments $\sigma_{i,2j}$ identify P/D segments. Each segment $\sigma_{i,j}$ is composed of a subset of threads $\theta_{i,j,k}$ with $k \in \{1, \dots, n_{i,j}\}$, where $n_{i,j} = 1$ for sequential segments and $n_{i,j} = m_i \leq m$ threads for P/D segments.

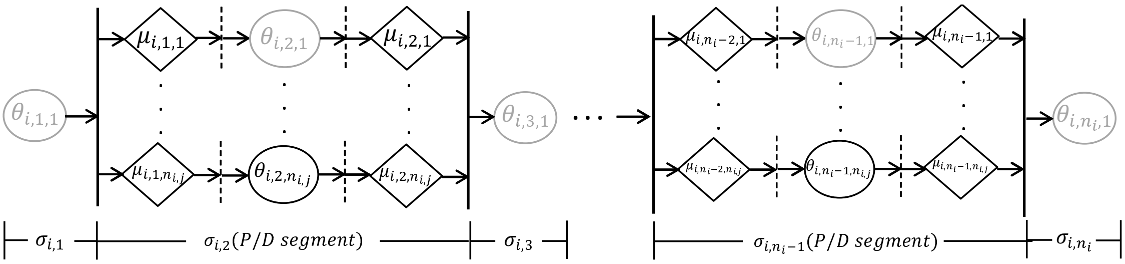

Figure 3.7: Master thread.

A P/D task starts by a master thread executing sequentially, and afterwards it forks to be executed in parallel on a remote or local processors. When the parallel execution has completed on each of the remote processors, the results are aggregated by performing a join operation and the execution of the sequential thread is resumed within the master thread. These operations are referred as Distributed-Fork (D-Fork) and Distributed-Join (D-Join). All sequential segments within a P/D task τ_i must execute within the same processor. This means that the processor that performs a D-Fork operation (invoker node) is in charge of aggregating the result by performing a D-Join operation. Thus, the *master thread* of a P/D task τ_i is denoted as τ_i^{master} and defined as:

Definition 3.1. ***(Master Thread).*** *The master thread of a P/D task τ_i is the collection of all threads $\theta_{i,j,1}$ belonging to all segments $\sigma_{i,j}$ that execute on the invoker node. A master thread can be represented as:*

$$\tau_i^{master} = \{\theta_{i,1,1}, \theta_{i,2,1}, \theta_{i,3,1}, \ldots, \theta_{i,n_i-1,1}, \theta_{i,n_i,1}\}. \tag{3.3}$$

Figure 3.7 shows an example of the threads $\theta_{i,j,k}$ of a P/D task τ_i that belong to the master thread τ_i^{master}.

Threads in a P/D segment are possibly executed on remote nodes. Consequently, for each thread $\theta_{i,2j,k}$ belonging to a P/D segment (P/D thread), two P/D messages $\mu_{i,2j-1,k}$ and $\mu_{i,2j,k}$ are considered for communication between the invoker and remote nodes. This is, P/D threads and messages that belong to a P/D segment and execute on a remote processor, have a precedence relation: $\mu_{i,2j-1,k} \prec \theta_{i,2j,k} \prec \mu_{i,2j,k}$. That precedence relation is called *Distributed Execution Path* (DEP), and it is denoted as $DP_{i,2j,k}$. For each P/D segment, there exists a *synchronization point* at the end of the segment, indicating that no thread that belongs to the segment after the synchronization point can start executing before all threads of the current segment have completed execution.

Each thread $\theta_{i,j,k}$ has a Worst-Case Execution Time (WCET) of $C_{i,j,k}$, and each message $\mu_{i,j,k}$ has a WCML $M_{i,j,k}$. P/D threads are preemptive, but messages are non-preemptive.

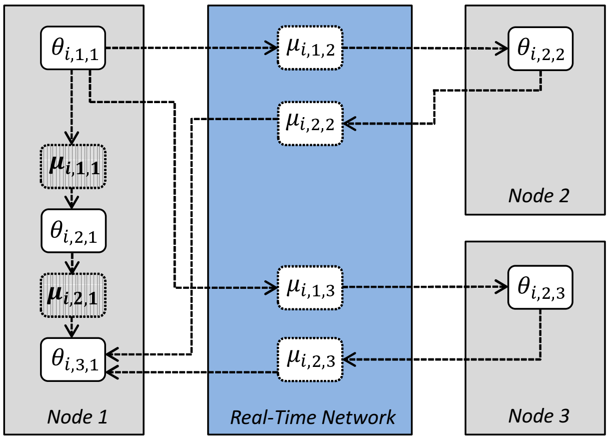

Figure 3.8: Generic distributed computing platform.

Communications between threads can be carried out within the same or between different processor nodes. If two threads $\theta_{i,j,k}$ and $\theta_{i,j+1,k}$ communicate via a message $\mu_{i,j,k}$ and execute on the same processor, we consider that the message transmission time is negligible, thereby assuming that $M_{i,j,k} = 0$. Figure 3.8 shows an example of a generic distributed computing platform. On the figure, it is possible to notice a task τ_i with three P/D threads $\theta_{i,2,1}, \theta_{i,2,2}$, and $\theta_{i,2,3}$. Since $\theta_{i,2,1}$ is kept for local execution, its respective messages $\mu_{i,1,1}$ and $\mu_{i,2,1}$ are omitted.

3.5 Summary

This chapter proposed the system model to be used through this dissertation. The model considers a case study of programs that are written with a combination of OpenMP and MPI programming models. However, the P/D task model is able to capture the behaviour of any program that implements the fork-join paradigm. Furthermore, the proposed technique enables the timing characterization of these type of tasks (applications), transforming the code block structure of such programs into the execution graph represented by a graph of threads. Once the graph of threads have been obtained a proper schedulability analysis can be performed (e.g., the one presented in Section 6).

A limitation of the fork-join Parallel/Distributed real-time (P/D) task model does not consider nested parallelism.

The following publications are related to the work presented in this chapter:

- **R. Garibay-Martínez**, L.L. Ferreira, and L.M. Pinho. A framework for the development of parallel and distributed real-time embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 39–46, Sept 2012. doi: 10.1109/SEAA.2012.60.

- **R. Garibay-Martínez**, L.L. Ferreira, C. Maia, and L.M. Pinho. Towards transparent parallel/distributed support for real-time embedded applications. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 114–117, June 2013. doi: 10.1109/SIES.2013.6601483.

Chapter 4

Scheduling P/D Tasks in Distributed Uni-processor Systems

4.1 Introduction

Works on fixed-priority multi-threaded parallel task models for multiprocessor systems are presented in 2.2.4. This chapter presents an effort towards the integration of the parallel real-time task models and distributed systems. Contrarily to multi-core systems, the transmission delay of messages sent between nodes of the distributed system have to be considered and cannot be deemed negligible.

In this chapter, the Partitioned/Distributed-Deadline Monotonic Scheduling (P/D--DMS) algorithm (Garibay-Martínez et al., 2014b) for P/D tasks is presented. The P/D-DMS algorithm is shown to have a resource augmentation bound of 4, which implies that any task set that is feasible on m unit-speed processors and a single shared bus real-time network, is schedulable by this algorithm on m processors and a single shared real-time network that are 4 times faster. Section 4.2 presents the Distributed Stretch Transformation model for P/D tasks. The resource augmentation bound for the Partitioned-Distributed-DMS algorithm is explained in Section 4.3. The simulations that confirm the analytical results are provided in Section 4.4, and finally a summary of the chapter is given in Section 4.5.

4.1.1 Chapter Considerations

This chapter considers a distributed computing platform composed of a set of m identical uni-processor nodes $\pi = \{\pi_1, \dots, \pi_m\}$ interconnected with a fixed-priority shared bus real-time network ϖ. It also considers that for a task τ_i, every P/D thread $\theta_{i,2j,k}$ and their respective messages $\mu_{i,j,k}$ within a P/D segment $\sigma_{i,2j}$, have identical WCETs denoted as $P_{i,2j,k}$ and identical WCMLs $M_{i,j,k}$, respectively. However, the WCET and the WCML of

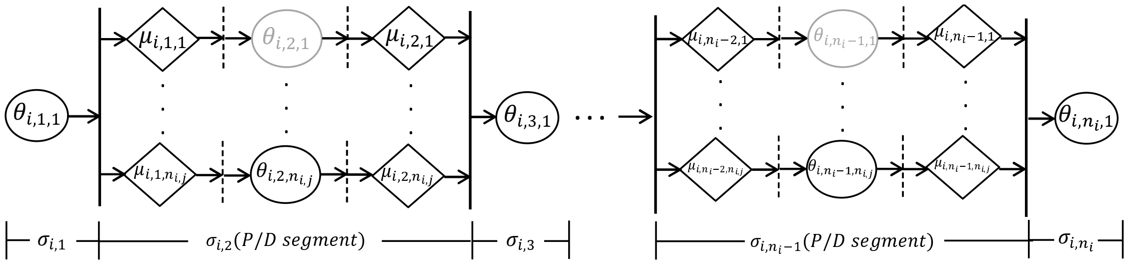

Figure 4.1: Parallel execution length.

P/D threads and their messages can vary between different P/D segments. Therefore, $P_{i,2j}$ is the WCET of a single P/D thread within a segment $\sigma_{i,2j}$ (since all k P/D threads on the same P/D segment have exactly the same WCET). Also, P/D threads and messages, share the same period T_i.

For notational convenience some definitions that simplify the explanation and analysis of the proposed algorithms are introduced.

The parallel execution length refers to the execution time that is required to execute all P/D threads if all P/D threads are executed in parallel. Since all P/D threads have the same WCET (denoted as $P_{i,2j}$) only one the WCET of P/D thread $P_{i,2j}$ has to be considered for each P/D segment $\sigma_{i,2j}$. Figure 4.1 shows the threads that contribute to the parallel execution length. Thus, the *parallel execution length* of a P/D task τ_i is denoted as P_i and defined as:

Definition 4.1. ***(Parallel execution length).*** *The parallel execution length P_i is the sum of all WCET of all P/D threads within the master thread:*

$$P_i = \sum_{j=1}^{\frac{n_i-1}{2}} P_{i,2j}. \tag{4.1}$$

The minimum execution length η_i, represents the minimum execution time a P/D task τ_i needs to execute, if all P/D threads are executed in parallel. That is, the parallel execution length plus the sum of all threads that execute in the sequential threads (i.e., the ones executing on the invoker processor). Figure 3.7 shows the threads that contribute to the minimum execution length. Please, note that those threads are the same as the ones belonging to the master thread. Thus, the *minimum execution length* of a P/D task τ_i is denoted as η_i and defined as:

Definition 4.2. ***(Minimum execution length).*** *The minimum execution length is equal to the sum of the WCET of all the threads described in the master thread (i.e., the sum of the* *sequential threads plus the parallel execution length):*

$$\eta_i = \left(\sum_{j=0}^{\frac{n_i-1}{2}} C_{i,2j+1} \right) + P_i. \tag{4.2}$$

The maximum execution length η_i represents the maximum execution time a P/D task τ_i needs to execute. That is, if all sequential and P/D threads execute sequentially. Thus, the *maximum execution length* of a P/D task τ_i is denoted as C_i and defined as:

Definition 4.3. ***(Maximum execution length).*** *The maximum execution length C_i represents the maximum execution time a P/D task τ_i needs to execute when all P/D threads are executed sequentially on the invoker processor. This is equal to the sum of WCET of all threads in a task τ_i:*

$$C_i = \left(\sum_{j=0}^{\frac{n_i-1}{2}} C_{i,2j+1} \right) + P_i \times m_i. \tag{4.3}$$

Please note, that the messages $\mu_{i,j,k}$ are not considered in Eq. (4.2) or Eq. (4.3), since all inter-process communications are internal to the invoker processor. The synchronization cost between the sequential and P/D threads can therefore be considered negligible. Figure 3.8 shows an example in which some messages ($\mu_{i,1,1}$ and $\mu_{i,2,1}$) are transmitting within the same local processor, therefore, its cost is negligible.

The *slack time* of a task τ_i is denoted as L_i and defined as:

Definition 4.4. ***(Slack time).*** *The positive slack time L_i is the temporal difference between the task deadline D_i and the minimum execution length η_i:*

$$L_i = D_i - \eta_i. \tag{4.4}$$

If the slack L_i is a negative number, it means that η_i is larger than its deadline ($T_i = D_i$). Therefore, such a task is not schedulable on any number of processors with a speed of 1.

The P/D-DMS algorithm tries to coalesce as many threads as possible into the master threads. The number of possible threads to coalesce is given by the *task capacity*. Thus, the *task capacity* of a P/D task τ_i is denoted as f_i and defined as:

Definition 4.5. ***(Task Capacity).*** *The task capacity f_i is defined as the capacity of the master thread of a task τ_i to execute extra P/D threads from all P/D segments without missing its deadline:*

$$f_i = \frac{L_i}{P_i}. \tag{4.5}$$

4.2 The Distributed Stretch Transformation Model

The Distributed Stretch Transformation (DST) has been inspired by the Task Stretch Transformation (Lakshmanan et al., 2010) (TST) model and the Segment Stretch Transformation (Fauberteau et al., 2011; Qamhieh et al., 2011) (SST) model. The DST model is designed specifically for distributed systems, in which real-time tasks and messages need to be processed and transmitted by processors and a real-time network, respectively. Therefore, the main difference from DST, when compared with TST and SST, is that the two previous transformation algorithms were conceived for multi-core processors, thus not considering the transmission delays inherent to the synchronization between threads executing on different processors, as in the case of distributed systems.

The TST and SST transformations consider that tasks are scheduled by a partitioned preemptive fixed-priority algorithm, executed in a multi-core processor. In this model, it is also considered that tasks are scheduled with the preemptive fixed priority Deadline Monotonic (DM) algorithm on each processor. However, messages to be transmitted within the real-time network are scheduled with a non-preemptive version of the DM algorithm. This is because the transmission of a message cannot be interrupted once initiated.

4.2.1 The Task Stretch Transformation and Segment Stretch Transformation Models

In this subsection, the TST and the SST transformation models are studied, with the intention of showing the similarities and main differences with the DST model.

In the TST model (Lakshmanan et al., 2010), Lakshmanan *et al.* show that “F-J task sets on multiprocessor systems can have schedulable utilization bounds slightly greater than and arbitrarily close to uniprocessor schedulable utilization bound”, thus, it is desirable to avoid fork-join structures as much as possible. The main objective of the TST model is to convert the master thread into a fully stretched string in which the execution length of the master thread becomes equal to its period T_i. The transformation is done by inserting (or coalescing) threads (or part of them) into the master thread while paying attention to respect their precedence constraints. Thus, a subset of parallel threads executes with the master thread while the rest of them are partitioned among the cores using the partitioning heuristic Fisher-Baruah-Baker First-Fit-Decreasing (FBB-FFD) (Fisher et al., 2006). The authors showed that their scheduling algorithm has a resource augmentation bound of 3.42.

The main disadvantage of the TST is that it forces to stretch a master thread completely. In some cases, it may not be possible to fit complete threads within the master thread. This provokes a migration of the remaining part of such a thread for being executed in another processor. For this reason, the authors of (Fauberteau et al., 2011) proposed the SST model, which also tries to convert the parallel threads into sequential ones by creating a master thread, but with the difference that the coalescing operation is performed only when parallel threads can be fully inserted within the master thread. Thus, creating a master thread that can be fully stretched (with a WCET of the master thread equal to its period) or partially stretched (the WCET of the master thread is smaller or equal to its period). In a similar manner than in (Lakshmanan et al., 2010), the remaining parallel threads are scheduled with the partitioned scheduling algorithm FBB-FFD (Fisher et al., 2006). Later, the same authors (Qamhieh et al., 2011) proved that SST has the same resource augmentation bound of 3.42 than TST, although, it cannot be claimed that one of both algorithms dominates the other (Fauberteau et al., 2011).

4.2.2 The Distributed Stretch Transformation (DST) Algorithm

This work is inspired by the SST approach. Since “F-J task sets on multiprocessor systems can have schedulable utilization bounds slightly greater than and arbitrarily close to uni-processor schedulable utilization bound”, it is opted for the formation of a stretched master thread (denoted as $\tau_i^{stretched}$) for each P/D task τ_i.

However, some specific constraints that are related to distributed systems have to be addressed. In that case, when performing a D-Fork operation, it implies that some messages will be transmitted within the network that may affect the execution length of the P/D tasks.

Let the DST transformation be illustrated with the following example.

Example 4.1. *Consider two tasks: τ_1 with a single P/D segment. τ_i is composed by two sequential threads $\theta_{1,1,1}$ and $\theta_{1,3,1}$ with a WCET of* 1 *time unit; there P/D threads $\theta_{1,2,1-3}$ with a WCET of* 2 *time units, and their respective messages $\mu_{1,1,1-3}$ and $\mu_{1,2,1-3}$ with a WCML of* 1 *time unit. The period T_i is equal to* 8 *time units. Similarly, τ_2 is composed by two sequential threads $\theta_{2,1,1}$ and $\theta_{2,3,1}$ with a WCET of* 1 *time unit; there P/D threads $\theta_{2,2,1-3}$ with a WCET of* 3 *time units, and their respective messages $\mu_{1,1,1-3}$ and $\mu_{1,2,1-3}$ with a WCML of* 1 *time unit. The period T_i is equal to* 10 *time units. Tasks τ_1 and τ_2 are to be scheduled on* 3 *distributed processors interconnected with a real-time network. Figure 4.2a, shows the execution of a task to be scheduled under global DM scheduling. It is possible to see that task τ_2 having the lowest priority misses its deadline at time 10. This is due to the suffered interference provoked by threads of task τ_1 that have higher priority. Also, notice the presence of a high source of interference in the network, for example, the P/D thread $\theta_{2,2,3}$ with WCET $P_{2,2,3}$, is ready for execution at time* 1, *but due to the network interference it is only released for execution in processor* 3 *at time* 7, *therefore drastically increasing its response time.*

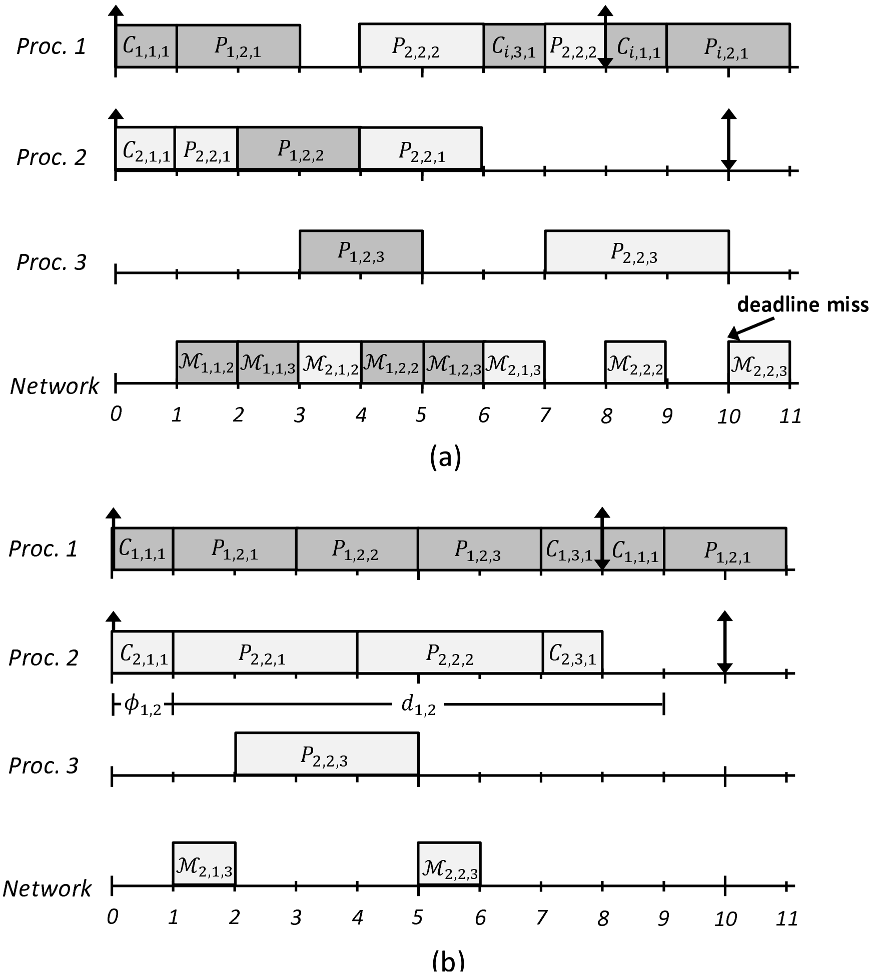

Figure 4.2: P/D tasks: (a) scheduled with global scheduling, (b) scheduled after the DST transformation.

Now consider the DST transformation explained below and illustrated in Figure 4.2b. By calculating the maximum execution length of tasks τ_1 and τ_2 (see Definition 4.3), it results that $C_1 = 8$ and $C_2 = 11$. Then, by looking at Figure 4.2b it is possible to observe two cases:

i. $C_i \leq T_i$. This is the case of τ_1 in this example; whenever such a case appears for a task τ_i, the task τ_i is fully stretched into a master thread and handled as a sequential task with execution time equal to C_i, a task period of T_i, and an implicit deadline equal to D_i. That is, all threads of the tasks are executed sequentially on a single processor.

ii. $C_i > T_i$. This is this case of τ_2 in this example; for such tasks, the DST transformation inserts (coalesces) as many P/D threads of τ_i into the master thread as possible. To do so, it is needed to calculate the available slack and capacity of task τ_i as indicated in Eq. (4.4) and Eq. (4.5). For τ_2, it gives $L_2 = 10 - 5 = 5$ and, $f_2 = \frac{5}{3}$. Thus, the number of P/D threads that each P/D segment can fully insert into the master thread without causing τ_i to miss its deadline is given by:

$$i_{i,2j} = \lfloor f_i \rfloor . \tag{4.6}$$

For example, in the case of τ_2, $i_{2,2} = \lfloor f_2 \rfloor = 1$. It can indeed be seen on Figure 4.2b that τ_2 executes two P/D threads per P/D segment on the invoker processor (one from the master thread and the inserted one) rather than only one when considering the non-stretched master thread.

In the DST only P/D threads that fit completely (since $i_{i,2j} = \lfloor f_i \rfloor$) can be inserted into the master thread. A master thread is assigned to be executed in its own processor and the remaining subset of P/D threads, have to be executed on other nodes in the system. The partitioning of the remaining P/D threads to the processors is done according to the FBB-FFD algorithm (Fisher et al., 2006).

The number $q_{i,2j}$ of the remaining P/D threads that have not been coalesced into the master thread is given by:

$$q_{i,2j} = m_i - i_{i,2j}. \tag{4.7}$$

The capacity f_i of task τ_i is equally distributed between all the P/D segments of a P/D task τ_i. This distribution can be considered as the available scheduling length for the execution of threads and transmission of messages in each P/D segment on a remote processor.

Thus, the maximum scheduling length for the subset of P/D threads and their respective messages is determined by defining a set of P/D intermediate deadlines $d_{i,2j}$:

$$d_{i,2j} = (f_i + 1) \times P_{i,2j} \; \forall 1 \leq j \leq \frac{n_i - 1}{2}. \tag{4.8}$$

In the case of task τ_2, $d_{2,2} = 3 \times (\frac{5}{3} + 1) = 8$. Also, each P/D segment $\sigma_{i,2j}$ has a static offset $\phi_{i,2j}$ defined as:

$$\phi_{i,2j} = \sum_{j=0}^{\frac{n_i-1}{2}} C_{1,2j+1,1} + \sum_{j=1}^{\frac{n_i-1}{2}} d_{i,j}. \tag{4.9}$$

Thus, at the end of the DST transformation, a P/D task τ_i will be composed of a single stretched master thread $\tau_i^{stretched}$ and a set of constrained deadline P/D threads $\{\tau_{i,j,k}^{cd}\}$ (and their respective constrained deadline messages $\{\mu_{i,j,k}^{cd}\}$) per each P/D segment $\sigma_{i,2j}$.

The P/D segments offsets $\phi_{i,2j}$ and the P/D segments deadlines $d_{i,2j}$, define the scheduling window, in which the remaining $q_{i,2j}$ P/D threads (and its corresponding messages) have to complete their execution (and transmission, respectively) in order for a task τ_i to respect its deadline. That is, the following inequality must be respected:

$$r_{\mu_{i,2j-1,k}} + r_{\theta_{i,2j,k}} + r_{\mu_{i,2j,k}} \leq d_{i,2j} \; \forall \theta_{i,2j,k} \notin masterthread, \tag{4.10}$$

where, $r_{\mu_{i,2j-1,k}}$, $r_{\mu_{i,2j,k}}$ and $r_{\theta_{i,2j,k}}$ are the Worst Case Response Time (WCRT) of messages $\mu_{i,2j-1,k}$ and $\mu_{i,2j,k}$, and the thread $\theta_{i,2j,k}$, respectively.

4.2.3 End-to-end Delay Computation in Distributed Systems

In this section some results for the calculation of the response time for the execution and transmission of threads and messages respectively, are summarised.

It is known from (Joseph and Pandya, 1986) that for periodic fixed priority preemptive tasks, the following recursive equation can be used to calculate the response time of a threads $\theta_{i,j,k}$:

$$r_{\theta_{i,j,k}}^{n+1} = C_{i,j,k} + \sum_{\theta_{i,j,l} \in hp(\theta_{i,j,k})} \left\lceil \frac{r_{\theta_{i,j,k}}^{n}}{T_{i,j,l}} \right\rceil C_{i,j,l}, \tag{4.11}$$

where $r_{\theta_{i,j,k}}$ is the Worst Case Response Time (WCRT) of a thread $\theta_{i,j,k}$ and $hp(r_{\theta_{i,j,k}})$ is the set of all threads $\theta_{i,j,l}$ with higher priority than $\theta_{i,j,k}$ that execute on the same processor π_i.

The recursion ends when $r_{\theta_{i,j,k}^{n+1}} = r_{\theta_{i,j,k}^{n}} = r_{\theta_{i,j,k}}$ and can be solved by successive iterations starting from $r_{\theta_{i,j,k}}^{1} = C_{i,j,k}$. The series is non-decreasing, and therefore converges if

$\sum_{\theta_{i,j,l} \in hp(\theta_{i,j,k}) \cup \theta_{i,j,k}} \frac{C_{i,j,l}}{T_{i,j,l}} \leq 1$. If the condition of convergence is not respected threads $\theta_{i,j,k}$ are not schedulable.

For the case of messages $\mu_{i,j,k}$, the calculation of the WCRT needs to consider the non-preemptability of messages on the network. Thus, for periodic fixed-priority non-preemptive messages, the following recursive equation can be used to calculate the worst-case response time (George et al., 1996):

$$r^{n+1}_{\mu_{i,j,k}} = M_{i,j,k} + \sum_{\mu_{i,j,l} \in hp(\mu_{i,j,k})} \left\lceil \frac{r^{n}_{\mu_{i,j,k}}}{T_{i,j,l}} \right\rceil M_{i,j,l} + \max_{\mu_{i,j,l} \in lp(\mu_{i,j,k})} \{\mu_{i,j,l}\}, \qquad (4.12)$$

where, the third term on the right hand side of (4.12), accounts for the maximum possible suffered interference of a higher priority message $\mu_{i,j,k}$, caused by lower priority message $\mu_{i,j,l}$, contained in the set of lower priority messages $lp(\mu_{i,j,k})$.

4.3 The P/D-DMS Algorithm

The P/D-DMS algorithm is the partitioning algorithm for partitioning the set τ of tasks τ_i onto the elements of the distributed system. The P/D-DMS algorithm realizes the partitioning by:

i. applying the DST to each P/D tasks τ_i in τ. Two possible cases can appear (see Section 4.2):

   a. $C_i \leq T_i$; the task is fully stretched in a single sequential thread and added to a list L, or

   b. the task τ_i is converted into a master thread τ_i^{master} and a subset of sequential P/D threads $\{\tau^{cd}_{i,j,k}\}$ with their respective messages $\{\mu^{cd}_{i,j,k}\}$. The master thread τ_i^{master} is allocated to its own processor and the subset of sequential P/D threads is added to the list L, and

ii. the set of threads in L, are partitioned onto processors according to the FBB-FFD algorithm (Fisher et al., 2006). Messages $\mu^{cd}_{i,j,k}$ are assigned to the single real-time network, accordingly.

In the following subsection, the demand bound function of a P/D task τ_i is analysed and the resource augmentation bound for the P/D-DMS algorithm is provided.

4.3.1 Demand Bound Function

Definition 4.6. ***(Demand Bound Function (DBF)(Baruah et al., 1990)).*** *The DBF is defined as the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t.*

For a sequential task τ_i with a total execution time of C_i, period T_i, and a deadline $D_i \leq T_i$, the DBF function is given by:

$$DBF\left(\tau_i, t\right) = \max\left\{0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right\}. \tag{4.13}$$

Theorem 4.1. *The DBF function of a stretched task $\tau_i^{stretched}$ that has been transformed by the DST algorithm is bounded from above by:*

$$DBF\left(\tau_i^{stretched}, t\right) \leq \max_j \left\{ \frac{C_i}{Ti - \eta_i - \frac{\left(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}\right) \times P_i}{P_{i,2j}}} \right\} t. \tag{4.14}$$

Proof. the concept of DBF is generalised for the case of P/D tasks τ_i composed of a master thread τ_i^{master} and a sequence of sequential P/D threads $\{\tau_i^{cd}\}$ and their respective messages $\{\mu_i^{cd}\}$. The two only possible cases when applying the DST algorithm to a P/D task τ_i are considered (see Section 4.2):

i. **Case $C_i \leq T_i$.** In that case, a P/D task τ_i is fully stretched after applying the DST into a single sequential thread with a total execution time of $C_i^{master} \leq C_i$, period T_i, and a deadline $D_i^{master} = T_i$, therefore, the DBF function (Definition 4.6) can be used without any modifications:

$$\begin{aligned}
DBF\left(\tau_i^{stretched}, t\right) &= DBF\left(\tau_i^{master}, t\right) \\
DBF\left(\tau_i^{stretched}, t\right) &= \max\left\{0, \left(\left\lfloor \frac{t - D_i^{master}}{T_i} \right\rfloor + 1\right) C_i^{master}\right\} \\
DBF\left(\tau_i^{stretched}, t\right) &= \max\left\{0, \left(\left\lfloor \frac{t}{T_i} \right\rfloor\right) C_i^{master}\right\} \leq \frac{C_i}{T_i} t \leq \frac{C_i}{T_i - \eta_i} t \\
&\leq \max_j \left\{ \frac{C_i}{Ti - \eta_i - \frac{\left(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}\right) \times P_i}{P_{i,2j}}} \right\} t,
\end{aligned} \tag{4.15}$$

where $0 \leq \eta_i \leq T_i$.

ii. **Case $Ci > T_i$.** In the second case, after applying the DST, a P/D task τ_i has been transformed into a master thread τ_i^{master}, and a set $\{\tau_{i,j,k}^{cd}\}$ of constrained deadline P/D threads associated to their respective constrained deadline messages $\{\mu_{i,j,k}^{cd}\}$. That is:

$$\tau_i^{stretched} = \tau_i^{master} + \{\tau_{i,j,k}^{cd}\}.$$

Thus, the DBF function can be computed as follows:

$$DBF\left(\tau_i^{stretched}, t\right) = DBF\left(\tau_i^{master}, t\right) + DBF\left(\{\tau_{i,j,k}^{cd}\}, t\right). \tag{4.16}$$

Since the master thread has been stretched, it results that:

$$\begin{aligned} \eta_i + P_i \lfloor f_i \rfloor \leq & C_i^{master} \leq T_i \\ & C_i^{master} \leq T_i \Rightarrow \frac{C_i^{master}}{T_i} < 1 \\ & \Longrightarrow DBF\left(\tau_i^{master}, t\right) = \max\left\{0, \left(\left\lfloor \frac{t}{T_i} \right\rfloor\right) C_i^{master}\right\} \\ & \leq \frac{C_i^{master}}{T_i} t \leq t. \end{aligned} \tag{4.17}$$

The set $\{\tau_{i,j,k}^{cd}\}$ of constrained deadline P/D threads and their respective messages $\{\mu_{i,j,k}^{cd}\}$ consist of the P/D threads and P/D messages of all P/D segments of a task τ_i. Since P/D segments within a P/D tasks have an offset, only one P/D segment can be activated at time t and the maximum number of P/D threads in each P/D region is equal to $(q_{i,2j} - 1)$ where $q_{i,2j} = m_i - \lfloor f_i \rfloor$.

Therefore, the previous property guarantees that the DBF of the subset of P/D threads $\{\tau_{i,j,k}^{cd}\}$ over any interval of length t, does not exceed $\delta_i^{\max}(q_{i,2j} - 1)t$:

$$DBF\left(\{\tau_{i,j,k}^{cd}\}, t\right) \leq \delta_i^{\max}(q_{i,2j} - 1)t. \tag{4.18}$$

The density of a constrained deadline task is given by:

$$\delta_i = \frac{C_i}{D_i}.$$

The DST transformation fills the available slack L_i with $\lfloor f_i \rfloor$ P/D threads per P/D segment (remember that only complete P/D threads are inserted within the master thread, since $\lfloor f_i \rfloor$ is an integer number). In each P/D segment within τ_i, all P/D threads have the same WCET $P_{i,2j}$, and a deadline $d_{i,2j} = P_{i,2j} \times (f_i + 1)$. Due to the fact that the P/D thread is executed on a remote processor in the system, two messages per P/D thread ($\mu_{i,2j-1,k}$ and $\mu_{i,2j,k}$) are sent through the real-time network. Thus, in the worst-case the time for a P/D thread to execute is reduced to: $P_{i,2j} \times (f_i + 1) - r_{\mu_{i,2j-1,k}} - r_{\mu_{i,2j,k}}$) (see Eq.(4.8) and Eq.(4.12)).

Therefore, the maximum density of the P/D threads $\{\tau^{cd}_{i,,k}\}$ can be calculated as follows:

$$\begin{aligned}
\delta_i^{\max} &= \max_j \left\{ \frac{P_{i,2j}}{P_{i,2j} \times (f_i+1) - r_{\mu_{i,2j-1,k}} - r_{\mu_{i,2j,k}}} \right\} \\
&= \max_j \left\{ \frac{1}{(f_i+1) - \frac{r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}}{P_{i,2j}}} \right\}.
\end{aligned} \tag{4.19}$$

By substituting Eq. (4.19) in Eq. (4.18), the DBF of the P/D threads $\{\tau^{cd}_{i,j,k}\}$ can be calculated as:

$$DBF\left(\{\tau^{cd}_{i,j,k}\}, t\right) \leq \max_j \left\{ \frac{1}{(f_i+1) - \frac{r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}}{P_{i,2j}}} \right\} (q_{i,2j} - 1)t, \tag{4.20}$$

and since $q_{i,2j} = m_i - \lfloor f_i \rfloor$, we get:

$$\begin{aligned}
DBF\left(\{\tau^{cd}_{i,j,k}\}, t\right) &\leq \max_j \left\{ \frac{m_i - \lfloor f_i \rfloor - 1}{(f_i+1) - \frac{r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}}{P_{i,2j}}} \right\} t \\
&\leq \max_j \left\{ \frac{m_i - \lfloor f_i \rfloor - 1}{f_i - \frac{r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}}{P_{i,2j}}} \right\} t.
\end{aligned} \tag{4.21}$$

By substituting inequality (4.17) and inequality (4.21) in Eq. (4.16), it is possible to compute the DBF of $\tau_i^{stretched}$ as:

$$\begin{aligned} DBF\left(\tau_i^{stretched}, t\right) &\leq t + \max_j \left\{ \frac{m_i - \lfloor f_i \rfloor - 1}{f_i - \frac{r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}}{P_{i,2j}}} \right\} t \\ &\leq \max_j \left\{ 1 + \frac{m_i - \lfloor f_i \rfloor - 1}{f_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})}{P_{i,2j}}} \right\} t \\ &\leq \max_j \left\{ \frac{f_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})}{P_{i,2j}} + m_i - \lfloor f_i \rfloor - 1}{f_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})}{P_{i,2j}}} \right\} t \\ &\leq \max_j \left\{ \frac{m_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})}{P_{i,2j}}}{f_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})}{P_{i,2j}}} \right\} t, \end{aligned} \tag{4.22}$$

because $f_i = \frac{T_i - \eta_i}{P_i}$ and $m_i \times P_i < C_i$ (from Eq.(4.3)), it results that:

$$\begin{aligned} DBF\left(\tau_i^{stretched}, t\right) &\leq \max_j \left\{ \frac{m_i \times P_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}) \times P_i}{P_{i,2j}}}{T_i - \eta_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}) \times P_i}{P_{i,2j}}} \right\} t \\ &\leq \max_j \left\{ \frac{C_i}{T_i - \eta_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}}) \times P_i}{P_{i,2j}}} \right\} t. \end{aligned} \tag{4.23}$$

Then, in the two possible cases, the DBF of a task $\tau_i^{stretched}$ resulting of the application the DST transformation is bounded by the same value (Eq. (4.15) and Eq. (4.23)). □

4.3.2 Resource Augmentation Bound

This section presents the *resource augmentation bound* of the P/D-DMS algorithm. The resource augmentation bound of the P/D-DMS is equal 4, this implies that any task set that is feasible on m unit-speed processors, can be scheduled by the P/D-DMS algorithm on *m* processors and a real-time network that are 4 times faster.

The results of Theorem 4.2 from (Fisher et al., 2006) are re-used.

Theorem 4.2. *(Fisher et al., 2006). Any constrained sporadic task system τ is successfully schedulable by FBB-FFD on m unit-capacity processors if:*

$$m \geq \frac{\delta_{sum} + u_{sum} - \delta_{\max}}{1 - \delta_{\max}}, \tag{4.24}$$

where,

$$\delta_{sum} = \max_{t>0} \left\{ \frac{\sum_{i=1}^{n} DBF\left(\tau_i^{stretched}, t\right)}{t} \right\}. \tag{4.25}$$

Using Eq. (4.24) and Eq. (4.25), it is possible to provide a resource augmentation bound for the Distributed-DMS partition algorithm.

Theorem 4.3. *If any set τ of P/D tasks τ_i is feasible on m unit-speed processors (and messages are feasible on a single unit-speed real-time network), then the Distributed-DMS partition algorithm is guaranteed to successfully schedule this task set on m processors and one real-time network that are* 4 *times faster.*

Proof. The set τ of P/D tasks is feasible on m unit-speed processors:

$$u_{sum} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq m, \tag{4.26}$$

and because the minimum response time of a thread is its execution time, Eq. (4.8) and Eq. (4.9) imply that the task set τ is feasible if and only if:

$$\begin{aligned} & r_{\mu_{i,2j-1,k}} + P_{i,2j} + r_{\mu_{i,2j,k}} \leq (f_i + 1) \times P_{i,2j} \\ & \Leftrightarrow r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}} \leq f_i \times P_{i,2j}. \end{aligned} \tag{4.27}$$

Consider the *minimum execution length* η_i of any task τ_i. It must respect that:

$$\forall 1 \leq i \leq n \quad \eta_i \leq T_i. \tag{4.28}$$

Otherwise, τ_i would be unschedulable on a unit-speed processor. On a processor that is v times faster, the minimum execution length η_i^v is given by:

$$\forall 1 \leq i \leq n \quad \eta_i^v = \frac{\eta_i}{v} \leq \frac{T_i}{v}. \tag{4.29}$$

For each task τ_i, it was proven in Theorem 4.1 that:

$$DBF\left(\tau_i^{stretched}, t\right) \leq \max_j \left\{ \frac{C_i}{T_i - \eta_i - \frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})P_i}{P_{i,2j}}} \right\} t. \tag{4.30}$$

Using the above inequality together with Eq. (4.25) we have:

$$\delta^v_{sum} \le \sum_{i=1}^{n} \max_j \left\{ \frac{C_i^v}{T_i - \eta_i^v - \frac{(r_{\mu^v_{i,2j-1,k}} + r_{\mu^v_{i,2j,k}})P_i^v}{P^v_{i,2j}}} \right\}, \tag{4.31}$$

using inequality (4.29):

$$\begin{aligned} \delta^v_{sum} &\le \sum_{i=1}^{n} \max_j \left\{ \frac{C_i^v}{T_i\left(1-\frac{1}{v}\right) - \frac{(r_{\mu^v_{i,2j-1,k}} + r_{\mu^v_{i,2j,k}})P_i^v}{P^v_{i,2j}}} \right\} \\ &\le \frac{1}{v}\sum_{i=1}^{n} \max_j \left\{ \frac{C_i}{T_i\left(1-\frac{1}{v}\right) - \frac{1}{v}\frac{(r_{\mu_{i,2j-1,k}} + r_{\mu_{i,2j,k}})P_i}{P_{i,2j}}} \right\}. \end{aligned} \tag{4.32}$$

From inequality (4.27) and Eq. (4.5):

$$\begin{aligned} \delta^v_{sum} &\le \frac{1}{v}\sum_{i=1}^{n} \frac{C_i}{T_i\left(1-\frac{1}{v}\right) - \frac{1}{v} f_i P_i} \\ &\le \frac{1}{v}\sum_{i=1}^{n} \frac{C_i}{T_i\left(1-\frac{1}{v}\right) - \frac{T_i}{v}} \\ &\le \frac{1}{v-2}\sum_{i=1}^{n} \frac{C_i}{T_i} \\ \Leftrightarrow \delta^v_{sum} &\le \frac{1}{v-2} u_{sum}. \end{aligned} \tag{4.33}$$

Also on v speed processors, $u^v_{sum} = \frac{u_{sum}}{v}$ and $\delta^v_{\max} = \frac{\delta_{\max}}{v}$. Using Eq. (4.25), the task set τ is schedulable on m processors of speed v if:

$$\begin{aligned} m &\ge \frac{\delta^v_{sum} + u^v_{sum} - \delta^v_{\max}}{1 - \delta^v_{\max}} \\ &\ge \frac{\frac{u_{sum}}{v-2} + \frac{u_{sum}}{v} - \frac{\delta_{\max}}{v}}{1 - \frac{\delta_{\max}}{v}}. \end{aligned}$$

The right-hand side of the inequality above is an increasing function of δ_{max} for $m \ge \frac{v(v-2)}{2v-2}$.

Since $\delta_i = \frac{C_i}{D_i}$ and because the task set τ is feasible if and only if $C_i \le D_i$ for all tasks τ_i, the greatest possible density for a feasible task set is given by $\delta_i^{max} \le 1$.

Thus, when $m \geq \frac{v(v-2)}{2v-2}$, the schedulability is guaranteed if:

$$\begin{aligned} m &\geq \frac{\frac{m}{v-2} + \frac{m}{v} - \frac{1}{v}}{1 - \frac{1}{v}} \\ m\left(1 - \frac{1}{v}\right) &\geq \frac{m}{v-2} + \frac{m}{v} - \frac{1}{v} \\ v - \frac{2}{v-2} &\geq 3 - \frac{1}{m}. \end{aligned} \tag{4.34}$$

This inequality is respected with $v = 4$ and $m \geq 2$. Hence, any feasible P/D task set τ feasible on $m \geq 2$ unit-speed processors and a unit-speed network, is guaranteed to be schedulable by the P/D-DMS algorithm on m processors and a single real-time network with speed $v = 4$. □

4.4 Evaluation of the P/D-DMS Algorithm

This section presents the simulation results that validate the resource augmentation bound of the P/D-DMS algorithm after applying the DST transformation presented in Section 4.3 and Section 4.2, respectively.

To generate feasible P/D task sets, the guidelines presented in (Emberson et al., 2010) for generating random task sets for multiprocessor systems, using the Stafford's Randfixedsum algorithm (Stafford, 2004) had been followed. The Randfixedsum algorithm generates a set of n values which are evenly distributed and whose components sum to a constant value. Thus, the Randfixedsum algorithm is used for generating unbiased sets of P/D tasks with a fixed total density $\delta_{tot} = \sum \delta_i$. For a given total density δ_{tot}, the Randfixedsum algorithm returns n P/D tasks with density δ_{tot}. For generating the P/D threads densities the Randfixedsum algorithm is used again taking as an input the previous generated densities $\delta_i = \sum \delta_{i,j,k}$, obtaining a set of values $\delta_{i,j,k}$ for each P/D thread. The WCETs and end-to-end deadlines D_i are also generated as recommended in (Emberson et al., 2010). Once all P/D threads are generated, their respective messages are generated and inserted within a P/D task by preserving their execution order. The total message density δ_{tot}^{msg}, represents the utilization of the network. Thus, when a total message density is given, the Randfixedsum algorithm returns n messages with a density of δ_i^{msg} for each task τ_i. For generating the messages densities $\delta_i^{msg} = \sum \delta_{i,j,k}^{msg}$ the Randfixedsum algorithm is used again taking as an input the previous generated densities δ_i^{msg}. It is considered that applications have implicit end-to-end deadlines ($D_i = T_i$) following a uniform distribution between the values $D_i^{\min} = 100$ and $D_i^{\max} = 10000$.

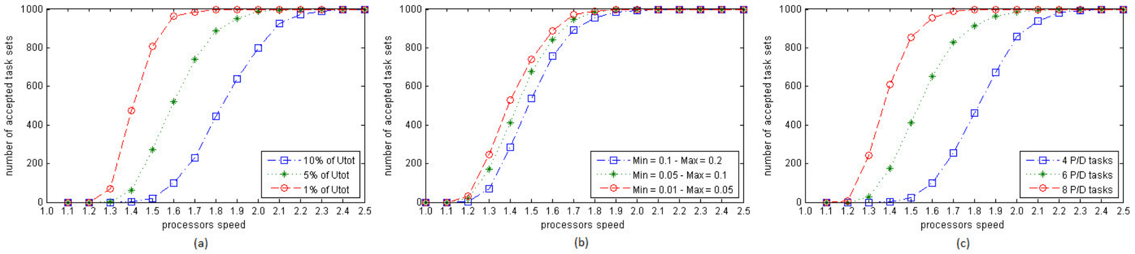

Figure 4.3: 1000 generated task sets varying (a) the total message density δ_{tot}^{msg}, (b) the minimum thread density $\delta_{i,j,k}^{\min}$ and maximum thread density $\delta_{i,j,k}^{\max}$, and (c) the number of P/D tasks in the set τ

Figure 4.3a shows the number of accepted task sets over 1000 experiments for different given total message densities δ_{tot}^{msg}. The simulation considers 4 P/D tasks that are partitioned by the P/D-DMS algorithm in a computing platform of 8 processors and 1 network. Thus the total utilization U_{tot} for these experiments is fixed to 8. Three different total message densities are analyzed:

i. δ_{tot}^{msg} equal to the 10% of the fixed total utilization U_{tot} ; $\delta_{tot}^{msg} = 0.8$,

ii. δ_{tot}^{msg} equal to the 5% of U_{tot}; $\delta_{tot}^{msg} = 0.4$, and

iii. δ_{tot}^{msg} equal to the 1% of the total utilization; $\delta_{tot}^{msg} = 0.08$.

It is possible to see that when δ_{tot}^{msg} increases, more speed v is required by the processors and the network to be able to schedule 100% of the task sets. This effect is modelled by Eq. 4.10.

In Figure 4.3b the number of accepted task sets for 1000 experiments is shown. 4 P/D tasks have to execute in a computing platform composed of 1 network and 8 distributed processors. The total density δ_{tot} is fixed to 8. In this case the variations in respect of different individual thread density $\delta_{i,j,k}^{\min}$ and $\delta_{i,j,k}^{\max}$ are analysed. Three different variations are compared:

i. $\delta_{i,j,k}^{\min} = 0.1$ and $\delta_{i,j,k}^{\max} = 0.2$,

ii. $\delta_{i,j,k}^{min} = 0.05$ and $\delta_{i,j,k}^{\max} = 0.1$, and

iii. $\delta_{i,j,k}^{\min} = 0.01$ and $\delta_{i,j,k}^{\max} = 0.05$. δ_{tot}^{msg} is fixed to 5% of the δ_{tot}.

It is possible to observe that if densities of tasks are larger, the more speed is needed to successfully schedule 100% of the task sets.

Figure 4.3c shows the number of accepted task sets over 1000 experiments, in which the number of P/D tasks is varied with a fixed total density $U_{tot} = \delta_{tot} = 8$ to be scheduled in a computing platform of 8 processors and 1 network. The total message density δ_{tot}^{msg} is fixed to 5% of U_{tot}. We compared three possible variations:

i. 4 P/D tasks,

ii. 6 P/D tasks, and

iii. 8 P/D tasks.

It is possible to see that when generating fewer P/D tasks for the same density δ_{tot}, the more speed v is required by the processors and the network to successfully schedule 100% of the task sets. Thus, whenever P/D densities δ_i increase, the probability of finding a schedulable partitioning with the P/D-DMS algorithm for P/D tasks, decreases.

Therefore, it is possible to observe through Figures 4.3(a-c) that in all cases the P/D-DMS algorithm is able to find a schedulable partition by respecting its resource augmentation bound of 4.

4.5 Summary

This chapter presented the P/D-DMS algorithm. The P/D-DMS algorithm makes use of the DST model for scheduling parallel/distributed fixed-priority fork-join real-time tasks. The P/D-DMS algorithm is shown to have a resource augmentation bound of 4. The DST is designed with two main objectives. The first one is to eliminate as many messages of a P/D task as possible by stretching a master thread, since a master thread is executed locally on its own processor. And the second objective is to reduce the possible interference in the network and in the processors by forcing P/D threads to execute within the master thread.

A limitation of the algorithm presented in this chapter is the need to assume that in a task τ_i, every P/D thread $\theta_{i,2j,k}$ and their respective messages $\mu_{i,j,k}$ within a P/D segment $\sigma_{i,2j}$, have identical WCETs $P_{i,2j,k}$ and identical WCMLs $M_{i,j,k}$, respectively.

The following publication is related to the work presented in this chapter:

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, and L.M. Pinho. On the scheduling of fork-join parallel/distributed real-time tasks. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 31–40, June 2014b. doi: 10.1109/SIES.2014.6871184.

Chapter 5

Task Partitioning and Priority Assignment for Sequential Transactional Tasks and P/D Tasks on Hard Real-Time Distributed Systems

5.1 Introduction

The problem of task allocation of sequential transactional tasks (Palencia and Gonzalez Harbour, 1998) and the Parallel/Distributed model (P/D tasks) (Garibay-Martínez et al., 2014b) for distributed systems can be viewed as a two-sided problem:

i. finding the partitioning of tasks and messages onto the processing elements of the distributed system, and;

ii. finding the priority assignment for threads and messages in that partition so that the real-time tasks complete their execution within their deadline.

Those two sub-problems are strongly interrelated as the decision of assigning a thread to a given node should depend on the priorities of the other threads already assigned to that node. Conversely, the priorities of threads executing on a node might need to be adapted if new threads are later added to that node. Therefore, a careful trade-off between the solutions of these two sub-problems needs to be taken in order to obtain an efficient global solution.

Works related to the problem of task partitioning and priority assignment on hard real-time distributed systems are presented in Section 2.4. The work presented in this chapter differs from previous related works in two main aspects:

i. None of the previous works had addressed the allocation of multi-threaded parallel tasks onto elements of a distributed system (with the exception of (Garibay-Martínez et al., 2014b), presented in Chapter 4); and

ii. In works related to sequential tasks and messages, commonly DM is used for assigning priorities, but in this chapter the OPA algorithm is used to assign priorities to tasks and messages. The OPA algorithm is optimal for the case of preemptive fixed-priority tasks with offsets (Audsley, 1991). Furthermore, the OPA is useful for cases in which the deadline of tasks is larger than their periods (e.g. $D > T$) and it is optimal in the sense that if any algorithm can find a schedulable solution OPA can also do it.

This chapter presents the Distributed using Optimal Priority Assignment (DOPA) heuristic (Garibay-Martínez et al., 2013b) that finds a feasible partitioning and priority assignment for distributed tasks based on the linear transactional model. The DOPA heuristic is extended for the assignment of Parallel/Distributed tasks (P/D tasks), therefore a second heuristic called Parallel-DOPA (P-DOPA) is presented. Both DOPA and P-DOPA partition the tasks and messages onto elements of the distributed system, and make use of the Optimal Priority Assignment (OPA) algorithm, known as Audsley's algorithm (Audsley, 1991), to find the priorities of tasks for that partition.

However, the OPA algorithm requires tasks to be independent, therefore, in order to use the OPA algorithm for task sets with dependencies; it is needed to transform them into sets of independent tasks, by imposing artificial intermediate deadlines. Two different methods for adding intermediate deadlines are presented in this chapter; one for linear transactional tasks and one for P/D tasks.

Section 5.2 describes the DOPA heuristic for the linear transactional model which is evaluated through simulations in Section 5.2.3. The P-DOPA heuristic for P/D tasks is described in Section 5.3 and its evaluation is shown in Section 5.3.3. Finally, in Section 5.4 a summary of the chapter is presented.

5.1.1 System Model Adaptations

This chapter considers two different task models:

- the linear transactional model for distributed systems (Palencia and Gonzalez Harbour, 1998); and

- the P/D task model (Garibay-Martínez et al., 2014b).

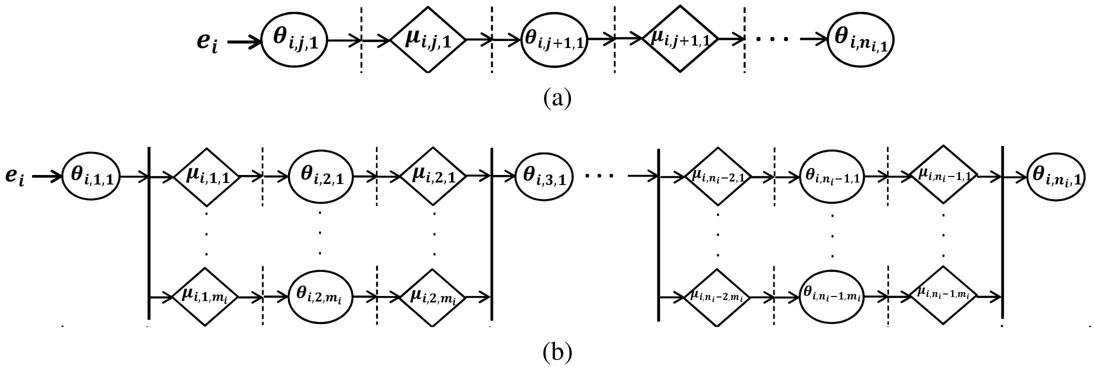

Figure 5.1: (a) Linear transactional task and (b) Parallel/Distributed task (P/D task).

In the linear transactional model, the first thread $\theta_{i,1,1}$ of each application τ_i is activated by an external event e_i with a minimum inter-arrival time T_i. Also, every segment $\sigma_{i,j} \in \tau_i$ consists of a single thread $\theta_{i,j,1}$ (i.e., $n_{i,j} = 1, \forall j$). In that case, whenever a thread $\theta_{i,j,1}$ completes its execution, it sends a message $\mu_{i,j,1}$ to the next segment $\sigma_{i,j+1}$ (consisting of a single thread $\theta_{i,j+1,1}$) and triggers its execution (see Figure 5.1a).

With the P/D tasks model however, threads of different segments $\sigma_{i,j}$ alternatively comprise 1 and m_i threads as introduced in Section 3.4 (see Figure 5.1b). Similarly to Chapter 4, in this chapter it is assumed that all k threads (and messages) belonging to the same parallel segment have the same WCET $C_{i,j,k}$ ($M_{i,j,k}$, resp.).

The density δ_i of a task τ_i is given by $\delta_i = \frac{\sum_{j=1}^{n_i}(C_{i,j,1}+M_{i,j,1})}{D_i}$ and the total density of the system is defined as $\delta_{tot} = \sum_{\tau_i \in \tau} \delta_i$.

In this chapter, the set of nodes π are interconnected with a fixed-priority real-time shared network ϖ.

It is also considered that some threads of a task can be restrained to execute on a specific processor due to design constraints, such as safety reasons or the need to access specific resources (e.g. sensors, actuators, specific instruction sets, etc.) offered by that processor only. Therefore, there exists a set $\mathscr{A} \subseteq \{\cup \ \forall \ \tau_i \in \tau \ \theta_{i,j,k}\}$ of threads that are resource constrained and are statically assigned to their respective processor. Also, there exists a set $\Upsilon = \{\cup \ \forall \ \tau_i \in \tau \ \theta_{i,j,k}\} \setminus \mathscr{A}$ of threads that do not have any resource constraints and can be allocated onto any processor.

Figure 5.2, shows an example of the allocation of two real-time tasks. In Figure 5.2a one sequential transaction and one P/D task that have to be allocated onto the elements of the distributed system shown in Figure 5.2b. The distributed system is composed of 3 processors and 1 real-time network. Threads $\theta_{1,1,1}$ and $\theta_{2,1,1}$ are resource constrained (pre-assigned to processors 1 and 2, respectively), and thus belong to the set $\mathscr{A}$. Also, there exists a list Υ of unallocated threads, which can be allocated to any processor.

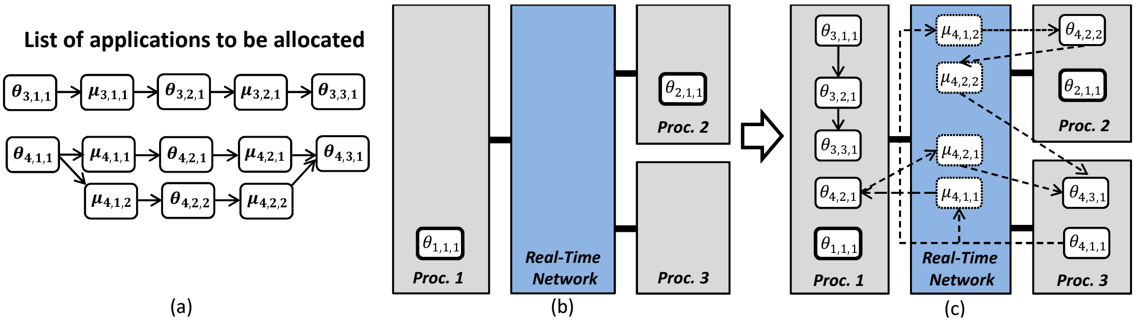

Figure 5.2: Allocation of real-time tasks onto the elements of the distributed system.

An example of the allocation of the threads and messages is shown in Figure 5.2c. By looking at Figure 5.2c, one can notice that threads $\theta_{3,1,1}$, $\theta_{3,2,1}$, and $\theta_{3,3,1}$ are allocated to the same processor, and therefore messages $\mu_{3,1,1}$ and $\mu_{3,2,1}$ can be omitted.

5.2 The Distributed using Optimal Priority Assignment (DOPA) Heuristic

The DOPA heuristic simultaneously addresses the two sub-problems of:

i. finding the partitioning of threads and messages onto the elements of the distributed system, and;

ii. finding the priority assignment for that partition.

In this section, the case of linear transactions (upper task in Figure 5.1a) is considered. The assignment of P/D tasks (lower task in Figure 5.1a) onto elements of the distributed system is treated in Section 5.3.

5.2.1 Optimal Priority Assignment (OPA) Algorithm

Regarding the problem of priority assignment, there are some techniques to assign priorities to a set of preemptive independent threads. DM (Leung and Whitehead, 1982) is the most commonly used in distributed systems. DM is optimal for assigning priorities if there is an instant in the schedule at which all threads release a job simultaneously. However, in distributed systems threads and messages have dependencies on other threads and/or messages of the same task. Because a thread $\theta_{i,j+1,1}$ never starts its execution before the completion of a thread $\theta_{i,j,1}$, then $\theta_{i,j,1}$ and $\theta_{i,j+1,1}$ will never release a job simultaneously, thereby violating the optimality condition of DM. One should therefore conclude that DM is not optimal for distributed systems. On the other hand, Davis and

Burns (Davis and Burns, 2009) proved that the Audsley's OPA algorithm is optimal regarding the assignment of tasks priorities as long as there exists a schedulability test S respecting the following three conditions:

- (C1) the schedulability of a thread $\theta_{i,j,1}$ according to the test S may be dependent on the set of higher priority threads (denoted as $hp(\theta_{i,j,1})$), but not on the relative priority order of those threads;
- (C2) the schedulability of a thread $\theta_{i,j,1}$ according to the test S may be dependent on the set of lower priority threads, but not on the relative priority order of those threads, and;
- (C3) for two threads with adjacent priority, if their priorities are swapped then the threads that has been assigned the higher priority cannot become unschedulable according to the test S if it was schedulable at the lower priority.

The OPA algorithm is based on three simple steps (see Algorithm 5.1):

i. check the schedulability according to the test S of all non-priority-assigned threads, by assuming that they have the lowest priority;

ii. arbitrarily choose one thread that respects its deadline;

iii. remove the chosen thread from the list of non-priority-assigned thread and start again.

To verify the schedulability of the thread set (line 3), the schedulability analysis presented in (Tindell and Clark, 1994) is used. Note however that other tests could also be used (e.g., (Palencia and Gonzalez Harbour, 1998, 1999)).

Algorithm 5.1: OPA$(\theta_{\mathrm{i,j,1}}, \pi_{\mathrm{k}})$

```
1 for each priority level k, the lowest first do
2     for each unassigned task $\theta_{i,j,1}$ do
3         if $\theta_{i,j,1}$ is schedulable at priority k according to S with all unassigned tasks assumed to have higher priorities then
4             assign $\theta_{i,j,1} \leftarrow$ priority $k$;
5             break; //continue outer loop
6     return unschedulable;
7 return schedulable;
```

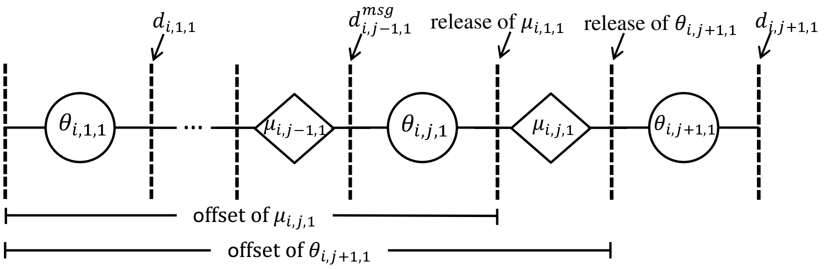

Figure 5.3: Intermediate deadlines for sequential applications.

The worst-case response time $r_{\theta_{i,j,1}}$ of an independent thread $\theta_{i,j,1}$ scheduled with a preemptive fixed priority scheduling algorithm can be calculated as in Eq. 2.1 ((Joseph and Pandya, 1986)), and for the case of $\theta_{i,j,1}$ is given by Eq. 5.1 :

$$r^{n+1}_{\theta_{i,j,1}} = C_{i,j,1} + \sum_{\theta_{a,b,1} \in hp(\theta_{i,j,1})} \left\lceil \frac{r^{n}_{\theta_{i,j,1}}}{T_a} \right\rceil C_{a,b,1} \tag{5.1}$$

where $hp(\theta_{i,j,1})$ is the set of threads with a higher priority than $\theta_{i,j,1}$ that can interfere with its execution. Due to the presence of the term $r_{\theta_{i,j,1}}$ on both sides of equation 5.1, this equation can be solved in an iterative manner, $r^{n+1}_{\theta_{i,j,1}} = C_{i,j,1} + \sum_{\theta_{a,b,1} \in hp(\theta_{i,j,1})} \left\lceil \frac{r^{n}_{i,j,1}}{T_a} \right\rceil C_{a,b,1}$ with $r^{1}_{\theta_{i,j,1}} = C_{i,j,1}$. The iteration stops when $r^{n}_{\theta_{i,j,1}} = r^{n+1}_{\theta_{i,j,1}}$.

In a distributed system, the Worst-Case Response Time (WCRT) of a thread $\theta_{i,j,1}$ (denoted as $r_{\theta_{i,j,1}}$) can then be computed as in (Tindell and Clark, 1994). That is:

$$r_{\theta_{i,j,1}} = r_{\theta_{i,j,1}} + \sum_{l=1}^{j-1} (r_{\theta_{i,l,1}} + r_{\mu_{i,l,1}}) \tag{5.2}$$

where $r_{\mu_{i,l,k}}$ is the response time of a message $\mu_{i,l,k}$, obtained with a network dependent analysis such as (Davis et al., 2013). A task τ_i (and hence its constituting threads and messages) is deemed schedulable if $r_{\theta_{i,n_i,1}} \leq D_i$. Unfortunately, this schedulability test makes the schedulability of a thread $\theta_{i,j,1}$ dependent on the response time of a previous message $\mu_{i,j-1,1}$, and hence the priority ordering of all the other threads $\theta_{i,j,1}$ and messages $\mu_{i,j,1}$ in τ_i. Conditions C1 and C2 are thus broken, making OPA unusable. Therefore the threads and messages with dependencies are transformed into an equivalent set of threads and messages without dependencies by imposing an intermediate deadline $d_{i,j,1}$ ($d^{msg}_{i,j,1}$, resp.) to each thread $\theta_{i,j,1}$ (each message $\mu_{i,j,1}$, resp.) (see Figure 5.3). The intermediate deadline $d_{i,j,1}$ of $\theta_{i,j,1}$ then becomes an offset on the release of the message $\mu_{i,j,1}$, and the deadline $d^{msg}_{i,j,1}$ of $\mu_{i,j,1}$, becomes an offset on the release of $\theta_{i,j+1,1}$. Therefore:

$$\begin{cases} r_{\theta_{i,j,1}} = d^{msg}_{i,j-1,1} + r_{\theta_{i,j,1}} \\ r_{\mu_{i,j,1}} = d_{i,j-1,1} + r_{\mu_{i,j,1}}, \end{cases} \tag{5.3}$$

implying that the WCRT of each task and message becomes independent on the relative priority order of higher and lower priority threads. Now, a thread $\theta_{i,j,1}$ (a message $\mu_{i,j,1}$, resp.) is deemed schedulable, if $r_{\theta_{i,j,1}} \leq d_{i,j,1}$ ($r_{\mu_{i,j,1}} \leq d^{msg}_{i,j,1}$, resp.), implying that the three Audsley's OPA algorithm validity conditions (C1, C2 and C3) are respected.

The threads and messages intermediate deadlines are computed as a function of the task end-to-end deadline and the threads and messages WCETs ($C_{i,j,1}$ and $M_{i,j,1}$, respectively). For threads and messages, the intermediate deadlines are given by:

$$d_{i,j,1} = d^{msg}_{i,j-1,1} + \frac{C_{i,j,1}}{\sum_{l=1}^{n_i}(C_{i,l,1} + M_{i,l,1})} D_i \tag{5.4}$$

$$d^{msg}_{i,j,1} = d_{i,j,1} + \frac{M_{i,j,1}}{\sum_{l=1}^{n_i}(C_{i,l,1} + M_{i,l,1})} D_i \tag{5.5}$$

Note that from those definitions, it results that $d_{i,n_i,1} = D_i$. Hence, if all threads (and messages) respect their intermediate deadlines $d_{i,j,1}$ ($d^{msg}_{i,j,1}$, resp.), i.e., $r_{\theta_{i,j,1}} \leq d_{i,j,1}$, the end-to-end deadline D_i of task τ_i is also respected.

5.2.2 Distributed using Optimal Priority Assignment (DOPA)

The problem of partitioning a set of threads onto the processors of a distributed platform and assigning priorities to threads and messages composing such a set, is solved by the DOPA algorithm presented in Algorithm 5.2. The algorithm is based on the following idea. If two successive threads $\theta_{i,j,1}$ and $\theta_{i,j+1,1}$ of the same task τ_i are assigned to the same processor π_k, the message $\mu_{i,j,1}$ sent between $\theta_{i,j,1}$ and $\theta_{i,j+1,1}$ can be omitted, thereby reducing the load on the network and increasing the acceptable response time for the other threads and messages in τ_i. Therefore, DOPA(τ) optimises the number of successive threads of the same task being assigned on the same processor.

5.2.3 Comparing the use of OPA and DM

In this section some simulation results validating the DOPA heuristic are presented. For all experiments the Algorithm 5.2 is used for the partition of threads and messages onto the elements of the distributed system, also two different priority assignment algorithms are used, namely DM and OPA.

One of the main objectives of this chapter is to demonstrate that by using the OPA algorithm, for the case of threads with dependencies, it is possible to increase in average the number of schedulable tasks and messages in a distributed system when compared

Algorithm 5.2: DOPA(τ)

```
for all $\tau_i$ ordered by non-increasing $\delta_i$ do
    for all $\theta_{i,j,1} \in \tau_i \cap \Upsilon$ do
        assign $\theta_{i,j,1}$ to $\pi_k \mid \theta_{i,j-1,1} \in \pi_k$ assuming $C^{msg}_{i,j-1,1} = 0$;
        recompute intermediate deadlines;
        call OPA( $\theta_{i,j,1}, \pi_k$ );
        if OPA( $\theta_{i,j,k}, \pi_k$ ) succeeds to assign $\theta_{i,j,1}$ then
            break;
        else if $\theta_{i,j+1,1} \in \mathscr{A}$ then
            assign $\theta_{i,j,1}$ to $\pi_l \mid \theta_{i,j+1,1} \in \pi_l$ assuming $M_{i,j,1} = 0$;
            recompute intermediate deadlines;
            call OPA( $\theta_{i,j,1}, \pi_l$ );
            if OPA( $\theta_{i,j,1}, \pi_l$ ) succeeds to assign $\theta_{i,j,1}$ then
                continue;
        for all $\pi_k$ in Worst-Fit order do
            assign $\theta_{i,j,1} \rightarrow \pi_k$;
            call OPA( $\theta_{i,j,1}, \pi_k$ );
            if OPA( $\theta_{i,j,1}, \pi_k$ ) succeeds to assign $\theta_{i,j,1}$ then
                assign message $\mu_{i,j-1,1}$ to the network;
                verify schedulability of $\mu_{i,j-1,1}$
                if message $\mu_{i,j-1,1}$ is schedulable then
                    declare schedulable;
                    break; //continue outer loop
                else
                    return unschedulable;
            else
                return unschedulable;
```

to the utilization of the DM priority assignment, frequently used in other works (e.g., (Tindell et al., 1992; García and Harbour, 1995; Richard et al., 2003)).

The use of the OPA versus de OPA through is evaluated through simulations. For generating the tasks τ_i and their respective threads $\theta_{i,j,1}$ and messages $\mu_{i,j,1}$ the guidelines presented in (Emberson et al., 2010) are followed. For generating random task sets for multiprocessor systems, using the Stafford's Randfixedsum algorithm (Stafford, 2004). The Randfixedsum algorithm generates a set of n values which are evenly distributed and whose components sum to a constant value. Thus, the Randfixedsum algorithm for generating unbiased sets of tasks with a fixed total density δ_{tot} is used. For a given total density δ_{tot}, the algorithm returns n different densities δ_i with values ranging between a minimum density $\delta_i^{min} = 0.1$ and a maximum density $\delta_i^{max} = 0.9$. For generating the threads

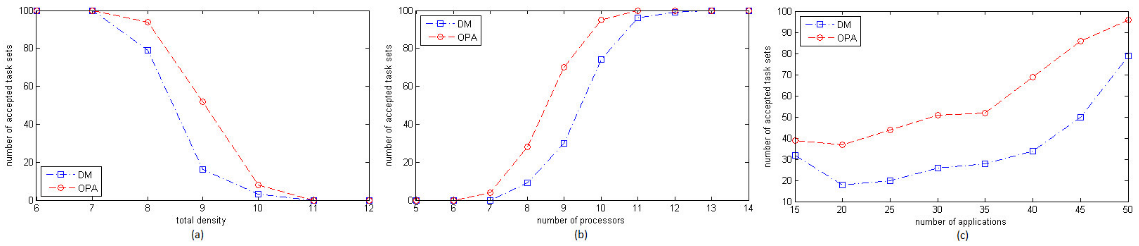

Figure 5.4: 100 experiments varying (a) the total density δ_{tot}, (b) the number of processors, and (c) the number of tasks in the system.

and messages densities the Randfixedsum algorithm is used again, taking as an input the previous generated densities $\delta_i = \sum(\delta_{i,j,1} + \delta^{msg}_{i,j,1})$, obtaining a set of values $\delta_{i,j,1} = \frac{d_{i,j,1}}{D_i}$ for tasks and $\delta^{msg}_{i,j,1} = \frac{d^{msg}_{i,j,1}}{D_i}$ for messages with values ranging between a minimum density bound for tasks and messages $\delta^{min}_{i,j,1} = 0.01$ and a maximum density of $u^{max}_{i,j,1} = 0.9$. The WCETs of tasks $C_{i,j.1}$, messages $M_{i,j,1}$ and end-to-end deadlines D_i are generated as recommended in (Emberson et al., 2010); it is considered that tasks have implicit end-to-end deadlines (i.e., $D_i = T_i$) following a uniform distribution. For each experiment 100 task sets are generated.

Figure 5.4a shows the number of accepted task sets over 100 experiments for different total densities δ_{tot}. 50 tasks that execute threads and transmit messages in a computing platform of 10 processors and 1 network are simulated. It is possible to see that OPA in average performs better in terms of the number of accepted task sets. For example, the OPA algorithm accepts 52% of task sets with a total system density of 9. In contrast, the DM algorithm reaches 16% with the same system density.

Figure 5.4b shows the number of accepted task sets for 100 experiments simulating 50 tasks that execute threads and transmit messages in a computing platform composed of 1 network and a varying number of processors. The density is fixed to $\delta_{tot} = 8$. It is possible to see that OPA in average performs better, for example, when the number of processors is equal to 9, the OPA algorithm accepts 70% of task sets, whilst the DM algorithm only accepts 30% of task sets.

Figure 5.4c shows the number of accepted tasks sets over 100 experiments, in which the number of tasks with a fixed total density $U_{tot} = 8$ is varied to be scheduled in a computing platform of 10 processors interconnected by a real-time network. In the range between 10 and 50 tasks, OPA always accepts more task sets than DM. For example, for the case of 40 tasks, the OPA algorithm accepts 69% of task sets, in contrast the number of accepted tasks sets obtained by the DM algorithm is 34%. Note that the number of accepted task sets increases with the number of generated tasks. This behaviour can be explained by the fact that the average density of threads and messages decreases, thereby meaning that more threads can be accommodated on each processor in average.

The effects presented in Figures 5.4(a-c) can be explained because when DM is used for assigning priorities, it fails more often than OPA due to its non-optimality. Therefore, such non-schedulable tasks need to be partitioned onto other processor in the distributed system, thus increasing the number of messages in the network, which leads to an increasing number of unschedulable systems.

5.3 The Parallel-DOPA (P-DOPA) Heuristic

The DOPA heuristic presented in Section 5.2 considers the partition and priority assignment of linear transactions. The Parallel-DOPA (P-DOPA) heuristic is an extension of DOPA heuristic that considers the allocation of threads and messages for the P/D task model (Garibay-Martínez et al., 2014b) introduced in Section 3.4. The straightforward extension of the algorithm presented in Section 5.2 would involve to impose intermediate deadlines to each thread and each message of every task τ_i by using the same proportional assignment heuristic. That is, each thread $\theta_{i,j,k} \in \sigma_{i,j}$ and each message $\mu_{i,j,k} \in \sigma_{i,j}$ would be assigned an intermediate deadline $d_{i,j,k}$ and $d^{msg}_{i,j,k}$, respectively. This approach is called Proportional heuristic hereafter. The deadlines are given by:

$$d_{i,j,k} = d^{msg}_{i,j-1,k} + \frac{C_{i,j,k}}{\sum_{l=1}^{n_i}(C_{i,l,1} + M_{i,l,1})} D_i \tag{5.6}$$

$$d^{msg}_{i,j,k} = d_{i,j,k} + \frac{M_{i,j,k}}{\sum_{l=1}^{n_i}(C_{i,l,1} + M_{i,l,1})} D_i \tag{5.7}$$

A release offset $\phi_{i,j,k} = d^{msg}_{i,j-1,k}$ and $\phi^{msg}_{i,j,k} = d_{i,j,k}$, is given to threads and messages, respectively.

5.3.1 Intermediate Deadlines for Distributed Execution Paths (DEP)

Since the master thread resulting of the DST is assigned to its own reserved processor, no other task can interfere with its execution (see Section 4.2). Therefore, the master thread will always respect its end-to-end deadline D_i, and no intermediates deadlines must be computed for the threads constituting it. Because no more parallel threads can be added to the master thread without causing a deadline miss, the messages associated to the parallel threads that are not part of the master thread could not be omitted, thus a partitioning algorithm has to be used. All messages related to the P/D task that must be transited through the network are known a *priori* by the partitioning algorithm.

When scheduling a P/D task τ_i, an offset $\phi^{DPath}_{i,j}$ and a deadline $d^{DPath}_{i,j}$ define the scheduling window in which threads and messages of each Distributed Execution Path

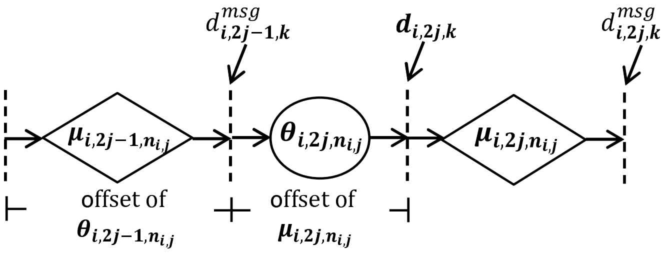

Figure 5.5: Intermediate deadlines of a DEP.

(DEP) have to start and complete their execution. However, inside a DEP, the activation of the threads and messages depends on the response times of the previous messages and threads in the DEP. Similarly to the case of linear transactions addressed in Section 5.2, OPA is not directly usable and therefore the threads and messages within a DEP are transformed into an equivalent set of threads and messages without dependencies by imposing an intermediate deadline $d_{i,j,k}$ ($d^{msg}_{i,j,k}$, resp.) to each thread $\theta_{i,j,k}$ (each message $\mu_{i,j,k}$, resp.) (see Figure 5.5). The intermediate deadline $d^{msg}_{i,j-1,k}$ of $\mu_{i,j-1,k}$ then becomes an offset on the release of the thread $\theta_{i,j,k}$, and the deadline $d_{i,j,k}$ of $\theta_{i,j,k}$, becomes an offset on the release of $\mu_{i,j,k}$. Therefore, in a similar way as in Section 5.2, it results that:

$$\begin{cases} r_{\mu_{i,j-1,k}} = \phi^{DPath}_{i,j-1} + r_{\mu_{i,j-1,k}} \\ r_{\theta_{i,j,k}} = d^{msg}_{i,j-1,k} + r_{\theta_{i,j,k}} \\ r_{\mu_{i,j,k}} = d_{i,j,k} + r_{\mu_{i,j,k}} \end{cases} \tag{5.8}$$

implying that the WCRT of each thread and message becomes independent on the relative priority order of higher and lower priority tasks. Therefore, a thread $\theta_{i,j,k}$ (a message $\mu_{i,j,k}$, resp.) is deemed schedulable if $r_{\theta_{i,j,k}} \leq d_{i,j,k}$ ($r_{\mu_{i,j,k}} \leq d^{msg}_{i,j,k}$, resp.), and the three OPA validity conditions C1, C2 and C3 are respected.

The intermediate deadlines for threads and messages within a DEP are computed as a function of the DEP window length $d^{DPath}_{i,j}$, and the threads and messages WCETs:

$$d^{msg}_{i,j-1,k} = \phi^{DPath}_{i,j} + \frac{M_{i,j-1,k}}{M_{i,j-1,k} + C_{i,j,k} + M_{i,j,k}} d^{DPath}_{i,j} \tag{5.9}$$

$$d_{i,j,k} = d^{msg}_{i,j-1,k} + \frac{C_{i,j,k}}{M_{i,j-1,k} + C_{i,j,k} + M_{i,j,k}} d^{DPath}_{i,j} \tag{5.10}$$

$$d^{msg}_{i,j,k} = d_{i,j,k} + \frac{M_{i,j,k}}{M_{i,j-1,k} + C_{i,j,k} + M_{i,j,k}} d^{DPath}_{i,j} \tag{5.11}$$

5.3.2 P-DOPA heuristic

The problem of partitioning the set of remaining threads and messages after applying the DST to the elements of the distributed platform and assigning priorities to those threads and messages, is solved by the P-DOPA heuristic presented in Algorithm 5.3.

Algorithm 5.3: $\mathrm{P-DOPA}(\tau)$

```
call DST( $\tau, \pi$ )
for all non-assigned $\theta_{i,j,k}$ in a DEP do
    for all $\pi_k$ in Worst-Fit order do
        assign $\theta_{i,j,k} \rightarrow \pi_k$
        call OPA( $\theta_{i,j,k}, \pi_k$ )
        if OPA( $\theta_{i,j,k}, \pi_k$ ) succeeds to assign $\theta_{i,j,k}$ then
            assign message $\mu_{i,j,k}$ to the network
            verify schedulability of $\mu_{i,j-1,k}$
            if message $\mu_{i,j,k}$ is not schedulable then
                return unschedulable
        else
            return unschedulable
```

By looking at Algorithm 5.3, it can be noticed that the complexity of the partitioning algorithm has been reduced in comparison to Algorithm 5.2, when P/D tasks are considered and the DST transformation is performed first. Thanks to the DST, the number of messages that must be transmitted over the network is minimal and cannot be further reduced. Thus, there is no reason to try to perform a specific assignments to reduce the workload on the network as it is the case in Algorithm 5.2. Algorithm 5.3 simply assigns the threads of the DEPs using a Worst-Fit heuristic. Their priority being determined using OPA, the interest of which was already shown through the simulation results provided in Section 5.2.3.

5.3.3 Evaluating the Parallel-DOPA Heuristic

This section presents some experiments for evaluating the P-DOPA heuristic. Because the advantage of using OPA instead of DM for the assignment of priorities to tasks with precedence constraints has already been shown in Section 5.2.3, this section focuses on the evaluation of the use of the DST transformation versus the use of the Proportional heuristic for assigning intermediate deadlines to threads $\theta_{i,j,k}$ and messages $\mu_{i,j,k}$ in a task τ_i. The objective of this comparison is to show that the DST is superior when assigning

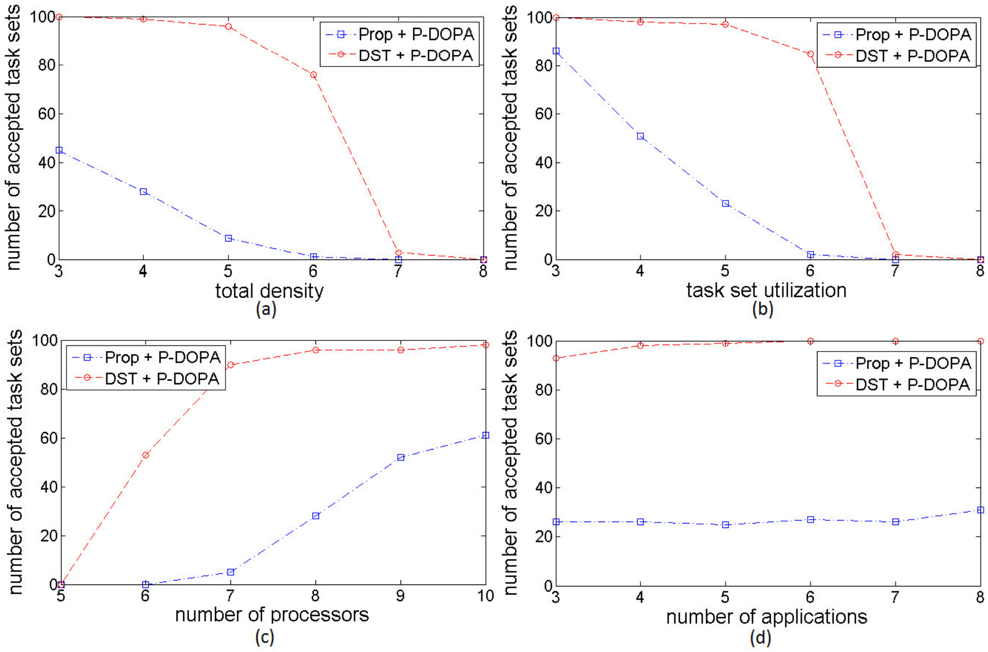

Figure 5.6: 100 experiments varying (a) the total density δ_{tot} with a $SpeedUP = 10$, (b) the total density δ_{tot} with a $SpeedUP = 20$ (c) the number of processors with a $SpeedUP = 20$, and (d) the number of tasks in the system with a $SpeedUP = 20$.

intermediate deadlines to sequential and parallel segments of a P/D task, when compared to the Proportional heuristic, thus allowing P-DOPA to schedule more task sets.

Similarly to Section 5.2.3, in this section the Randfixedsum algorithm (Stafford, 2004) is used for the generation of P/D tasks τ_i and their respective threads $\theta_{i,j,k}$ and messages $\mu_{i,j,k}$. The Randfixedsum generates unbiased sets of tasks with a fixed total density δ_{tot}. For a given total density δ_{tot}, the algorithm returns n different densities δ_i with values ranging between a minimum density $\delta_i^{min} = 0.5$ and a maximum density $\delta_i^{max} = 2$. For generating the threads and messages densities the Randfixedsum algorithm is used again taking as an input the previous generated densities $\delta_i = \sum(\delta_{i,j,k} + \delta_{i,j,k}^{msg})$, obtaining a set of values $\delta_{i,j,k}$ for threads (and $\delta_{i,j,k}^{msg}$ for messages) ranging between a minimum density $\delta_{\theta_{i,j,k}}^{min} = 0.05$ and a maximum density of $\delta_{\theta_{i,j,k}}^{max} = 0.3$ (a minimum density $\delta_{\mu_{i,j,k}}^{min} = 0.0025$ and a maximum density of $\delta_{\mu_{i,j,k}}^{max} = 0.075$ for messages, resp.). It is considered that tasks have implicit end-to-end deadlines ($D_i = T_i$) following a uniform distribution. P/D tasks are hardly constrained when compared to sequential tasks, since their density δ_i can be larger than 1 and each parallel segment is composed of multiple threads transmitting messages simultaneously. Due to this, a large amount of messages is generated (2 messages $\mu_{i,j,k}$ and $\mu_{i,j+1,k}$ per each thread $\theta_{i,j+1,k}$), thus, in order to be able to schedule such messages, the network speed needs to be increased by a factor $SpeedUp$. For each experiment 100 task sets are generated.

Figure 5.6a shows the number of accepted tasks sets over 100 experiments for different total densities δ_{tot}. 4 P/D tasks that execute threads and transmit messages in a computing platform of 8 processors and 1 real-time network are simulated. It is possible to see that DST + P-DOPA in average performs better than Proportional + P-DOPA in terms of number of accepted task sets. For example, the DST + P-DOPA algorithm accepts 96% of task sets with a total system density of $\delta_{tot} = 5$. In contrast, the Proportional + P-DOPA algorithm reaches 16% with the same system density. Those results are obtained when $SpeedUp = 10$. The main reason to use the $SpeedUp$ factor is due to the fact that the observed majority of failed assignments in the case of Proportional + P-DOPA algorithm were due to the lack of capacity in the network.

For the experiments depicted in Figures 5.6(b-d) it was decided to use a $SpeedUp = 20$ for the network. The reason behind that is to be more fair with the Proportional heuristic.

Similarly to Figure 5.6a, Figure 5.6b shows the number of accepted task sets for 100 experiments simulating 4 P/D tasks that execute tasks and transmit messages in a computing platform composed of 1 real-time network. It is possible to see that DST + P-DOPA on average perform better than Proportional + P-DOPA. The DST + P-DOPA algorithm accepts 97% of task sets with a total system density of $\delta_{tot} = 5$. In contrast, the Proportional + P-DOPA algorithm reaches 23% with the same system density. It is possible to see that when $SpeedUp = 20$ there are less scheduling failures in the network, allowing both heuristics to increase their number of accepted task sets.

Figure 5.6c shows a variation over the number of processors. The density is fixed to $\delta_{tot} = 5$ to be scheduled in a computing platform of 8 processors and 1 network. It is possible to see that DST + P-DOPA in average performs better than Proportional + P-DOPA. The maximum difference found for these experiments happens when the number of processors is equal to 7, the DST + P-DOPA algorithm accepts 90% of task sets, whilst the Proportional + P-DOPA algorithm only accepts 5% of task sets.

Figure 5.6d shows the variation over the number of tasks with a fixed total density $U_{tot} = 5$ to be scheduled in a computing platform of 8 processors and 1 network. In the range between 3 and 8 tasks, DST + P-DOPA always accepts more task sets than Proportional + P-DOPA. If δ_{tot} stays constant and the number of tasks increases, it can be the case that $C_i < T_i$, therefore the DST is able to transform the P/D tasks into a sequential task by omitting all messages, and increasing the chances of successfully accepting the task set.

The effects presented in Figures 5.4(a-d), can be explained because, when the DST is used for assigning intermediate deadlines, the length of the scheduling window for threads and messages within a parallel segment, is the maximum possible for the case of a P/D task. Therefore, it will always be better or equal to the Proportional heuristic, previously used for sequential tasks.

5.4 Summary

This chapter presented the DOPA heuristic for the simultaneous partitioning and priority assignment of threads and messages onto the constituting elements of the distributed system by using the OPA algorithm known as Audsley's algorithm (Audsley, 1991).

A method that imposes intermediate deadlines to threads and messages has been proposed with the objective of permitting the use of OPA for task sets with dependencies (distributed tasks). It is demonstrated through simulations that OPA increases, in average, the number of schedulable tasks and messages in a distributed system, when compared to the DM algorithm, when using the same partition algorithm.

The results of the DOPA huristic are extended for P/D tasks and showed that the DST transformation helps to reduce the complexity of the assignment and to relax the constraints on the intermediate deadlines that must be respected by the threads and messages constituting the P/D tasks. It is demonstrated through simulations, that the use of DST for the intermediate deadline assignment for threads and messages considerably increases the number of schedulable tasks in a distributed system while compared to the Proportional heuristic used for the linear transactional model.

The following publications have been derived from the research related this chapter:

- **R. Garibay-Martínez**, G. Nelissen, L. L. Ferreira, and L. M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems. In Marisol García-Valls and Tommaso Cucinotta, editors, *Second International Workshop on Real-time and distributed computing in emerging applications*. Universidad Carlos III de Madrid, 2013b.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, and L.M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems, J. Compt. Syst. Sci. (2015), `http://dx.doi.org/10.1016/j.jcss.2015.05.005`.

Chapter 6

Holistic Analysis for P/D Tasks using the FTT-SE Protocol

6.1 Introduction

In current automotive applications, tens of Electronic Control Units (ECUs) are interconnected by different network technologies (see Section 2.3). But such network technologies only provide low bandwidth. Emerging automotive applications, such as infotainment and video-based driver assistance systems, require significantly bigger processing capacity and a network that conciliate high bandwidth with real-time guarantees, heterogeneous traffic types and dynamic scheduling. Several Real-Time Ethernet (RTE) protocols (e.g. TTEthernet, AV-Bridges) are currently being investigated for vehicular data networks, but they have some limitations (e.g. AV-Bridges do not support scheduled traffic; TTEthernet is inflexible regarding the time-triggered traffic, because the support to real-time event-triggered traffic is limited, since only provides a basic bandwidth reservation mechanism). For this reasons, this work uses Flexible Time Triggered - Switched Ethernet (FTT-SE) which is a research protocol that satisfies the requirements of emerging automotive applications (e.g., high bandwidth with real-time guarantees, handling of heterogeneous real-time traffic and dynamic scheduling). Furthermore, those systems will require higher computing power, therefore, the use of more powerful computing models such as the P/D tasks model seem a promising alternative.

When scheduling P/D tasks, the interaction between the threads executing on different nodes and their respective messages must be considered. A well accepted technique for the verification of the temporal correctness of a distributed real-time system is the holistic analysis. The holistic analysis studies the behaviour of each of the elements of the distributed system as a whole.

This chapter presents a holistic timing analysis for the computation of the Worst-Case Response Time (WCRT) for P/D tasks when transformed by the DST algorithm (see

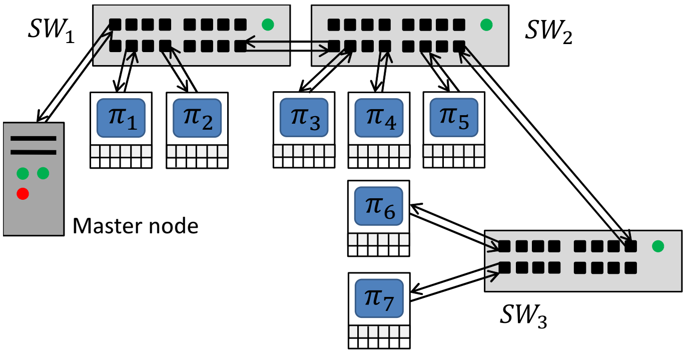

Figure 6.1: FTT-SE single-master architecture.

Section 4.2). This chapter also presents an extension for the analysis by considering an FTT-SE transmission network as the one presented in Figure 6.1. Both synchronous and asynchronous communication patterns in the FTT-SE protocol are considered. Finally, a technique for reducing the pessimism when computing the WCRT of P/D tasks by considering a pipeline effect observed in such systems is introduced. Note that this improvement is not limited to the use of the DST algorithm and can be used in any distributed system, using a FTT-SE network interconnecting computing nodes scheduled with a fixed-priority algorithm.

Section 6.2 briefly describes a technique for computing the WCRT of messages scheduled with the FFT-SE protocol (Ashjaei et al., 2013). The proposed holistic analysis presented in Section 6.3. Section 6.4 shows how to improve the WCRT computation presented in Section 6.3. A numerical example is shown in Section 6.5, and finally, a summary is presented in Section 6.7.

6.2 The FTT-SE Protocol

The FFT-SE protocol makes use of the master/slave paradigm (Marau et al., 2006, 2012; Ashjaei et al., 2013, 2014), where a dedicated node (the *master node*) schedules messages on the network. The communications within a FTT-SE network are done based on fixed duration time slots called *Elementary Cycles* (ECs). Figure 6.2, shows the structure of an EC.

The FTT-SE protocol is able to manage the transmission of real-time traffic and non-real-time traffic. An EC is divided on three main windows: the *signalling window*, the *real-time window*, and the *non-real-time window*. The real-time window is further divided into two sub-windows: the *synchronous window* and the *asynchronous window*. These windows are reserved for transmission of periodic (synchronous) and sporadic

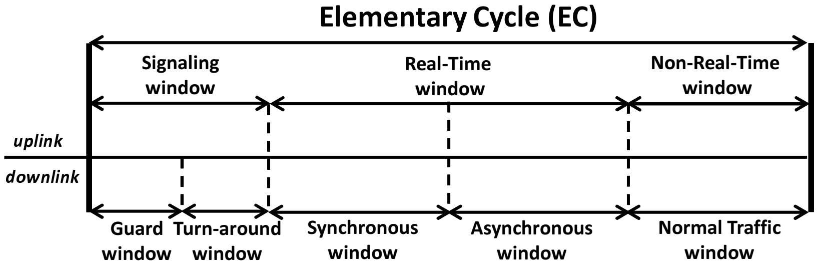

Figure 6.2: FTT-SE Elementary Cycle (EC) structure.

(asynchronous) traffic, respectively. At the end of the EC there exists a non-real-time window reserved for best-effort Ethernet traffic.

The duration of the EC and its corresponding windows is tuneable (Marau, 2009), and defines the system resolution, thus, it defines the message periods and deadlines. Deadlines and periods are expressed as integer multiples of the EC duration.

6.2.1 Message Scheduling on the FTT-SE Protocol

Synchronous messages are scheduled autonomously by the master, without any petition/feedback from the slave nodes. Thus, at the right time, imposed by a global clock (owned by master node), the nodes send the periodic traffic. Thus, the master node is responsible for triggering the transmission of periodic messages.

Asynchronous messages are also scheduled by the master node, but asynchronous messages are activated in response to events that happen in the environment, thus, *slave nodes* must report its activation to the Master via a *signalling mechanism* (Marau, 2009). This signalling informs to the master the desire for transmission from the slave nodes. A similar notification and processing scheme is employed for the non-real-time traffic. The difference is that real-time traffic is subject to an admission control procedure when registered in the system. Therefore, if the real-time traffic is accepted, its timing requirements are guaranteed. Non-real-time traffic is not subject to registration and therefore it has no guarantees.

The construction of the EC schedule is done by keeping updated tables for synchronous and asynchronous messages. The scheduler applies a scheduling policy (e.g., Deadline Monotonic) over these tables, generating the ready queues for transmission during that EC. The scheduler picks messages from the ready queue and verifies if they fit on that scheduling window, considering all delays for that EC in each of the transmission links. That process is repeated until no other message fits on the scheduling window for that EC (i.e., considering all messages from higher to lower priority). If they fit, they are removed from the ready queue and transmitted in the next EC. The remaining messages are kept in the ready queue and wait for being scheduled in the following ECs. The current EC schedule is sent to the nodes via the Triggered Message (TM).

For building the EC, it is important to consider:

i. the characteristics of the transmission links; switched Ethernet has full-duplex transmission links, namely the uplink $l^{u}_{i,j}$ that connects the nodes to the switch, and the downlink $l^{d}_{i,j}$ connecting the ports exiting the switch to the nodes;

ii. the multiple switching delays; when transmitting messages with FTT-SE, a switching delay for a message $\mu_{i,j,k}$ (denoted as $SD_{i,j,k}$) must be considered when crossing a switch SW_x. In this chapter it is considered that the switching delay has two components, the switch relaying latency (denoted as Δ), and the Store-and-Forward Delay of a message $\mu_{i,j,k}$ (denoted as $SFD_{i,j,k}$), i.e., $SD_{i,j,k} = SFD_{i,j,k} + \Delta$. Δ is related to the hardware specifications of the switch. $SFD_{i,j,k}$ is related to the store-and-forward function of the switch when conveying messages, thus it depends on the message size and link speed, consider the following switching delay with an example.

   Example 6.1. *Consider the system architecture shown in Figure 6.1. Assume two synchronous messages, $\mu_{1,1,1}$ and $\mu_{2,1,1}$ that are transmitted from π_1 to π_4 and from π_2 to π_4, respectively. Message $\mu_{1,1,1}$ has higher priority than message $\mu_{2,1,1}$. These messages share the link $l^{d}_{SW_2,\pi_4}$. In Figure 6.3a and Figure 6.3b, it is possible to observe two different scenarios. In Figure 6.3a the amount of allocated bandwidth for the downlink $l^{d}_{SW_2,\pi_4}$ is represented as a bin. Since message $\mu_{1,1,1}$ has the highest priority, it is assigned first to the bin as well as its switching delay. Message $\mu_{2,1,1}$ is second to be assigned into the bin and since the switching delay of $\mu_{2,1,1}$ is larger than the one of $\mu_{1,1,1}$, the switching delay of $\mu_{2,1,1}$ is considered. Conversely, Figure 6.3b shows the case in which the switching delay of $\mu_{2,1,1}$ is smaller than the one of $\mu_{1,1,1}$, therefore, only the switching delay of $\mu_{1,1,1}$ is considered.*

   Thus, when scheduling messages that share a downlink, all the WCML are considered, but only the maximum switching delay of all messages has to be considered for that link, for that specific EC. This process is repeated in each EC, and;

iii. the length of the specific transmission window for each type of traffic (e.g., synchronous or asynchronous window), the length of such a window is the reserved bandwidth for transmission in that EC.

6.2.2 Worst-Case Response Time in FTT-SE Networks

There exist different studies related to the FTT protocol over Ethernet (e.g., (Marau et al., 2006, 2012; Ashjaei et al., 2013, 2014)). In this section an analysis based on network

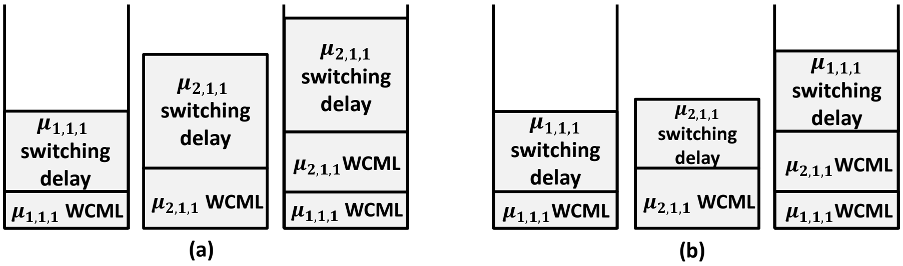
$\mu_{1,1,1}$ switching delay
$\mu_{1,1,1}$ WCML
$\mu_{2,1,1}$ switching delay
$\mu_{2,1,1}$ WCML
$\mu_{2,1,1}$ switching delay
$\mu_{2,1,1}$WCML
$\mu_{1,1,1}$ WCML
(a)
$\mu_{1,1,1}$ switching delay
$\mu_{1,1,1}$WCML
$\mu_{2,1,1}$ switching delay
$\mu_{2,1,1}$ WCML
$\mu_{1,1,1}$ switching delay
$\mu_{2,1,1}$WCML
$\mu_{1,1,1}$WCML
(b)

Figure 6.3: Switching delay: (a) maximum switching delay: $\mu_{1,1,1}$, and (b) maximum switching delay: $\mu_{2,1,1}$.

calculus presented in (Ashjaei et al., 2013) for the computation of the WCRT of messages within the FTT-SE protocol is reviewed. For notational convenience, the original notation is replaced, with the notation introduced in Section 3.4.

The *request bound function* $rbf_{i,j,k}(t)$ represents the maximum transmission requirements generated by a message $\mu_{i,j,k}$ and all its higher priority messages during an interval $[0,t]$. The $rbf_{i,j,k}(t)$ is computed as:

$$rbf_{i,j,k}(t) = M_{i,j,k} + sn_{i,j,k} \times SD_{i,j,k} + Wl_{i,j,k}(t) + Wr_{i,j,k}(t), \tag{6.1}$$

where, $sn_{i,j,k}$ is the number of switches that a message $\mu_{i,j,k}$ traverses from the origin node to its destination node, $Wl_{i,j,k}(t)$ is the *"Shared Link Delay"*, and $Wr_{i,j,k}(t)$ is the *"Remote Link Delay"*. The Shared Link Delay and the Remote Link delay are briefly explained below. For further details, please refer to (Ashjaei et al., 2013).

Shared Link Delay. The transmission of a message $\mu_{i,j,k}$, may be delayed by all the higher priority messages that share a link with $\mu_{i,j,k}$. However, such interference occurs only once, so messages that caused such interference on a previous link are excluded from the analysis for the next links. Also, when building the schedule for each EC, the scheduler has to consider the maximum switching delay $SD_{i,j,k}$, only once. Therefore, $Wl_{i,j,k}(t)$ is computed by separating the interference of messages from the *switching-delay-effect* (denoted as $Is_{i,j,k}(t)$) for each EC. The shared link delay is computed in Eq. (6.2):

$$Wl_{i,j,k}(t) = \sum_{\forall \mu_{a,b,c} \in SLD_{i,j,k}} \left\lceil \frac{t}{T_a} \right\rceil M_{a,b,c} + Is_{i,j,k}(t), \tag{6.2}$$

where $SLD_{i,j,k} = \{\mu_{a,b,c} \in \tau \mid \mu_{a,b,c} \neq \mu_{i,j,k} \wedge SW_i \cap SW_j \neq 0 \wedge \mu_{i,j,k} \in hp(\mu_{i,j,k}) \wedge \mu_{i,j,k} \in WT(\mu_{i,j,k})\}$, where, SW_i and SW_j represent the set of switches crossed by messages $\mu_{i,j,k}$ and $\mu_{a,b,c}$, respectively; $hp(\mu_{i,j,k})$ is the set of messages with priority higher than $\mu_{a,b,c}$ and $WT(\mu_{i,j,k})$ is the set of messages that are scheduled in the same window as $\mu_{a,b,c}$ (i.e., the synchronous or the asynchronous window).

For computing the switching-delay-effect $Is_{i,j,k}(t)$, it is needed to compute an upper bound on the number of switching delays ($\{SD_{i,j,k}\}$) for each message that contribute to Eq. (6.2), per EC. Depending on time t, a number of switching delays $SD_{i,j,k}$ are inserted into an array $G_{i,j,k}(t)[l] = \{SD_{1,2,1}, \dots, SD_{n,n_i-1,k}\}$, when a message crosses a switch in the network. In order to consider the maximum switching delays only, a sorted (in non-increasing order) array $G^{sort}_{i,j,k}(t)[l]$ containing the switching delays in $G_{i,j,k}(t)[l]$ is considered. The number of ECs in an interval $[0,t]$ is given by: $z(t) = \lceil \frac{t}{EC} \rceil$, thus, in order to consider the worst-case scenario for the computation of the WCRT, the first $z(t)$ elements from $G^{sort}_{i,j,k}(t)[l]$ are selected. Then, the switching-delay-effect is computed as:

$$Is_{i,j,k} = \sum_{l=1}^{z(t)} G^{sort}_{i,j,k}(t)[l]. \tag{6.3}$$

Remote Link Delay. A message $\mu_{i,j,k}$ can be blocked by other higher priority messages even if they do not share a transmission link. Let the source of interference be illustrated with a brief example.

Example 6.2. *Consider three messages $\mu_{i,j,1}$, $\mu_{i,j,2}$ and $\mu_{i,j,3}$. Message $\mu_{i,j,1}$ having the highest priority and $\mu_{i,j,3}$ having the lowest priority. It may be the case that messages $\mu_{i,j,1}$ and $\mu_{i,j,3}$ do not share a link, but $\mu_{i,j,2}$ does share a link with both $\mu_{i,j,1}$ and $\mu_{i,j,3}$. If $\mu_{i,j,1}$ delays the transmission of $\mu_{i,j,2}$ in their shared link, it is possible that $\mu_{i,j,2}$ "pushes" $\mu_{i,j,3}$ to be transmitted in the next EC due to the prior interference on $\mu_{i,j,2}$ caused by $\mu_{i,j,1}$.*

Thus, a higher priority message can delay a lower priority message even though they do not share a transmission link. Therefore, to compute the worst-case remote link delay, it is needed to consider all messages that share links with the messages that contributed to the shared link delay (Eq. (6.2)), excluding all messages that are already considered in Eq. (6.2). Hence:

$$Wr_{i,j,k}(t) = \sum_{\forall \mu_{p,q,r} \in RLD_{i,j,k}} \left\lceil \frac{t}{T_p} \right\rceil M_{p,q,r}, \tag{6.4}$$

where $RLD_{i,j,k} = \{\mu_{p,q,r} \in \tau \mid \mu_{p,q,r} \neq \mu_{a,b,c} \neq \mu_{i,j,k} \wedge SW_k \cap SW_j \neq 0 \wedge SW_k \cap SW_i = 0 \wedge SW_j \cap SW_i \neq 0 \wedge \mu_{p,q,r} \in hp(\mu_{a,b,c}) \wedge \mu_{p,q,r} \in WT(\mu_{a,b,c})\}$.

The demand bound function is then compared with the *supply bound function* $sbf_{i,j,k}(t)$, which represents the minimum effective communication capacity that the network supplies during the time interval $[0,t]$. In each EC, the bandwidth provided for transmitting each type of message is equal to $\frac{(LW-I)}{EC}$, where LW is the length of the specific transmission window and I is the maximum inserted idle time of such a window. The inserted idle time results from the fact that the maximum window duration cannot be exceeded. In the worst case, the idle time equals the maximum message size (Marau et al., 2012), thus, the supply bound function of the network is given by:

$$sbf_{i,j,k}(t) = (\frac{LW - I}{EC}) \times t. \tag{6.5}$$

Then, the response time of a message $\mu_{i,j,k}$ is computed by determining the time instant t^* such that:

$$t^* = \min(t > 0) : sbf_{i,j,k}(t) \geq rbf_{i,j,k}(t). \tag{6.6}$$

For determining the time instant t^* it is necessary to verify Eq. (6.6) at all instants in which $sbf_{i,j,k}(t)$ is modified due to the interference of other messages. Such time instants are given by:

$$CP_{rbf_{i,j,k}} = [\cup cp_{\mu_{i,j,q}}, \forall \mu_{i,j,q} \in hp(\mu_{i,j,k})] \cup T_{\mu_{i,j,k}}, \tag{6.7}$$

where, $cp_{\mu_{i,j,q}} = \{T_{\mu_{i,j,q}}, 2T_{\mu_{i,j,q}}, \ldots, n_{\mu_{i,j,q}} T_{\mu_{i,j,q}}\}, n_{\mu_{i,j,q}} = \left\lfloor \frac{T_{\mu_{i,j,k}}}{T_{\mu_{i,j,q}}} \right\rfloor$. Since it is not possible to determine the specific time of transmission of messages inside an EC, the computation of the WCRT for a message $\mu_{i,j,k}$ is in terms of a number of ECs, thus the WCRT (denoted as $r_{\mu_{i,j,k}}$) of a message $\mu_{i,j,k}$, in a synchronous system is given by:

$$r^{syn}_{\mu_{i,j,k}} = \left\lceil \frac{t^*}{EC} \right\rceil. \tag{6.8}$$

The previous analysis considers the transmission of synchronous messages. This overhead can be simply added to Eq. 6.8 as:

$$r^{asyn}_{\mu_{i,j,k}} = \left\lceil \frac{t^*}{EC} \right\rceil + 2. \tag{6.9}$$

When messages are scheduled within a single switch, messages may suffer from both shared link delay (Eq. (6.2)) and remote link delay (Eq. (6.4)), but only a single switching delay is considered. However, the $rbf_{i,j,k}(t)$ of messages for single switch is computed as in Eq. (6.1) except that $sn_{i,j,k} = 1$.

6.3 A Holistic Analysis for Stretched Tasks

In distributed systems, the impact of messages used for communication/synchronization purposes cannot be deemed negligible as in the case of multiprocessor systems. The main goal of the holistic analysis approach is to calculate the end-to-end response time associated to a chain of tasks and messages. Two types of communication patterns are identified: time-triggered and event-triggered. This depends on the type of messages used for transmission within the FTT-SE network, which can be synchronous or asynchronous messages. In the following, the holistic analysis for P/D tasks that have been stretched using the DST transformation (see Section 4.2) is presented. Both time-triggered and event-triggered systems are considered.

In FTT-SE networks, messages are transmitted in periodic time windows called ECs. Thus, if a thread completes its execution just after the beginning of an EC, it has to wait for the beginning of the next EC in order to initiate the transmission of a message. This delay is called *node queuing delay*. In the worst case it has a length of 1 EC. The node queuing delay has to be considered whenever a transmission is initiated by the P/D task (i.e., during each D-fork and D-join operation). This means that Eq. (6.8) must be incremented by 1 EC for synchronous messages, and incremented by 2 ECs for the case of asynchronous messages (1 ECs due to the signalling overhead inherent to asynchronous messages in FTT-SE (Ashjaei et al., 2013), and 1 due to the node queuing delay).

6.3.1 Time-triggered Systems

For the case in which the activation of a P/D message or of a P/D thread is based on specific time instants (i.e., P/D tasks are strictly periodic), it is possible to use a time-triggered communication pattern in which synchronous messages are used. For time-triggered systems, an offset indicates the earliest moment at which a thread $\theta_{i,j,k}$ (or message $\mu_{i,j,k}$) of a segment $\sigma_{i,j}$ can start its execution (or transmission, respectively). This offset is equal to the worst-case response time $r_{\mu_{i,j-1,k}}$ (resp., $r_{\theta_{i,j,k}}$) of the message (resp., thread) preceding $\theta_{i,j,k}$ (resp., $\mu_{i,j,k}$) in the fork-join task, thereby ensuring that the threads and messages never experience any release jitter.

Two cases must be considered when computing the response time of a parallel task stretched with the DST transformation (see Section 4.3):

Fully stretched tasks: if a task has been fully stretched, no message is sent over the network (Case 1 in Section 4.3). Therefore, its WCRT only depends on the interference caused by other higher priority threads executing on the same processor. This can be computed by using the response time analysis for fixed-priority tasks (Audsley et al., 1993):

$$r_{\tau_i} = C_i + \sum_{\forall \theta_{p,q,r} \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i} + J_{\theta_{p,q,r}}}{T_p} \right\rceil C_{p,q,r}, \tag{6.10}$$

where $hp(\tau_i)$ is the set of threads with higher priority than τ_i and executed on the same processor than τ_i, the term $J_{\theta_{p,q,r}}$ being the maximum jitter on the arrival of $\theta_{p,q,r}$, T_p is the period of the task τ_p to which thread $\theta_{p,q,r}$ belongs, and $C_{p,q,r}$ is the WCET of thread $\theta_{p,q,r}$. Note that, as already explained, this jitter is always equal to 0 in time-triggered systems. Equation 6.10 can be solved with a fixed point iteration over r_{τ_i}, where r_{τ_i} is initialised at C_i for the first iteration.

Non-fully stretched tasks: for non-fully stretched tasks, one must consider the sequential and parallel segments independently (Case 2 in Section 4.3). Remember that for each sequential and P/D segment, there exists a synchronization point at the end of the segment, indicating that no thread that belongs to the segment after the synchronisation point can start executing before all threads of the current segment have completed their execution and the associated messages completed their transmission. Therefore, the WCRT of a task τ_i is computed based on the sum of the WCRTs of each segment $\sigma_{i,j}$ (denoted as $r_{\sigma_{i,j}}$):

$$r_{\tau_i} = \sum_{j=1}^{n_i} (r_{\sigma_{i,j}}), \tag{6.11}$$

where $r_{\sigma_{i,j}}$ can be computed as described below for sequential and parallel segments, respectively:

i. *Sequential segments*. Sequential segments are executed on their own processors. Therefore, they do not suffer any interference from other threads. Hence:

$$r_{\sigma_{i,2j+1}} = C_{i,2j+1,1}. \tag{6.12}$$

ii. *Parallel segments*. For a parallel segment $\sigma_{i,2j}$, the WCRT is given by the maximum of the following two values:

a. The sum of the worst-case execution times of the set of threads coalesced in $\tau_i^{stretched}$ (denoted by $CThr_{i,2j}$), which are executed sequentially on their own processor. That is,

$$r_{CThr_{i,2j}} = \sum_{\theta_{i,2j,k} \in \{\sigma_{i,2j} \cap \tau_i^{stretched}\}} C_{i,2j,k}. \tag{6.13}$$

b. The maximum WCRT (denoted as $WR^{\max}_{DP_{i,2j}}$) of each distributed execution paths $DP_{i,2j,k}$ within the parallel segment. The WCRT of a distributed execution path $DP_{i,2j,k}$, is upper-bounded by the sum of the WCRT of its constituting messages $\mu_{i,2j-1,k}$ and $\mu_{i,2j,k}$, and its thread $\theta_{i,j,k}$, i.e.,

$$r_{DP_{i,2j,k}} = r_{\mu_{i,2j-1,k}} + r_{\theta_{i,j,k}} + r_{\mu_{i,2j,k}}, \tag{6.14}$$

under the FTT-SE protocol, $r_{\mu_{i,2j-1,k}}$ and $r_{\mu_{i,2j,k}}$ can be computed using Eq. (6.8) increased by 1 EC (due to the node queuing delay), and Eq. (6.15) can be used for computing the WCRT of the thread $\theta_{i,j,k}$ executed on its remote node:

$$r_{\theta_{i,j,k}} = C_{i,j,k} + \sum_{\theta_{p,q,r} \in hp(\theta_{i,j,k})} \left\lceil \frac{r_{\theta_{i,j,k}} + J_{\theta_{p,q,r}}}{T_p} \right\rceil C_{p,q,r}. \tag{6.15}$$

Therefore, the maximum WCRT experienced by a distributed execution path in $\sigma_{i,2j}$ is:

$$WR^{\max}_{DP_{i,2j}} = \max_{DP_{i,2,j,k} \in \sigma_{i,2j}} \{r_{DP_{i,2j,k}}\}. \tag{6.16}$$

Thus, the WCRT of a parallel segment $\sigma_{i,2j}$, is the maximum between the sum of the execution times of all coalesced threads $CThr_{i,2j}$ (Eq. (6.13)) and the longest distributed execution path $WR^{\max}_{DP_{i,2j}}$ (Eq. (6.16)). That is,

$$r_{\sigma_{i,2j}} = \max\{r_{CThr_{i,2j}}, WR^{\max}_{DP_{i,2j}}\}. \tag{6.17}$$

6.3.2 Event-triggered Systems

In some situations the use of time-triggered systems is not adequate when the process is even-based. In those cases using the FTT-SE network in time-triggered mode can waste considerable amount of bandwidth. Therefore, in some cases the usage of asynchronous features of the FTT-SE may result on smaller response time and on better utilisation of the network bandwidth.

For the case in which the activation of a P/D message or P/D thread is based on the response time of previous processing events, it implies an event-triggered communication pattern which uses asynchronous messages. The fact that, in an event-triggered system, threads and messages are sent on completion of the previous message (or thread) in the fork-join sequence, implies that each thread and message may experience a release jitter $J_{\mu_{i,j,k}}$ and $J_{\theta_{i,j,k}}$ respectively, equal to the difference between the best-case and the worst-case response time of the preceding message (or thread, respectively). As shown by Eq. 6.10 and 6.15, these jitters have an impact on the worst-case response time of the threads. The same is true for messages. Hence, Eq. (6.8) is adapted to consider their release jitter. Note that Eq. 6.10–6.17 remain unchanged.

Only the computation of $r_{\mu_{i,2j-1,k}}$ and $r_{\mu_{i,2j,k}}$ are altered by the release jitters. In fact, using the same reasoning than in (Audsley et al., 1993), it is possible to see that a message $\mu_{p,q,r}$ with a release jitter $J_{\mu_{p,q,r}}$ and interfering with $\mu_{i,j,k}$ may release at most $\lceil \frac{t+J_{\mu_{p,q,r}}}{T_p} \rceil$

message instances in a time window of length t. Therefore, Eq. 6.2 and 6.4 must be modified as follows:

$$Wl_{i,j,k}(t) = \sum_{\forall \mu_{p,q,r} \in SLD_{i,j,k}} \left\lceil \frac{t + J_{\mu_{p,q,r}}}{T_p} \right\rceil M_{p,q,r} + Is_{i,j,k}(t), \tag{6.18}$$

$$Wr_{i,j,k}(t) = \sum_{\forall \mu_{p,q,r} \in RLD_{i,j,k}} \left\lceil \frac{t + J_{\mu_{p,q,r}}}{T_p} \right\rceil M_{p,q,r}, \tag{6.19}$$

thus, by assuming the best-case response time equal to zero for all preceding events, $J_{\theta_{i,j,k}}$ ($J_{\mu_{i,j,k}}$, resp.) is equal to the largest sum of the WCRT of each predecessor, computed by Eq. 6.10–6.19, i.e:

$$J_{\theta_{i,j,k}} = \max_{\forall \mu_{p,q,r} \in predec(\theta_{i,j,k})} \{J_{\mu_{p,q,r}} + WCRT(\mu_{p,q,r})\}, \tag{6.20}$$

$$J_{\mu_{p,q,r}} = \max_{\forall \theta_{i,j,k} \in predec(\mu_{p,q,r})} \{J_{\theta_{i,j,k}} + WCRT(\theta_{i,j,k})\}, \tag{6.21}$$

where, $predec(\theta_{i,j,k})$ ($predec(\mu_{i,j,k})$) is the set of all threads (messages, resp.) which are direct predecessors of thread $\mu_{i,j,k}$ (message $\theta_{i,j,k}$, resp.) in the P/D task τ_i.

6.4 Improved Response Time Analysis for Distributed - Execution Paths

This section presents an improved WCRT analysis for the execution of the distributed execution paths. The improvement is based on a pipeline effect that occurs when simultaneously transmitting P/D messages on an FTT-SE network and executing their respective P/D threads on remote nodes. Consider the following example:

Example 6.3. *Consider a tasks $\tau_1 = ((1, 0.25, 1, 0.25, 1), 9, 10)$ stretched with the DST transformation and mapped onto processors by an arbitrary partitioning algorithm (see Figure 6.5). Also, consider the system architecture depicted in Fig. 6.4. If a message $\mu_{i,j,k}$ is transmitted from the ECU Head-Unit (H-U) to CTRL-2, it has to cross two links in the network; from H-U to SW_1, and from SW_1 to CTRL-2. This is shown in Fig. 6.5. Assume that two threads $\theta_{1,2,8}$ and $\theta_{1,2,9}$ of task τ_1 are assigned to processor CTRL-2, thus, τ_1 sends two messages $\mu_{1,1,8}$ and $\mu_{1,1,9}$ from H-U to CTRL-2. After their remote execution is completed, both threads perform a D-Join operation sending the corresponding messages to their invoker node. One can notice that the transmission of message $\mu_{1,1,9}$ during the D-fork operation is occurring in parallel with the execution of thread $\theta_{1,2,8}$. Also, one can note that the transmissions of messages $\mu_{1,2,8}$ and $\mu_{1,2,9}$ during the D-join operation do not interfere with each other on the uplink $l^u_{\pi_2,SW_1}$.*

This example illustrates the fact that contrarily to what is assumed in Eq. (6.14), the WCRT of a distributed execution path is not simply the sum of the WCRT of $\mu_{i,j-1,k}$,

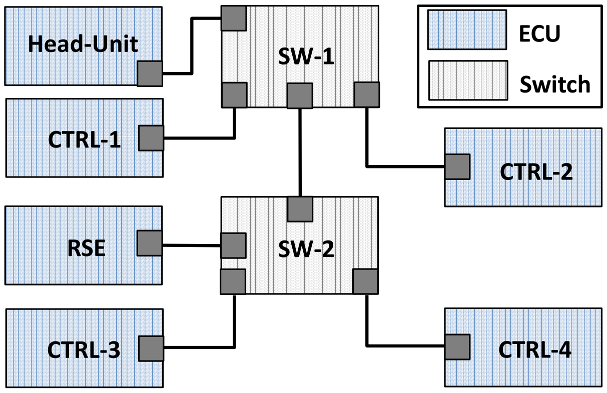

Figure 6.4: Automotive architecture interconnected with an FTT-SE network.

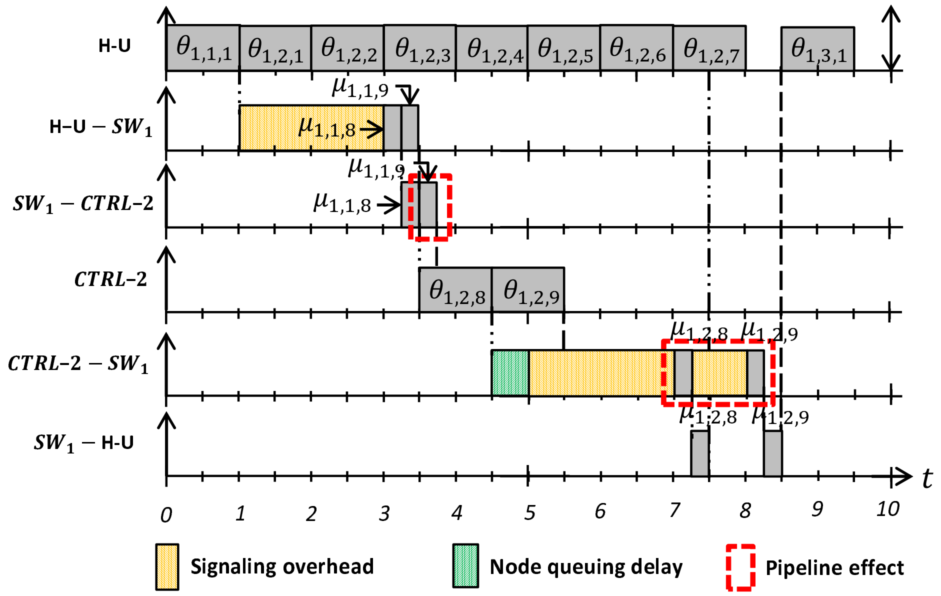
H-U
$\theta_{1,1,1}$ $\theta_{1,2,1}$ $\theta_{1,2,2}$ $\theta_{1,2,3}$ $\theta_{1,2,4}$ $\theta_{1,2,5}$ $\theta_{1,2,6}$ $\theta_{1,2,7}$ $\theta_{1,3,1}$
H-U − SW_1
$\mu_{1,1,9}$
$\mu_{1,1,8}$
SW_1 − $CTRL$-2
$\mu_{1,1,9}$
$\mu_{1,1,8}$
$CTRL$-2
$\theta_{1,2,8}$ $\theta_{1,2,9}$
$CTRL$-2 − SW_1
$\mu_{1,2,8}$ $\mu_{1,2,9}$
SW_1 − H-U
$\mu_{1,2,8}$ $\mu_{1,2,9}$
0 1 2 3 4 5 6 7 8 9 10
t
Signaling overhead
Node queuing delay
Pipeline effect

Figure 6.5: Pipeline effect of a P/D task interconnected with an FTT-SE network.

$\theta_{i,j,k}$ and $\mu_{i,j,k}$. Under the reasonable assumption that the WCML of P/D messages is smaller than or equal to the WCET of their corresponding P/D threads (i.e., $M_{i,j-1,k} \leq C_{i,j,k}$ and $M_{i,j,k} \leq C_{i,j,k}$) and assuming that the priority ordering of the P/D threads is identical to the priority ordering of the associated P/D messages (i.e., if $\theta_{p,q,r} \in hp(\theta_{i,j,k})$, then $\mu_{p,q-1,r} \in hp(\mu_{i,j-1,k})$), an overlap (denoted as $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$) exist on the downlink $l^d_{SW_x,\pi_i}$ connecting the last switch SW_x in the transmission path of a message $\mu_{i,j-1,k}$ to the remote processor node π_i on which $\theta_{i,j,k}$ executes. Similarly, a non-interference (denoted as $OvJ^{l^d_{SW_x}}_{\pi_i}(\mu_{i,j,k})$) occurs during the D-join operation when multiple messages belonging to the same task τ_i. Therefore, the pessimism on the computation of the WCRT of a distributed execution path $r_{DP_{i,j,k}}$ can be reduced.

6.4.1 Overlap on the Downlink

Consider the two following situations:

i. assume that a low priority thread θ^l is executing on a remote processor node π_i. If the execution of a thread of higher priority θ^h is triggered on π_i, θ^l is preempted by θ^h. However, because messages are non-preemptible, the message μ^l that triggered the execution of θ^l must have reached π_i before the message μ^h could start being transmitted, thereby implying that the transmission of μ^h occurred in parallel with the execution of θ^l;

ii. assume a thread of high priority θ^h executing on a remote node processor π_i. If the execution of a lower priority thread θ^l is triggered on π_i, θ^l is delayed until θ^h completes its execution. Similarly to the previous case, because only one message can be transmitted at a time, it implies that the transmission of μ^l occurred in parallel with the execution of θ^h.

Let $IntT(\theta_{i,j,k})$ be the set of jobs that contribute to the WCRT of a thread $\theta_{i,j,k}$ (including the job $\theta_{i,j,k}$ itself) on a remote processor node π_ℓ. And let $IntM(\theta_{i,j,k})$ be the set of messages that contributed to the WCRT of $\mu_{i,j-1,k}$ and triggered the execution of jobs in $IntT(\theta_{i,j,k})$. Extrapolating the two situations discussed above, it is possible to conclude that:

Property 6.1. *Only one message in $IntM(\theta_{i,j,k})$ was not transmitted in parallel with the execution of the jobs in $IntT(\theta_{i,j,k})$. This message is the message of the first job in $IntT(\theta_{i,j,k})$ that started executing on π_ℓ.*

In the worst-case, the message that did not overlap with the response time of $\theta_{i,j,k}$ is the message with the largest WCML in $IntM(\theta_{i,j,k})$. Let $M^{\max}_{i,j,k}$ be the WCML of that message. Then, the overlap $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$ is lower bounded by:

$$OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}) = \sum_{\mu_{p,q,r} \in IntM(\theta_{i,j,k})} \mu_{p,q,r} - M^{\max}_{i,j,k}. \tag{6.22}$$

Therefore, denoting $RT(\mu_{i,j-1,k} + \theta_{i,j,k})$ the response time of a P/D message $\mu_{i,j-1,k}$, and its corresponding thread $\theta_{i,j,k}$ during a D-Fork operation, the following theorem is proved.

Theorem 6.1. *The response time $RT(\mu_{i,j-1,k} + \theta_{i,j,k})$ is upper bounded by:*

$$RT(\mu_{i,j-1,k} + \theta_{i,j,k}) \leq r_{\mu_{i,j-1,k}} + r_{\theta_{i,j,k}} - OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}),$$

where $r_{\mu_{i,j-1,k}}$ and $r_{\theta_{i,j,k}}$ are computed with Eq. 6.8 increased by 1 or 3 ECs (see Section 6.3) and Eq. 6.15, respectively.

Proof. The proof is done by contradiction. Assume that there exists a scenario such that:

$$RT(\mu_{i,j-1,k} + \theta_{i,j,k}) > r_{\mu_{i,j-1,k}} + r_{\theta_{i,j,k}} - OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}). \tag{6.23}$$

It is known that there is an overlap $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$ between the transmission of the messages and the execution of the threads participating to the response time of $\mu_{i,j-1,k}$ and $\theta_{i,j,k}$. Therefore, at least:

$$RT(\mu_{i,j-1,k} + \theta_{i,j,k}) \leq r_{\mu_{i,j-1,k}} + r_{\theta_{i,j,k}} - OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}).$$

This implies that Eq. (6.23) is true iff $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) < OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k})$. The only possible reason for such a situation to happen, is that at least one transmission of a message $\mu^h_{i,j-1,k} \in IntM(\theta_{i,j,k})$ accounted in $OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k})$ does not contribute to $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$. Assume that there is only one such instance[1]. Then,

$$OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) = OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}) - M^h_{i,j-1,k}. \tag{6.24}$$

Two cases must be considered:

i. the thread $\theta^h_{i,j,k}$ triggered by $\mu^h_{i,j-1,k}$ does not interfere with the execution of $\theta_{i,j,k}$. This implies that $RT(\theta_{i,j,k}) \leq r_{\theta_{i,j,k}} - C^h_{i,j,k}$, and because by assumption $C^h_{i,j,k}$

[1] If multiple message instances in $IntM(\theta_{i,j,k})$ do not contribute to $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$, then the reasoning developed in the following of this proof can be applied iteratively by considering one more instance at each iteration.

$\geq M^h_{i,j-1,k}$:

$$\begin{aligned} & RT(\theta_{i,j,k}+\mu_{i,j-1,k}) \\ \leq & r_{\theta_{i,j,k}} - C^h_{i,j,k} + r_{\mu_{i,j-1,k}} - OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) \\ \leq & r_{\theta_{i,j,k}} - M^h_{i,j,k} + r_{\mu_{i,j-1,k}} - OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) \\ \leq & r_{\theta_{i,j,k}} + r_{\mu_{i,j-1,k}} - OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}), \end{aligned}$$

thereby contradicting Eq. (6.23).

ii. the thread $\theta^h_{i,j,k}$ triggered by $\mu^h_{i,j-1,k}$ interferes with the execution of $\theta_{i,j,k}$. Because by Property 6.1, only one message in $IntM(\theta_{i,j,k})$ does not contribute to $OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k})$, thus:

$$OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) = \sum_{\mu_{p,q,r} \in IntM(\theta_{i,j,k})} \mu_{p,q,r} - M^h_{i,j,k}.$$

And using Eq. (6.22)

$$\begin{aligned} OvF^{\pi_i}_{l^d_{SW_x}}(\theta_{i,j,k}) &= OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}) + M^{\max}_{i,j-1,k} - M^h_{i,j,k} \\ &\geq OvF^{*}_{\pi_i,l_{SW_x}}(\theta_{i,j,k}), \end{aligned}$$

which contradicts Eq. (6.24) and therefore Eq. (6.23).

Consequently, Eq. (6.23) can never be true. □

6.4.2 Non-interference on the Uplink

As illustrated on Fig. 6.5, if all the P/D threads $\theta_{i,j,k}$ of a same parallel segment $\sigma_{i,j}$ share the same priority, then they do not preempt each other when executing on the same remote node π_ℓ. Consequently, the messages $\mu_{i,j,k}$ sent by those threads from π_ℓ to their invoker processor, start their transmissions at least $C_{i,j,k}$ time units apart. Because by assumption the WCML $M_{i,j,k}$ is smaller than or equal to the WCET $C_{i,j,k}$ of the threads triggering their execution, a non-interference effect $OvJ^{l^d_{SW_x}}_{\pi_i}(\mu_{i,j,k})$ between the messages of the same P/D segment sent from the same remote node occurs during the D-join operation. This effect is given by:

$$OvJ^{l^d_{SW_x}}_{\pi_i}(\mu_{i,j,k}) = \sum_{\substack{\forall \mu_{i,j,p} \in \pi_i \\ p \neq k}} M_{i,j,p}, \tag{6.25}$$

where $\mu_{i,j,p} \in \pi_i$ means that the message $\mu_{i,j,p}$ has π_i as service node. This gives the following theorem:

Theorem 6.2. *The response time $RT(\mu_{i,j,k})$ of a P/D message $\mu_{i,j,k}$ during a D-Join operation is upper bounded by:*

$$RT(\mu_{i,j,k}) \leq r_{\mu_{i,j,k}} - OvJ_{\pi_i}^{l^d_{SW_x}}(\mu_{i,j,k}).$$

Proof. Assume that only two messages[2] of the same segment $\sigma_{i,j}$ are sent from the remote processor π_ℓ. Let us denote them by $\mu_{i,j,1}$ and $\mu_{i,j,2}$ and assume that $\mu_{i,j,1}$ is the first to be triggered. Therefore, $\mu_{i,j,2}$ does not participate to the response time of $\mu_{i,j,1}$, yet Eq. (7) assumes that $\mu_{i,j,2}$ interferes with $\mu_{i,j,1}$. Therefore:

$$RT(\mu_{i,j,1}) \leq r_{\mu_{i,j,1}} - M_{i,j,1} = r_{\mu_{i,j,1}} - OvJ_{\pi_i}^{l^d_{SW_x}}(\mu_{i,j,1}).$$

Two cases must be considered for $\mu_{i,j,2}$:

i. $\mu_{i,j,1}$ and $\mu_{i,j,2}$ are triggered in the same EC. Because $\mu_{i,j,2}$ was triggered at least $M_{i,j,1}$ time units after $\mu_{i,j,1}$, the node queuing delay cannot be longer than $|EC| - M_{i,j,1}$. Since $r_{\mu_{i,j,2}}$ always considers a node queuing delay of $1 \times |EC|$ (see Section VI-A), there is:

$$RT(\mu_{i,j,2}) \leq r_{\mu_{i,j,2}} - M_{i,j,2} = r_{\mu_{i,j,2}} - OvJ_{\pi_i}^{l^d_{SW_x}}(\mu_{i,j,2}).$$

ii. $\mu_{i,j,1}$ and $\mu_{i,j,2}$ are triggered in different ECs. If $\mu_{i,j,1}$ already completed its transmission, then it does not interfere with $\mu_{i,j,2}$ and the theorem obviously holds. Otherwise, if $\mu_{i,j,1}$ is still waiting to be transmitted when $\mu_{i,j,2}$ is triggered, then it means that $\mu_{i,j,1}$ was delayed by higher priority messages for a time at least equal to the length LW of its transmission window. Those higher priority messages cannot interfere with $\mu_{i,j,2}$ anymore and because $LW \geq M_{i,j,1}$, the theorem holds for $\mu_{i,j,2}$.

□

In conclusion, combining the results of Theorem 6.1 and 6.2, it is possible to improve the WCRT of a distributed execution path (Eq. (6.14)) as follows:

$$WR(DP_{i,2j,k}) = r_{\mu_{i,2j-1,k}} + r_{\theta_{i,2j,k}} + r_{\mu_{i,2j,k}} - OvF^*_{\pi_i,l_{SW_x}}(\theta_{i,j,k}) - OvJ_{\pi_i}^{l^d_{SW_x}}(\mu_{i,j,k}).$$

6.5 Numerical Example

This section shows how to apply the results presented in previous sections, showing (i) how the technique introduced in Section 6.4 reduces the pessimism of computing the WCRT of P/D tasks, and (ii) the usefulness of the DST algorithm (see Section 4.3).

[2] If more than two messages of the same segment should be sent from the same remote processor, the proof still holds by applying the argumentation iteratively, adding one more message at each iteration.

Table 6.1: Automotive application characteristics.

App	Type	Period	WCET(F/J)	WCET(Rem)	WCML	Invok. Node	Rem. Node(a)	Rem. Node(b)
τ_1	Control	2000 μs	-	30 μs	10 μs	Head-Unit	CRTL-1	CRTL-1
τ_2	Control	2000 μs	-	30 μs	10 μs	Head-Unit	CTRL-2	CTRL-2
τ_3	Control	2000 μs	-	30 μs	10 μs	Head-Unit	CTRL-3	CTRL-3
τ_4	Video	4000 μs	-	-	-	-	-	-
-	Vid-Msg	4000 μs	-	-	30 μs	RSE	-	-
-	Vid-Thrs	4000 μs	200 μs	200 μs	-	RSE	CTRL- 1-4	CTRL- 1-2
τ_5	Audio	84000 μs	-	400 μs	30 μs	RSE	CTRL-3	CTRL-3

Our examples are based on the research presented in (Lim et al., 2011), however the base scenario and values have been modified with the intention of stressing the system by increasing the load. It is considered a 100 Mb/s Ethernet network and a double-star topology as the one presented in Fig. 6.4. That configuration simulates the location of the switches in a car (at the front and at the rear of the car). It is considered that messages are transmitted using the FTT-SE protocol. The Head-unit ECU operates a set of ECUs (CRTL-1, CRTL-2, CRTL-3, and CRTL-4). A Rear Seat Entertainment (RSE) system which manages audio and video applications is also part of the system. It is assumed that the video application can be processed in parallel. Each EC has a length of 500 μs, and the reserved bandwidth for the transmission of synchronous messages in each EC is equal to 80%, $\Delta = 1\,\mu$s, and the $SFD_{i,j,k} = \max(WCML)$. All the traffic is sent using the reserved bandwidth for synchronous messages within the FTT-SE protocol.

Consider that control applications τ_1, τ_2, and τ_3 are sequential and have an origin in the Head-Unit and destination in CRTL-1, CRTL-2, and CRTL-3, respectively. All control messages have the same WCML of 10 μs, they execute on their respective remote processor with a WCET of 30 μs, and they have a periodicity of 2000 μs. Infotainment applications τ_4 and τ_5 are video and audio applications, respectively. The origin of τ_4 and τ_5 is the RSE system. It is considered that τ_4 is a P/D tasks with a periodicity of 4000 μs, which is divided in 24 threads (with a remote WCET of 200 μs each) and 48 messages (with a WCML of 30 μs each). It is considered a D-fork and D-join execution time of 200 μs. 6 P/D threads are assigned to each ECU (from CTRL-1 to CTRL-4). The audio application τ_5 has only one remote thread assigned to CRTL-3 with a WCML of 30 μs, a WCET of 400 μs, and a periodicity of 8400 μs. It is assumed implicit deadlines ($D_i = T_i$) for all applications. A summary of the applications characteristics is shown in Table 6.1.

Figure 6.6, shows the WCRT of the applications when calculated with Eq. (6.17). The dark grey stack represents the gain of considering the pipeline effect (see Section 6.4), this gain can be subtracted from the WCRT given by Eq. (6.17), to compute an improved WCRT for a distributed execution path. For this example, the pipeline effect that can be subtracted from Eq. (6.17) is of 6% and 7% for applications τ_4 and τ_5, respectively.

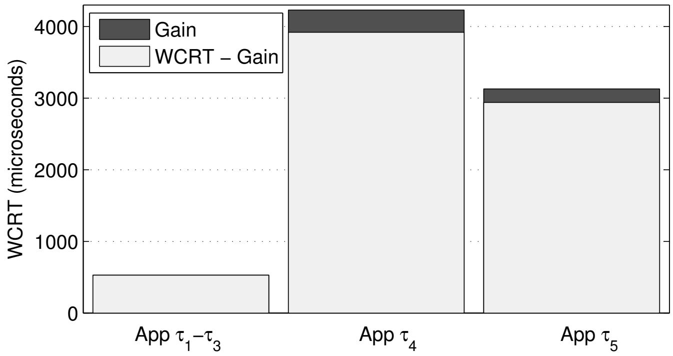

Figure 6.6: Improved end-to-end WCRT considering the pipeline effect.

Note that in this example, the pipeline effect for application τ_4, represents the difference between meeting or missing its deadline (equal to 4000 μs).

Now, it is showed the usefulness of the DST algorithm. It is possible to see that executing τ_4 sequentially (in a single node) would lead to a deadline miss (200 μs $\times$24 threads $> T_4$). However, by using the spare capacity of RSE ECU (invoker node) it is possible to stretch τ_4 with the DST and keep as many threads as possible for local execution, thus, avoiding transmitting over the network. Thus, after applying the DST algorithm, 18 threads out of 24 threads are kept for execution on the RSE ECU. Therefore, 6 P/D threads are allocated to CTRL-2, satisfying the execution requirements of τ_4.

Similarly to Figure 6.6, the dark gray stack in Figure 6.7 represents the gain of considering the pipeline effect. Figure 6.7 shows the WCRT after applying the DST algorithm to the same applications as in Figure 6.6. For this example, the pipeline effect for τ_4 is of 13%. Note, that when compared to Figure 6.6 the end-to-end WCRT is shorter. This effect is related to the lower load of messages in the network after applying the DST transformation.

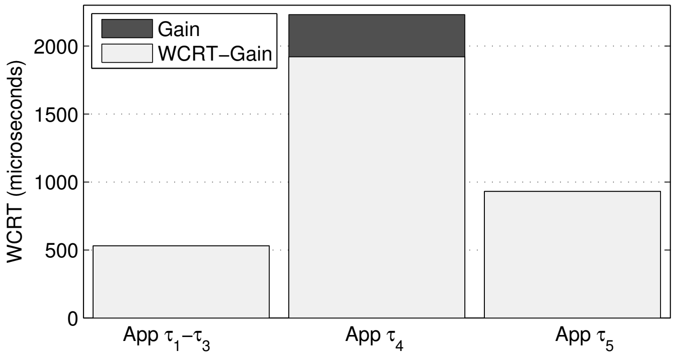

Figure 6.7: End-to-end WCRT when using the DST algorithm.

6.6 Assessing of the Gain in the Pipeline Effect

The pipeline effect for τ_4 in Figure 6.6 and in Figure 6.7 is the same (310 μs), because Eq. (6.22) and Eq. (6.25) depend on the subset of messages that are transmitted in parallel with the execution of threads in the remote processors. By observing the gain of τ_4 on those Figures, it is possible to notice that the percentage of the gain varies with respect to the WCRT of those applications. When not considering the DST the gain is equal to 6%, but when using the DST the gain increases to 13%, since there are fewer messages being transmitted to the same nodes.

In order to see the variation of the gain, let us consider the same system characteristics (systems architecture and applications) as in Section 6.5. Also, consider a load distribution as the one presented in Figure 6.6. In the experiment presented in Table I, application τ_4 is divided in 24 P/D threads (with a remote WCET of 200 μs each) and 48 messages (with a WCML of 30 μs each). We vary the number of P/D threads as:

1. 12 P/D threads (with a remote WCET of 400 μs each) and 24 messages (with a WCML of 60 μs each), 3 threads are executed in each CTRL node;
2. 24 threads (as before); and
3. 48 P/D threads (with a remote WCET of 100 μs each) and 96 messages (with a WCML of 15 μs each), 12 threads are assigned for execution to each CRTL node.

We consider a D-fork and D-join execution time of 200 μs.

Figure 6.8 shows the WCRT of application τ_4 varying the number of P/D threads and therefore its WCET (WCML for messages). The dark grey stacks represent the gain of considering the pipeline effect and the light gray stacks represent the gain subtracted from the WCRT given by Eq. (6.17). It is possible to see for the three cases:

1. 12 P/D threads, the gain is of 250 μs which represents 6% of the WCRT - 4230 μs;
2. 24 P/D threads, the gain is 250 μs which represents 7% of the WCRT (4230 μs); and
3. 48 P/D threads, the gain is of 340 μs which represents 8% of the WCRT (4230 μs).

Thus, it is possible to conclude that the gain of the pipeline effect depends on the WCML of the messages and the amount of messages that contribute to Eq. (6.22) and Eq. (6.25), and that the proportion of such a gain with respect to the WCRT applications in the system (P/D tasks) depends on all the elements that contribute to its WCRT Eq. (6.1)-(6.21).

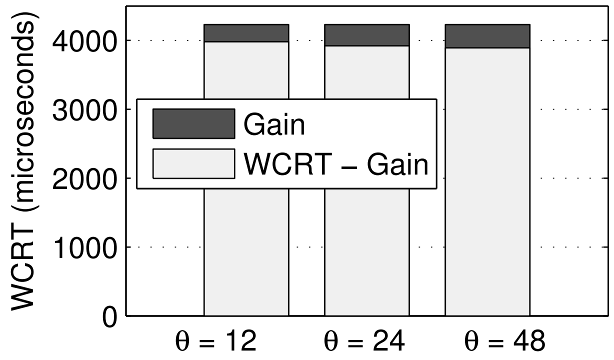

Figure 6.8: Variation over the number of P/D threads.

6.7 Summary

In this chapter it was presented a holistic timing analysis for the computation of the Worst-Case Response Time (WCRT) of P/D tasks when transformed by the DST algorithm. Both synchronous and asynchronous communication patterns were considered. The analysis to consider an FTT-SE transmission network has been extended, and an analysis technique for reducing the pessimism when computing the WCRT by considering a pipeline in the transmission over the network and the execution on the processors has been proposed. Note that the two presented techniques can be used jointly, but they can also be used independently.

The following publications are related to the work presented in this chapter:

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Towards holistic analysis for fork-join parallel/distributed real-time tasks. In *Work in Progress Session (ECRTS), 2014 26th Euromicro Conference on Real-Time Systems*, pages 21–24, July 2014a.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Holistic analysis for fork-join distributed tasks supported by the ftt-se protocol. In *Factory Communication Systems (WFCS), 2015 11th World Conference on*, May 2015.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. An Improved Holistic Analysis for Fork-Join Parallel Distributed Real-Time Tasks using the FTT-SE Protocol. Selected as candidate paper for a Special Section of *IEEE Transactions of Industrial Informatics* (revision process: second revision).

Chapter 7

Allocation of P/D Tasks in Multi-core Architectures supported by FTT-SE Protocol

7.1 Introduction

Works related to the problem of task partitioning and priority assignment on hard real-time distributed systems are presented in Section 2.4. This chapter considers the problem of allocating P/D tasks onto *distributed multi-core nodes* connected through a FTT-SE network. This is an extension for distributed multi-core systems of the work presented in (Garibay-Martínez et al., 2014b, 2013b, 2015a) (see Chapter 4 and Chapter 5).

The chapter discusses the system requirements and presents a set of formulations based on a *constraint programming* approach. A constraint programming approach allows to express the relations between variables in the form of constraints. This approach is guaranteed to find a feasible solution, if one exists, in contrast to other approaches based on heuristics (e.g., the ones presented in Chapter 5). The work presented in this chapter is supported on results presented in (Metzner and Herde, 2006), with the following specific characteristics:

i. this work models P/D tasks executing over a distributed multi-core architecture, and;

ii. it considers messages being transmitted through a FTT-SE network.

Furthermore, similar approaches based on constraint programming have shown that it is possible to obtain solutions for these type of formulations in reasonable time (Zhu et al., 2013; Metzner and Herde, 2006).

The constraint programming formulation is introduced in Section 7.2 which addresses issues related to consider P/D tasks, the constraints related to the partitioned approach and it also addresses the modelling constraints of the FTT-SE network. Finally, a summary of the chapter is provided in Section 7.3.

7.1.1 Chapter Considerations

This chapter considers that the set π is composed of m multi-core nodes to execute tasks. Each node π_p ($p \in \{1,\dots,m\}$) is composed of m_p identical cores $\pi_{p,s}$ ($s \in \{1,\dots,m_p\}$). The total number of cores in the system is therefore equal to $m_{tot} = \sum_{\pi_{p,s} \in \pi} m_p$. The processing nodes are interconnected by an FTT-SE network ϖ.

Note that the decision variables of the formulation are type setted. For example, the deadline of a segment $\sigma_{i,j}$ is denoted as $d_{i,j}$, but in this formulation is a decision variable of the problem, therefore, it is denoted as $\mathsf{d_{i,j}}$.

7.2 Constraint Programming Formulation

As discussed in Chapter 5. The problem of task allocation can be seen as a two-sided problem:

i. finding the partitioning of threads and messages onto the processing elements of the distributed system, and

ii. finding the priority assignment for the threads and messages in that partition so that the real-time tasks and messages complete their execution before reaching their respective end-to-end deadlines.

In this section the system requirements are analysed and a constraint programming formulation is provided.

7.2.1 P/D Tasks

In a similar manner as in Chapter 4, in this chapter threads composing a P/D task are transformed into a set of tasks with constrained deadlines. This transformation is based on the imposition of a set of *artificial intermediate deadlines* (denoted as $\mathsf{d_{i,j}}$), to threads $\theta_{i,j,k}$ and messages $\mu_{i,j,k}$ in each segment $\sigma_{i,j}$ composing a task τ_i.

The following two constraints must be associated to each intermediate deadline $\mathsf{d_{i,j}}$:

- Even if all threads execute in parallel, the relative deadline $\mathsf{d_{i,j}}$ cannot be smaller than the maximum WCET of a thread in that segment, thereby imposing that:

$$\bigwedge_{\forall \tau_i \in T} \bigwedge_{\forall \sigma_{i,j} \in \tau_i} \mathsf{d_{i,j}} \geq \max_{k \in \sigma_{i,j}} \{C_{i,j,k}\}. \tag{7.1}$$

- The total execution granted to all segments constituting a task τ_i must be smaller or equal than the relative deadline of τ_i, that is:

$$\bigwedge_{\forall \tau_i \in T} \sum_{\forall \sigma_{i,j} \in \tau_i} \mathsf{d_{i,j}} \leq D_i. \tag{7.2}$$

Thus, the artificial deadline $\mathsf{d_{i,j}}$ is the maximum time that threads of a segment $\sigma_{i,j}$ are permitted to take, from the moment they are released, to the moment they complete their execution. Therefore, the problem can be formulated as to find the artificial intermediate deadlines $\mathsf{d_{i,j}}$ for every segment $\sigma_{i,j}$, in a way that the Worst-Case Response Time (WCRT) of threads $\theta_{i,j,k}$ (and messages $\mu_{i,j,k}$) is smaller or equal to their respective intermediate deadlines and the sum of such intermediate deadlines is smaller or equal to its end-to-end deadline D_i. More constraints are presented in Sections 7.2.2 and Section 7.2.3.

7.2.2 Fully-Partitioned Distributed Multi-core Systems

It is assumed a fixed-priority fully-partitioned scheduling algorithm, it is also assumed that each core in the system (regardless the processing node they are part of) is assigned a unique identifier in the interval $[1, m_{tot}]$. An integer variable $\Pi_{\theta_{\mathsf{i,j,k}}}$ is defined, indicating the identifier of the core on which the thread $\theta_{i,j,k}$ is mapped. By definition of the core identifier, the following constraints apply:

$$\Pi_{\theta_{\mathsf{i,j,k}}} > 0, \tag{7.3}$$

$$\Pi_{\theta_{\mathsf{i,j,k}}} \leq m_{tot}. \tag{7.4}$$

A constraint of the P/D task model is that all sequential segments of a task τ_i must execute on the same core $\pi_{r,s}$. This is imposed by Eq. (7.5):

$$\bigwedge_{\forall \theta_{i,2j+1,1} \in T} \bigwedge_{\forall \theta_{i,2b+1,1} \in T} \Pi_{\theta_{\mathsf{i,2j+1,1}}} = \Pi_{\theta_{\mathsf{i,2b+1,1}}}. \tag{7.5}$$

The variable $\mathsf{p_{i,j,k}}$ denotes the priority of a thread $\theta_{i,j,k}$. Although $\mathsf{p_{i,j,k}}$ could be a decision variable of the problem for which the solver should find a valid value, in a concern of drastically reducing the number of variables and therefore the complexity of the problem, one may also assume that priorities are assigned using DM (Leung and Whitehead, 1982). In that case $\mathsf{p_{i,j,k}} = \mathsf{d_{i,j}}$, and $\mathsf{p_{i,j,k}}$ can be omitted in the description of the problem. Yet, it is necessary to evaluate if a certain partitioning leads to a valid solution. It is known from (Joseph and Pandya, 1986), that the worst-case response time $\mathsf{r_{i,j,k}}$ of an independent thread $\theta_{i,j,k}$ scheduled with a preemptive fixed-priority scheduling algorithm, is given by:

$$\mathsf{r_{i,j,k}} = C_{i,j,k} + \sum_{\theta_{a,b,c} \in \mathsf{HP_{i,j,k}}} \left\lceil \frac{\mathsf{r_{i,j,k}}}{T_a} \right\rceil C_{a,b,c},$$

where $\mathsf{HP_{i,j,k}}$ is the set of threads with higher or equal priority than $\theta_{i,j,k}$, and executing on the same core than $\theta_{i,j,k}$.

One of the challenges of using a constraint programming approach it to formulate conditions without complex operators (e.g., ceiling functions). Therefore, the previous example can be modelled in the constraint problem as:

$$\bigwedge_{\forall \theta_{i,j,k} \in T} \mathsf{r_{i,j,k}} = C_{i,j,k} + \sum_{\forall \theta_{a,b,c} \in T} \mathsf{IHP^{a,b,c}_{i,j,k}}, \tag{7.6}$$

where $\mathsf{IHP^{a,b,c}_{i,j,k}}$ is the interference caused by a thread $\theta_{a,b,c}$ on $\theta_{i,j,k}$.

The higher priority relation is represented by the following boolean variable:

$$\mathsf{p^{a,b,c}_{i,j,k}} = \begin{cases} 1 & \text{if } \theta_{a,b,c} \text{ has higher priority than } \theta_{i,j,k} \text{ (i.e., } \mathsf{p_{i,j,k}} \leq \mathsf{p_{a,b,c}}\text{)}, \\ 0 & \text{otherwise.} \end{cases}$$

Because $\Pi_{\theta_{\mathsf{i,j,k}}} = \Pi_{\theta_{\mathsf{a,b,c}}}$ indicates that the thread $\theta_{i,j,k}$ and the thread $\theta_{a,b,c}$ execute on the same core. Thus, the interference over a thread $\theta_{i,j,k}$ depends of two cases, if they execute in the same core, or not. This is expressed as:

$$\bigwedge_{\forall \theta_{i,j,k} \in T} \bigwedge_{\forall \theta_{a,b,c} \in T} \mathsf{IHP^{a,b,c}_{i,j,k}} = \begin{cases} \mathsf{I^{a,b,c}_{i,j,k}} \times C_{a,b,c} & \text{if } \Big((\mathsf{p^{a,b,c}_{i,j,k}} = 1) \wedge (\Pi_{\theta_{\mathsf{i,j,k}}} = \Pi_{\theta_{\mathsf{a,b,c}}}) \Big), \\ 0 & \text{otherwise,} \end{cases} \tag{7.7}$$

where $\mathsf{I^{a,b,c}_{i,j,k}}$ is the number of preemptions a thread $\theta_{i,j,k}$ suffers from a thread $\theta_{a,b,c}$. Since $\mathsf{I^{a,b,c}_{i,j,k}}$ is an integer, the ceiling operator can be rewritten as follows:

$$\left\lceil \frac{\mathsf{r_{i,j,k}}}{T_a} \right\rceil = \mathsf{I^{a,b,c}_{i,j,k}} \quad \Longrightarrow \quad \frac{\mathsf{r_{i,j,k}}}{T_a} \leq \mathsf{I^{a,b,c}_{i,j,k}} < \frac{\mathsf{r_{i,j,k}}}{T_a} + 1,$$

thereby, leading to the following constraints:

$$\begin{aligned} \bigwedge_{\forall \theta_{i,j,k} \in T} \bigwedge_{\forall \theta_{a,b,c} \in T} (\Pi_{\theta_{\mathsf{i,j,k}}} = \Pi_{\theta_{\mathsf{a,b,c}}}) \quad \rightarrow \quad & (\mathsf{I^{a,b,c}_{i,j,k}} \times T_a \geq \mathsf{r_{i,j,k}}) \\ & \wedge \Big((\mathsf{I^{a,b,c}_{i,j,k}} - 1) \times T_a < \mathsf{r_{i,j,k}} \Big), \end{aligned} \tag{7.8}$$

$$\bigwedge_{\forall \theta_{i,j,k} \in T} \bigwedge_{\forall \theta_{a,b,c} \in T} (\Pi_{\theta_{\mathsf{i,j,k}}} \neq \Pi_{\theta_{\mathsf{i,j,k}}}) \quad \rightarrow \quad \mathsf{I^{a,b,c}_{i,j,k}} = 0. \tag{7.9}$$

Furthermore, in the P/D task model, some threads within a P/D segment may be executed on remote nodes. Consequently, for each such thread $\theta_{i,j,k}$, two messages $\mu_{i,j-1,k}$ and $\mu_{i,j,k}$ are transmitted between the invoker and remote node. That is, a distributed execution path is generated ($\mu_{i,j-1,k} \rightarrow \theta_{i,j,k} \rightarrow \mu_{i,j,k}$).

$\mathsf{NV}(\theta_{\mathsf{i,j,k}})$ is a function denoting to which node π_q a thread $\theta_{i,j,k}$ has been assigned. Then, $\mathsf{NV}(\theta_{\mathsf{i,j,k}}) = \mathsf{NV}(\theta_{\mathsf{a,b,c}})$ indicates that the threads $\theta_{i,j,k}$ and $\theta_{a,b,c}$ execute on the same node, in which case no message is transmitted through the network. However, if $\mathsf{NV}(\theta_{\mathsf{i,j,k}}) \neq \mathsf{NV}(\theta_{\mathsf{a,b,c}})$, two messages $\mu_{i,j-1,k}$ and $\mu_{i,j-1,k}$ are generated. Thus, the WCRT $\mathsf{r}_{\mathsf{DP}_{\mathsf{i,j,k}}}$ of a distributed execution path $DP_{i,j,k}$ is calculated as follows:

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \; \bigwedge_{\forall \theta_{i,j,k} \in T} \mathsf{r}_{\mathsf{DP}_{\mathsf{i,j,k}}} = \begin{cases} \mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j-1,k}} + \mathsf{r}_{\mathsf{i,j,k}} + \mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j,k}} & \text{if } \mathsf{NV}(\theta_{\mathsf{i,j,k}}) \neq \mathsf{NV}(\theta_{\mathsf{a,b,c}}), \\ \mathsf{r}_{\mathsf{i,j,k}} & \text{otherwise,} \end{cases} \tag{7.10}$$

where $\mathsf{r}_{\mathsf{i,j,k}}$ is the WCRT of thread $\theta_{i,j,k}$ obtained with Eq. (7.6), and $\mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j-1,k}}$ and $\mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j,k}}$ are the WCRTs of messages $\mu_{i,j-1,k}$ and $\mu_{i,j,k}$ respectively, obtained with a network dependent analysis (as the one presented in Section 7.2.3). Similarly to previous chapters, in this chapter it is considered the network analysis presented Section 6.2 for FTT-SE networks.

For a partition of tasks τ_i to be considered a valid solution (all deadlines are met), the following condition has to be respected:

$$\bigwedge_{\forall \theta_{i,j,k} \in T} \mathsf{r}_{\mathsf{DP}_{\mathsf{i,j,k}}} \leq \mathsf{d}_{\mathsf{i,j}}. \tag{7.11}$$

7.2.3 FTT-SE Protocol

The communications within a FTT-SE network are done based on fixed duration slots called *Elementary Cycles* (ECs). The construction of the EC schedule is done by keeping updated tables for synchronous (i.e., periodic) and asynchronous (i.e., sporadic) messages. The scheduler applies a scheduling policy (e.g., Deadline Monotonic) over these tables, generating the ready queues for transmission for that EC. This process is repeated until no other message fits in its respective scheduling window for that EC (i.e., considering all messages from higher to lower priority). More details about the FTT-SE protocol can be found in Section 6.2. For building the ECs it is important to consider:

i. the *architecture of the distributed system*: for convenience, in this chapter the system architecture is represented as an *adjacency-matrix* of a graph $G = (V, E)$. The set $V = \{v_1, \ldots, v_{|V|}\}$ of vertices v_i represents the set of switches in ϖ and the set of nodes in π, and the set $E = \{(v_1, v_2), \ldots, (v_{|V|-1}, v_{|V|})\}$ of edges (v_i, v_j), represent the communication links, from nodes to switches, from switches to nodes or between switches. The main reason behind that is the use of the Breadth First Search (BFS) algorithm. The architectural model must include the full-duplex transmission links. Note that:

(a) direct links between nodes do not exist,

(b) links are directed; that is, (v_i, v_j) and (v_j, v_i) represent two different links, and

(c) the network is full-duplex; that is, if (v_i, v_j) is part of the graph, then (v_j, v_i) is also part of the graph.

Thus, the adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{i,j})$ such that:

$$a_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise,} \end{cases}$$

depending on the partitioning of threads onto the nodes π_l of the system, there exists a set $\mathsf{PN}_{\mu_{\mathsf{i,j,k}}} \subseteq V$ containing the vertices (i.e., switches) that a message $\mu_{i,j,k}$ traverses during a D-fork or a D-join operation. For determining $\mathsf{PN}_{\mu_{\mathsf{i,j,k}}}$, the BFS Algorithm (Cormen et al., 2001) is used for each message $\mu_{i,j,k}$. The BFS inputs are: the matrix A (representing the system architecture), the origin vertex (invoker core/remote core), and the destination vertex (the remote core/invoker core). The BFS finds the shortest path from the origin node to the destination node (See Algorithm 3.5). Therefore, the BFS algorithm finds the switches that a message $\mu_{i,j,k}$ crosses during a D-fork or a D-join operation. The set $\mathsf{PN}_{\mu_{\mathsf{i,j,k}}}$ is required for computing the WCRT of a message $\mu_{i,j,k}$ in the FTT-SE network.

ii. the *switching delays*: it is considered a switching delay (denoted as $\mathsf{SD}_{\mathsf{i,j,k}}$) when a message $\mu_{i,j,k}$ crosses a switch SW_x. $\mathsf{SD}_{\mathsf{i,j,k}}$ has two components, the switch relaying latency (denoted as Δ), which has a constant value related to the specifications of the switch, and the Store-and-Forward Delay (denoted as $\mathsf{SFD}_{\mathsf{i,j,k}}$), i.e., $\mathsf{SD}_{\mathsf{i,j,k}} = \mathsf{SFD}_{\mathsf{i,j,k}} + \Delta$. However, for each EC, only the maximum switching delay $\mathsf{SD}_{\mathsf{i,j,k}}$ is considered.

iii. the *EC windows*: the EC is subdivided into time slots for transmitting different types of traffic (e.g. synchronous window, asynchronous window, etc.). Thus, one must consider the length of the specific transmission window for each type of traffic (denoted as LW). The length of such a window is the reserved bandwidth for transmission in that EC, and cannot be exceeded when transmitting messages within the FTT-SE protocol. This is modeled by the *request bound function* in Eq. (7.12), and the *supply bound function* in Eq. (7.17), presented in the following.

7.2.3.1 Response Time Analysis for FTT-SE networks.

Depending on a given partition, it is required to calculate the WCRT of the messages in the network to verify if the condition in Eq. (7.11) is respected. The work presented in (Ashjaei et al., 2013) is used for the computation of the WCRT of messages within the FTT-SE protocol, with a slight modification.

The **request bound function** (denoted as $\mathsf{rbf_{i,j,k}(t)}$) represents the maximum transmission requirements generated by a message $\mu_{i,j,k}$ and all its higher priority messages during an interval $[0,t]$. The $\mathsf{rbf_{i,j,k}(t)}$ is computed as:

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \mathsf{rbf_{i,j,k}(t)} = M_{i,j,k} + \mathsf{sn_{i,j,k}} \times \mathsf{SFD_{i,j,k}} + \mathsf{Wl_{i,j,k}(t)} + \mathsf{Wr_{i,j,k}(t)}, \tag{7.12}$$

where, $\mathsf{sn_{i,j,k}}$ is the number of switches that a message $\mu_{i,j,k}$ traverses from the origin node to its destination node, $\mathsf{Wl_{i,j,k}(t)}$ is the *"Shared Link Delay"*, and $\mathsf{Wr_{i,j,k}(t)}$ is the *"Remote Link Delay"*, which are explained below.

Shared Link Delay. The transmission of a message $\mu_{i,j,k}$ may be delayed by all the higher priority messages that share a link with $\mu_{i,j,k}$. However, such interference occurs only once, so messages that caused such interference on a previous link are excluded from the analysis for the next links. Also, when building the schedule for each EC, the scheduler considers the maximum switching delay $\mathsf{SD_z}$ (see Eq. (7.14)), only once. Therefore, $\mathsf{Wl_{i,j,k}(t)}$ is computed by separating the interference of messages from the *switching-delay-effect* (denoted as $\mathsf{Is_{i,j,k}(t)}$) for each EC. The shared link delay is computed in (7.13):

$$\mathsf{Wl_{i,j,k}(t)} = \sum_{\forall \mu_{a,b,c} \in \mathsf{SLD_{i,j,k}}} \left\lceil \frac{t}{T_a} \right\rceil M_{a,b,c} + \mathsf{Is_{i,j,k}(t)}, \tag{7.13}$$

where $\mathsf{SLD_{i,j,k}} = \{\forall \mu_{a,b,c} : \mu_{a,b,c} \neq \mu_{i,j,k} \wedge (\mathsf{PN}_{\mu_{\mathsf{i,j,k}}} \cap \mathsf{PN}_{\mu_{\mathsf{a,b,c}}} \neq 0) \wedge \mu_{a,b,c} \in \mathsf{hp}(\mu_{\mathsf{i,j,k}}) \wedge \mu_{a,b,c} \in WT(\mu_{i,j,k})\}$, where, $\mathsf{hp}(\mu_{\mathsf{i,j,k}})$ is the set of messages with priority higher or equal than $\mu_{a,b,c}$ and $WT(\mu_{i,j,k})$ is the set of messages that are scheduled in the same window as $\mu_{a,b,c}$ (i.e. the synchronous or the asynchronous window). The set $\mathsf{hp}(\mu_{\mathsf{i,j,k}})$ for messages $\mu_{i,j,k}$ in Eq. (7.13), as well as the ceiling function, can be formulated in a similar manner as in Section 7.2.2.

For computing the switching-delay-effect $\mathsf{Is_{i,j,k}(t)}$, it is needed to compute an upper bound on the number of switching delays ($\mathsf{SD_{i,j,k}}$) from each message that contributes to Eq. (7.13), at time t. In (Ashjaei et al., 2013), depending on time t, a number of switching delays are inserted into an array whenever a message crosses a switch in the network. The array is sorted in order to consider the maximum switching delays only. A sorting operation is not amenable to optimization solvers. Therefore, a simpler upper bound with the cost of slightly increment the pessimism is introduced.

The number of ECs in an interval $[0,t]$ is given by: $\mathsf{z}(\mathsf{t}) = \lceil \frac{t}{EC} \rceil$ (the ceiling function, can be formulated as in Section 7.2.2), thus, in order to consider the worst-case scenario for the computation of the WCRT, the maximum switching delay ($\mathsf{SD}^{\max}_{\mathsf{i,j,k}}$) for each message that contributes to Eq. (7.13) it is considered, and computed as:

$$\mathsf{SD}^{\max}_{\mathsf{i,j,k}} = \max_{\forall \mu_{a,b,c} \in \mathsf{SLD}_{\mathsf{i,j,k}}} \{\mathsf{SFD}_{\mathsf{i,j,k}} + \Delta\}. \tag{7.14}$$

Then, the maximum switching delay is multiplied by the number of ECs at time t (given by $\mathsf{z}(\mathsf{t})$). Thus, the switching-delay-effect is computed as:

$$\mathsf{Is}_{\mathsf{i,j,k}} = \mathsf{SD}^{\max}_{\mathsf{i,j,k}} \times \mathsf{z}(\mathsf{t}). \tag{7.15}$$

Remote Link Delay. A message $\mu_{i,j,k}$ can be blocked by other higher priority messages even if they do not share a transmission link. Thus, a higher priority message can delay a lower priority message even though they do not share a transmission link (Ashjaei et al., 2013). Therefore, to compute the worst-case remote link delay, it is needed to consider all messages that share links with the messages that contributed to the shared link delay (see Eq. (7.13)), excluding all messages that are already considered in Eq. (7.13). Hence:

$$\mathsf{Wr}_{\mathsf{i,j,k}}(\mathsf{t}) = \sum_{\forall \mu_{p,q,r} \in \mathsf{RLD}_{\mathsf{i,j,k}}} \left\lceil \frac{t}{T_p} \right\rceil M_{p,q,r} \tag{7.16}$$

where, $\mathsf{RLD}_{\mathsf{i,j,k}} = \{\forall \mu_{p,q,r} : \mu_{p,q,r} \neq \mu_{a,b,c} \neq \mu_{i,j,k} \wedge (\mathsf{PN}_{\mu_{\mathsf{p,q,r}}} \cap \mathsf{PN}_{\mu_{\mathsf{a,b,c}}} \neq 0) \wedge (\mathsf{PN}_{\mu_{\mathsf{p,q,r}}} \cap \mathsf{PN}_{\mu_{\mathsf{i,j,k}}} = 0)(\mathsf{PN}_{\mu_{\mathsf{a,b,c}}} \cap \mathsf{PN}_{\mu_{\mathsf{i,j,k}}} \neq 0) \wedge \mu_{p,q,r} \in hp(\mu_{a,b,c}) \wedge \mu_{p,q,r} \in WT(\mu_{a,b,c})\}$.

The request bound function is compared with the **supply bound function** (denoted as $\mathsf{sbf}_{\mathsf{i,j,k}}(\mathsf{t})$). The supply bound function represents the minimum effective communication capacity that the network supplies to a message $\mu_{i,j,k}$ during the time interval $[0,t]$. In each EC, the bandwidth provided for transmitting each type of traffic (e.g., synchronous or asynchronous traffic) is equal to $\frac{(LW-I)}{EC}$, where LW is an input and represents the length of the specific transmission window and I is the maximum inserted idle time of such window. The inserted idle time results from the fact that the maximum window duration cannot be exceeded.

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \mathsf{sbf}_{\mathsf{i,j,k}}(\mathsf{t}) = (\frac{LW-I}{EC}) \times t. \tag{7.17}$$

Then, the response time of a message $\mu_{i,j,k}$ is computed by introducing a new variable

$\mathsf{t_{i,j,k}}$ such that:

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \mathsf{t_{i,j,k}} > 0, \tag{7.18}$$

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \mathsf{sbf_{i,j,k}(t_{i,j,k})} \geq \mathsf{rbf_{i,j,k}(t_{i,j,k})}. \tag{7.19}$$

Since it is not possible to determine the specific time of transmission of messages inside an EC, the computation of the WCRT for a message $\mu_{i,j,k}$ is rounded to a multiple of ECs, thus the WCRT of a message $\mu_{i,j,k}$ is given by:

$$\bigwedge_{\forall \mu_{i,j,k} \in T} \mathsf{r^{msg}_{i,j,k}} = \left\lceil \frac{\mathsf{t_{i,j,k}}}{EC} \right\rceil \times EC. \tag{7.20}$$

7.2.4 Constraint Satisfiability

The constraints sketched above are a combination of linear and non-linear constraints over a set of integer and boolean variables. This implies the use of extremely powerful optimization methods. It has been shown (e.g., (Metzner and Herde, 2006)) that such type of optimization problems are not amenable for conventional numerical optimization solvers. However, for real-time purposes, a correct solution is obtained by guaranteeing that all the constraints are satisfied, regardless of the value of a given objective function. Thus, the optimization problem gets reduced to a Satisfiability (SAT) problem, in which solutions can be obtained in reasonable time (Metzner and Herde, 2006). The constrains and optimization variables are summarized in the following.

7.2.4.1 Constraints Summary.

A set of P/D tasks τ_i is converted into a set of independent sequential tasks, by imposing a set of artificial intermediate deadlines. The constraints for intermediate deadline are presented in Eq. (7.1) and Eq. (7.2). A valid partition, in which all threads respect their intermediate deadlines $\mathsf{d_{i,j}}$, is constrained with Eq. (7.5) and Eq. (7.6). The WCRT of a distributed execution path ($DP_{i,j,k}$) depends on where the threads in a P/D segment are executed (i.e., locally or remotely), that is modeled in Eq. (7.10). If threads $\theta_{i,j,k}$ are executed remotely, the WCRT of messages transmitted through an FTT-SE network has to be considered. That is modeled with Eqs. (7.18)-(7.19). Finally, all tasks have to respect the condition in Eq.(7.11).

The system constraints are summarised in the following:

$$\bigwedge_{\forall \tau_i \in T} \bigwedge_{\forall \sigma_{i,j} \in \tau_i} \mathsf{d_{i,j}} \geq \max_{k \,\in\, \sigma_{i,j}} \{C_{i,j,k}\},$$

$$\bigwedge_{\forall\tau_i\in T}\ \sum_{\forall\sigma_{i,j}\in\tau_i} \mathsf{d}_{\mathsf{i,j}} \leq D_i,$$

$$\Pi_{\theta_{\mathsf{i,j,k}}} > 0,$$

$$\Pi_{\theta_{\mathsf{i,j,k}}} \leq m_{tot},$$

$$\bigwedge_{\forall\theta_{i,2j+1,1}\in T}\ \bigwedge_{\forall\theta_{i,2b+1,1}\in T} \Pi_{\theta_{\mathsf{i,2j+1,1}}} = \Pi_{\theta_{\mathsf{i,2b+1,1}}},$$

$$\bigwedge_{\forall\theta_{i,j,k}\in T} \mathsf{r}_{\mathsf{i,j,k}} = C_{i,j,k} + \sum_{\forall\theta_{a,b,c}\in T} \mathsf{IHP}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}},$$

$$\bigwedge_{\forall\theta_{i,j,k}\in T}\ \bigwedge_{\forall\theta_{a,b,c}\in T} \mathsf{IHP}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} = \begin{cases} \mathsf{I}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} \times C_{a,b,c} & \text{if } \left((\mathsf{p}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} = 1) \wedge (\Pi_{\theta_{\mathsf{i,j,k}}} = \Pi_{\theta_{\mathsf{a,b,c}}})\right), \\ 0 & \text{otherwise,} \end{cases}$$

$$\bigwedge_{\forall\theta_{i,j,k}\in T}\ \bigwedge_{\forall\theta_{a,b,c}\in T} (\Pi_{\theta_{\mathsf{i,j,k}}} = \Pi_{\theta_{\mathsf{a,b,c}}}) \quad \rightarrow \quad (\mathsf{I}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} \times T_a \geq \mathsf{r}_{\mathsf{i,j,k}}) \\ \wedge \left((\mathsf{I}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} - 1) \times T_a < \mathsf{r}_{\mathsf{i,j,k}}\right),$$

$$\bigwedge_{\forall\theta_{i,j,k}\in T}\ \bigwedge_{\forall\theta_{a,b,c}\in T} (\Pi_{\theta_{\mathsf{i,j,k}}} \neq \Pi_{\theta_{\mathsf{i,j,k}}}) \quad \rightarrow \quad \mathsf{I}^{\mathsf{a,b,c}}_{\mathsf{i,j,k}} = 0.$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T}\ \bigwedge_{\forall\theta_{i,j,k}\in T} \mathsf{r}_{\mathsf{DP}_{\mathsf{i,j,k}}} = \begin{cases} \mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j-1,k}} + \mathsf{r}_{\mathsf{i,j,k}} + \mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j,k}} & \text{if } \mathsf{NV}(\theta_{\mathsf{i,j,k}}) \neq \mathsf{NV}(\theta_{\mathsf{a,b,c}}), \\ \mathsf{r}_{\mathsf{i,j,k}} & \text{otherwise,} \end{cases}$$

$$\bigwedge_{\forall\theta_{i,j,k}\in T} \mathsf{r}_{\mathsf{DP}_{\mathsf{i,j,k}}} \leq \mathsf{d}_{\mathsf{i,j}}.$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T} \mathsf{rbf}_{\mathsf{i,j,k}}(\mathsf{t}) = M_{i,j,k} + \mathsf{sn}_{\mathsf{i,j,k}} \times \mathsf{SFD}_{\mathsf{i,j,k}} + \mathsf{Wl}_{\mathsf{i,j,k}}(\mathsf{t}) + \mathsf{Wr}_{\mathsf{i,j,k}}(\mathsf{t}),$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T} \mathsf{sbf}_{\mathsf{i,j,k}}(\mathsf{t}) = (\frac{LW - I}{EC}) \times t.$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T} \mathsf{t}_{\mathsf{i,j,k}} > 0,$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T} \mathsf{sbf}_{\mathsf{i,j,k}}(\mathsf{t}_{\mathsf{i,j,k}}) \geq \mathsf{rbf}_{\mathsf{i,j,k}}(\mathsf{t}_{\mathsf{i,j,k}}).$$

$$\bigwedge_{\forall\mu_{i,j,k}\in T} \mathsf{r}^{\mathsf{msg}}_{\mathsf{i,j,k}} = \left\lceil \frac{\mathsf{t}_{\mathsf{i,j,k}}}{EC} \right\rceil \times EC.$$

7.3 Summary

In this chapter the formulations for modelling the allocation of P/D tasks in a distributed multi-core architecture supported by an FTT-SE network was introduced, by using a constraint programming approach. The constraint programming approach is guaranteed to find a feasible allocation, if one exists, in contrast to other approaches based on heuristic techniques. Furthermore, similar approaches based on constraint program have shown that it is possible to obtain solutions for these formulations in reasonable time.

The following publication has been derived from the research related to this chapter:

- **R. Garibay-Martínez**, G. Nelissen, L. L. Ferreira, and L. M. Pinho. Allocation of parallel real-time tasks in distributed multi-core architectures supported by an ftt-se network. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015, volume 9017 of Lecture Notes in Computer Science*, pages 224–235. Springer International Publishing, 2015.

Chapter 8

Simulations and Experimental Evaluation

8.1 Introduction

This chapter presents some simulations and a experimental evaluation of the concepts presented in previous chapters. Section 8.3 introduces a simulator for P/D tasks that considers a FTT-SE network (Oliveira, 2015; Oliveira et al., 2015). The simulator is based on the ns-3 (network simulator version 3) which is introduced in Section 8.2. In Section 8.5 the Parallel/Distributed Real-Time (PDRT) library is introduced. The objective of presenting the PDRT library is to demonstrate that it is possible to implement the P/D task model in real systems. Finally, the results from the experimental evaluation are compared with the results obtained by simulation and with the timing analysis presented in this dissertation.

8.2 A brief Description of ns-3

This section presents a brief description of the Network Simulator version 3 (ns-3) architecture. The main objective is to introduce the super classes from ns-3 that are extended in the implementation of the FTT-SE protocol simulator and the P/D tasks execution model.

The ns-3 is an open source network simulator based on C++. Its design is based on modules which allow the efficient re-use of code. Figure 8.1 shows the organization of such modules. The *core* and *network* modules implement generic component that can be used with any network configuration. The modules on the upper layer, implement more specific functionalities. The *Internet* module implements the ARP, IPv4, IPv6, TCP e UDP protocols. The *Applications* module implement the applications that generate traffic on the network (e.g., `UdpClient, UdpServer, UdpEchoClient, UdpEchoServer, OnOffApplication` and `PacketSink`). The *Devices* module

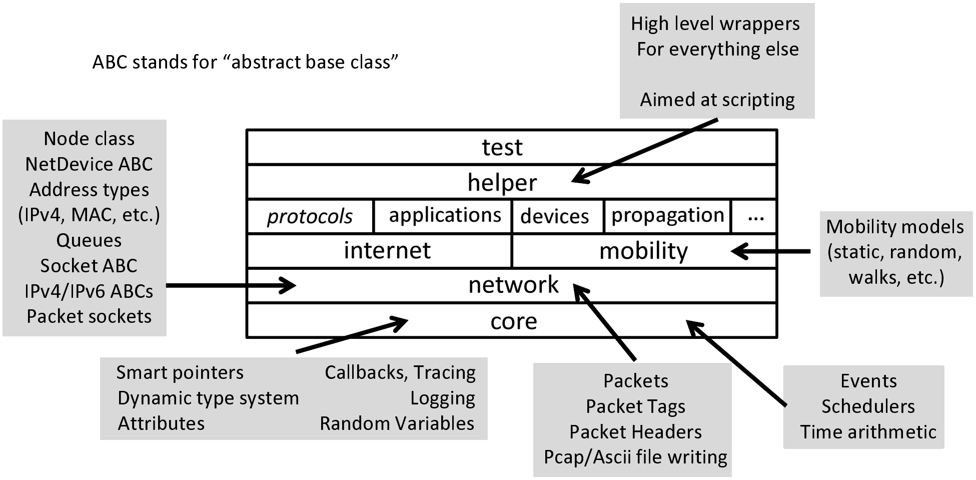

Figure 8.1: ns-3 modules (ns 3, 2015).

implement the network protocols like: CSMA, Wi-Fi, Point-to-Point, etc. ns-3 also provides specific *helpers* to each module which help on the configuration of such modules.

ns-3 simulations are based on discrete events, that is, events are executed from the beginning to the end and scheduled before the execution of the next event. In an ns-3 simulation there are fundamental elements: *Nodes*, *Applications*, *Sockets*, *NetDevices*, *Channels* and *Packets*.

Figure 8.2 provides a simplified class diagram (omitting the attributes and methods of the classes) and shows the relations between the main classes of the ns-3 simulator. Some of the main classes are described in the following:

- `Node`. An object of the class `node` represents a processing node belonging to the network. An object *node* contains a list of installed *Applications* which communicate through the implementation of *Sockets*. An object *node* also has a list of *NetDevices* that are installed in the node. Two of the most relevant methods in this class are:

  - `AddApplication`: this method installs an *application* in the *node*;
  - `AddDevice`: this method associated a network interface to a *node*.

- `Application`. All applications in ns-3 are implemented based on this class. The programmer defines in this class the attributes, for example, `StartTime` and `StopTime`, which specify the time instant at which an application starts and finishes, respectively. These attributes are used by the following methods:

  - `StartApplication`: this method is called by the `StartTime` attribute, and defines the time at which an application starts;

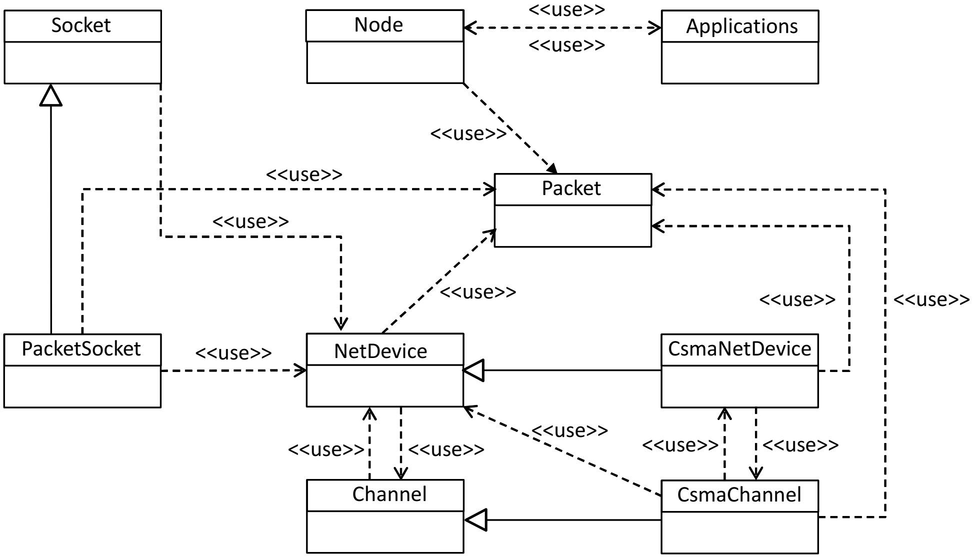

Figure 8.2: ns-3 standard classes.

- `StopApplication`: this method is called by the `StopTime` attribute, and defines the time at which an application stops.

• `Socket.` This is a class based on Berkeley sockets. It implements the transport protocols (layer 4 of the OSI model), for example, TCP or UDP. The more utilised methods of this class are:

- `Bind`: this method associates an address to a *socket*;
- `Connect`: this method initiates the connection between a *socket* and the destination address;
- `Recv`: this method reads data from its associated *socket*;
- `Send`: this method sends *Packets* to a remote *node*.

There are different implementations of *Sockets* in ns-3, but all the implementations inherit the attributes and methods of that class. Some of the more relevant implementations of the `Socket` class are:

- `TcpSocket`: implements a *Socket* for communications based on the TCP protocol;
- `UdpSocket`: implements a *Socket* for communications based on the UDP protocol;
- `PacketSocket`: implements a *Socket* that allows the communication between an *Application* and a *NetDevice* in the node.

- `NetDevice.` This class is an interface that describes how the third layer of the OSI model interacts with the second layer of the OSI model. All implementations of the class `NetDevice` emulate a real network device. Some of the most notable methods of the class `NetDevice` are:
  - `Send` : this method is in charge of sending packets thought the *NetDevice*;
  - `SetReceiveCallback`: this function is in charge of associating a `ReceiveCallback` to a *NetDevice*. This means, that every time a *Packet* reaches a *NetDevice*, the `ReceiveCallback` is executed.

  Some examples of relevant *NetDevices* for this work are:
  - `CsmaNetDevice`: implements a network interface for the communications that use the CSMA implemented in Ethernet (IEEE 802.3);
  - `WifiNetDevice`: this device emulates the interface of the IEEE 802.11 standard.
- `Channel.` This is a superclass that implements the network *Channels*. The class `Channel` implements the connection between the interfaces (i.e., *NetDevices*) of the nodes in the network and the *channels* in the network (e.g., if a `CsmaNetDevice` is considered, the channel should be a *CsmaChannel*). Some examples of *Channel* types are:
  - `CsmaChannel`: implements the *Chanel* between two nodes in a CSMA (i.e., Ethernet) network;
  - `WifiChannel`: it implements the physical properties of a Wi-Fi network.
- `Packet.` This implements a *packet* on the network. It contains a buffer, a set of byte tags, and a set of packet tags. The buffer contains the serialised information of a *Packet* (e.g., headers, trailers, data, etc.) which allows the proper modelling of a certain protocol (e.g., FTT-SE protocol).

8.3 Implementation of the FTT-SE Protocol and P/D Execution in ns-3

The FTT-SE protocol considers *full-duplex* links, but the current standard implementation of ns-3 does not provide full-duplex channels by default. In order to support full-duplex Ethernet communications, the patch Rietveld-Code-Review-Tool (2015) had to be applied to the `CsmaNetDevice` and the `CsmaChannel` classes.

8.3.1 Analysis of Requirements

The analysis of requirements is based on the study of (Marau, 2009) and on the requirements of P/D applications (presented in this dissertation). The *functional* requirements of the simulator are:

- *Master node.* The master application simulates the *Master node* within the FTT-SE protocol. The master node is in charge of coordinating and scheduling the traffic to be transmitted through the network. A fundamental functionality of the Master node is to build the TM.
- *Slave node.* The slave application simulates a *Slave node* within the FTT-SE protocol. The *Slave nodes* transmit *streams* (messages) as indicated by the TM sent by the *Master node*. The *Slave nodes* implement the sending of *Signalling messages* whenever asynchronous traffic is considered.
- *Plug-and-play (PnP) mechanism.* In the FTT-SE protocol, the *Master* and *Slave* applications have to be registered. The PnP mechanism is in charge of registering the *Master* and *Slave* applications.
- *Message scheduling.* The master application is in charge of scheduling the messages to be transmitted in each EC. Thus, the master node is the message scheduler that builds the TM in each EC.
- *Trigger Message (TM).* The information about which messages to transmit over the network is indicated though the TM.
- *Transmission of synchronous and asynchronous traffic.* Each application has to generate (publish) and transmit traffic (or streams) to the receiving nodes (subscribers). The asynchronous traffic can be divided into three subtypes: hard real-time, soft-real-time and best-effort traffic.
- *Fork-Join Parallel-Distributed Applications.* Perform simulations based on the Fork-Join Parallel/Distributed model (Chapter 3). The P/D task model considers *time-triggered* (synchronous transmissions) and *event-triggered* (asynchronous transmissions).
- *Calculating the offset for synchronization.* When considering P/D tasks based on the time-triggered (synchronous traffic) paradigm, the Master node has to calculate a synchronous offset that tells when the message associated to a D-Join can be transmitted. This offset is based on the WCRT presented in Section 6.3.

- *Task processing.* It simulates the processing of tasks in uniprocessor systems.
- *Exporting the results.* It exports the results of the simulations to text files, creating one file per each existing *stream* during the simulations.
- *Helper.* A helper facilitates the configuration (i.e. defining the architecture) of an FTT-SE network. It considers sequential messages and P/D tasks.

The *non-functional* requirements are related to the attributes and restrictions of the software. Some of of such requirements are:

- *Usability.* Easy to configure atributes and values for a simulation, start the simulation and obtain results of such simulation.
- *Implementation.* The chosen language for implemetation is C++, since the ns-3 simulator is based on C++.

8.3.2 Implemented Classes and Functionality

The simulator is implemented based on the `Applications` class of ns-3. An FTT-SE *Master* application and a set of FTT-SE *Slave* applications are derived from that class. Figure 8.3 shows a simplified class diagram of this implementation (Oliveira et al., 2015).

In Figure 8.3 it is possible to observe the `FttseMaster` class representing the *Master* node application, the class `FttseSlave` defines the characteristics of a *Slave* node application, the `FttseStream` class models the characteristics of the *streams* that are interchanged between the *Slave* node applications. The Class `SrdbNrdb` represents the *data base* containing *streams* that are part of the *Master* node and *Slave* node applications. Each application `FttseSlave` has an object of the class `Task` which represents a P/D

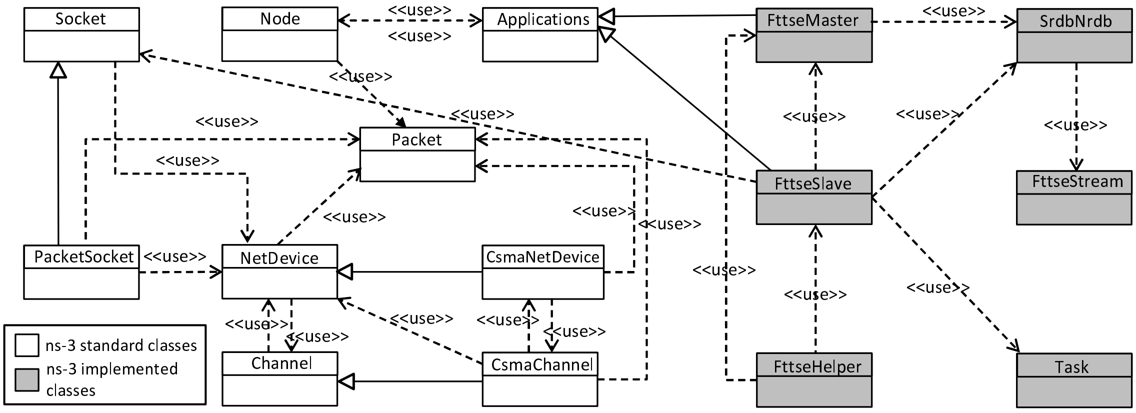

Figure 8.3: ns-3 implemented modules.

thread executing on a *Slave* application. Finally, the class `FttseHelper` is a helper that provides the necessary methods to configure the network topology and its parameters.

A brief description of functionalities of the implemented classes is presented in the following:

- `FttseMaster`. This class simulates the behaviour of the *Master node* in the FTT-SE protocol and it is extended from the class *Applications* from ns-3. The main implemented functionalities are:
  - Register of *streams* during the *Plug-and-Play* phase;
  - Scheduling of the traffic, given a certain scheduling policy;
  - Sending the TM which contain the transmission schedule for that EC;
  - Calculating the synchronization offsets for the synchronous communications used by the P/D tasks.

  This class defines the characteristics of the network (e.g. the duration of the EC and its respective sub-windows).

- `FttseSlave`. This class simulates the behaviour of the *Slave nodes* in the FTT-SE protocol. Similarly to the `FttseMaster`, the `FttseSlave` it is also extended from the *Applications* class of ns-3. It is important to recall that the FTT-SE Protocol is based on the *producer/consumer* paradigm. The main functionalities of the `FttseSlave` class are:
  - Request for registration of *streams* during the *Plug-and-Play* phase;
  - Sending *Signalling Messages* to inform their intentions of transmission;
  - Sending synchronous and asynchronous messages;
  - Simulating the concurrent execution of threads, by considering a processing node composed of a single processor.

- `FttseStream`. This class simulates a *stream* in the FTT-SE protocol. It is used to define the type of traffic to be transmitted (e.g. synchronous, asynchronous, best-effort), the size of the messages, the WCML of the message, period, and deadline. Those, characteristics are used by the *Master node* to schedule the traffic in the network.

- `SrdbNrdb`. This class simulates the *data base* that are kept in the nodes of the FTT-SE protocol containing the set of *streams* to be transmitted. There are two different types of *data bases* the ones in the *Slave nodes* containing the messages that have to compete for transmission, and the one kept in the *Master node* that aggregates the competing messages from all *Slave nodes*.

- `Task`. This class defines the tasks that are executed on the *Slave nodes*. It indicates the WCET of the task instance and simulates its execution and the scheduling policy of the *Slave node*.

- `FttseHelper`. This class has the objective of helping on network setup, thus, it starts the parameters of the network. Such parameters include: the EC size, the size of the sub-windows in the EC, the reserved time for the Signalling Window and the number of switches used in the simulation. The main functionalities of the `FttseHelper` are:

  - Installing the `FttseMaster` application in the *master node*;
  - Configuring the communications between the slave nodes (e.g. indicating the origin and destination of *streams* (messages) in the network).

8.4 Simulation Results of the ns-3 Module Implementation

In this section we evaluate the performance of an FTT-SE network through a series of simulations based on the implementation described in Section 8.3. These experiments are inspired on research presented in (Lim et al., 2011) in which automotive applications are considered. In Section 8.4.1 the evaluation of the transmission of messages within the FTT-SE protocol is presented and the evaluation of the execution of P/D tasks is shown in Section 8.4.2. The objective of these experiments is to show how different parameters chosen in the configuration of the FTT-SE network can affect the response time of messages and therefore the execution of P/D tasks.

8.4.1 Evaluating the Transmission of Messages

Figure 8.4 shows a system architecture that is used on the experiments presented in this section. It is assumed that all links in the network are *full-duplex* and have a capacity of 100 Mb/s. The Maximum Transmission Unit (MTU) of the network is fixed to 1400 Bytes.

In this section it is evaluated the effect of the parameters of the FTT-SE network on the response time of messages. There are four categories of messages that are considered:

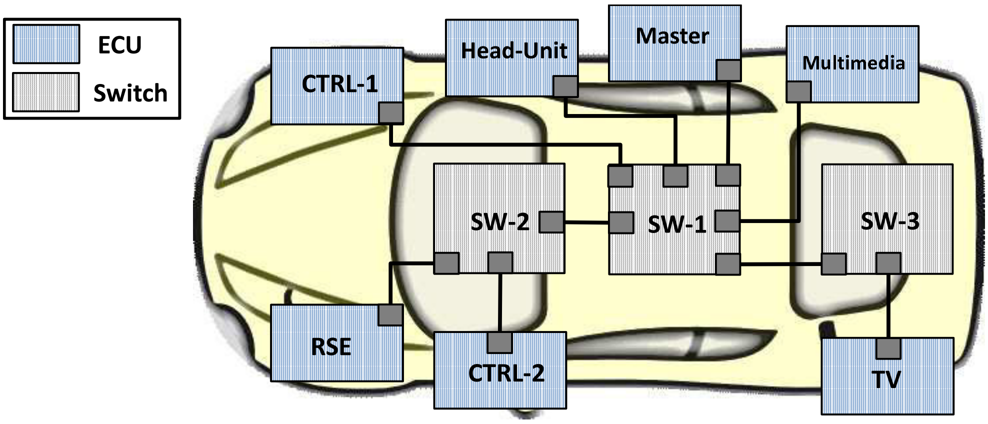

Figure 8.4: Simulated automotive architecture for the evaluation of message transmissions.

control, navigation, multimedia (which include video and audio), and TV (which include video and audio). Messages have implicit deadlines (i.e. $D_i = T_i$).

8.4.1.1 Synchronous Messages

The characteristics of the synchronous messages are as follows. There are two control messages μ_1 and μ_2 with origin in the CTRL-1 ECU producing a 64 bytes message with a WCML of 7 μs and a period of 11 ECs. In the case of message μ_1, the destination node is the CTRL-2 and μ_2 has as destination the Rear Seat Entertainment (RSE) node. The navigation message μ_3, has an origin in the Head-Unit (HU) and sends a message to the RSE of 5000 bytes with a WCML of 406 μs and a period of 100 ECs. The multimedia messages μ_4 (Video) and μ_5 (Audio) send streams of 1400 bytes with a WCML of 114 μs which has an origin in the Multimedia node and is sent to the RSE node. The Video message has a period of 1 EC and the Audio message has a period of 2 ECs. Messages μ_6 (Video) and μ_7 (Audio) produce data for the TV node. Both send messages of 1400 bytes with a WCML of 114 μs. The origin of messages μ_6 and μ_7 is the TV node, they are sent to the HU node. μ_6 and μ_7 have a periodicity of 2 ECs and 3 ECs, respectively. The characteristics of the applications are summarised in Table 8.1.

For this experiments the characteristics of the FTT-SE network are: the EC is fixed with a duration of 1000 μs, 100 μs are reserved for the *Signaling Window* (SIG), 60% of 900 μs (i.e. 540 μs) are reserved for the transmission of synchronous messages (the Synchronous Window (SW)) and 40% of 900 μs (i.e. 360 μs) are reserved for the transmission of asynchronous messages (the Asynchronous Window (AW)). Table 8.2 shows the characteristics of the FTT-SE network for the simulation of synchronous message transmissions.

Table 8.1: Characteristics of the synchronous messages for the simulation in th ns-3 module presented in Section 8.3.

Message	Type	Period	Size	WCML	Origin	Destination
μ_1	Control	11 ECs	64 B	7 μs	CTRL-1	CTRL-2
μ_2	Control	11 ECs	64 B	7 μs	CTRL-1	RSE
μ_3	Navigation	100 ECs	5000 B	406 μs	Head Unit	RSE
μ_4	MM Video	1 EC	1400 B	114 μs	Multimedia	RSE
μ_5	MM Audio	2 ECs	1400 B	114 μs	Multimedia	RSE
μ_6	TV Video	2 ECs	1400 B	114 μs	TV	Head-Unit
μ_7	TV Audio	3 ECs	1400 B	114 μs	TV	Head-Unit

Table 8.2: Parameters of the FTT-SE network for the simulation of the sequential synchronous applications in ns-3.

Elementary Cycle (EC)	1000 μs
Signalling Window (SIG)	100 μs
Synchronous Window (SW)	60% of 900 μs(540 μs)
Asynchronous Window (AW)	40% of 900 μs(360 μs)

Figure 8.5 shows the response times for the synchronous messages. Messages μ_1, μ_2, μ_3, μ_4, μ_5, μ_6 and μ_7 have an average response time of 109 μs, 117 μs, 5250 μs, 430 μs, 548 μs, 425 μs and 482 μs, respectively. It is possible to observe that all application are sent within 1 EC (1000 μs) with the exception of μ_3 which has an average response time inferior to 6 ECs (6000 μs). The increased response time on μ_3 is due to the fact that it has the largest WCML and it has the lowest priority between all other applications, therefore, it was not possible to schedule it in a single EC. Please notice that if the message size is superior to the defined MTU, it is divided in a set of messages with maximum size equal to the MTU. It is also possible to observe that μ_5 and μ_7 present some slight variations in their response times. In the case of μ_5 this is due to the interference provoked by the control messages (μ_1 and μ_2) since they share some links and sometimes they coincide in their activation. In the case of message μ_7 such interference is due to message μ_6.

An important parameter when choosing the parameters of the FTT-SE network is the EC size. Figure 8.6 shows the response times of message μ_3 when varying the EC size to 1000 μs, 1500 μs and 2000 μs. The average response times are for EC sizes equal to 1000 μs, 1500 μs and 2000 μs are 5250 μs, 1750 μs and 958 μs, respectively. It is also important to notice that the number of ECs in which the transmission is completed is 6 ECs (6000 μs), 2 ECs (3000 μs) and 1 EC (2000 μs), respectively.

One can notice that when the EC size increases the SW also increases therefore there is more bandwidth to transmit messages. In this case, the bandwidth is required and harnessed, but it is important to carefully chose this parameter since it could lead to band-

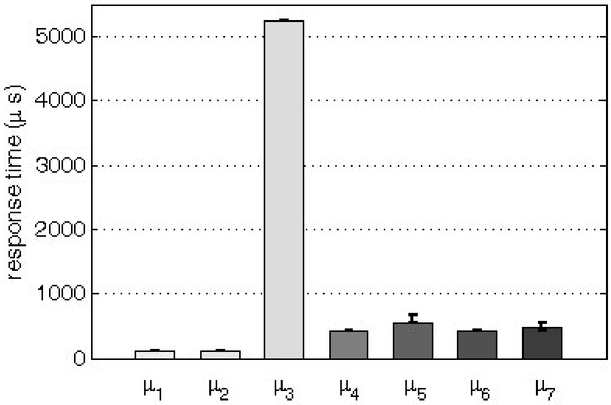

Figure 8.5: Average response times for synchronous messages: the messages characteristics are summarised in Table 8.1 and the FTT-SE network configuration is shown in Table 8.2.

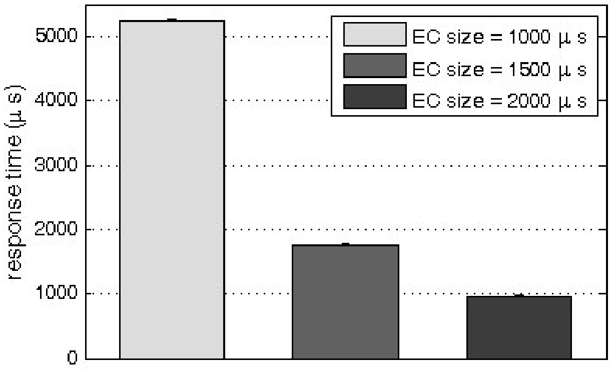

Figure 8.6: Average response times for sequential synchronous message μ_3 varying the EC to 1000μs, 1500μs and 2000μs.

width wasting if the EC is too large in comparison with the requirements of the messages to transmit. Thus, it is possible to conclude that the EC size (and therefore the SW and AW size) directly impact the response time of messages transmitted within a FTT-SE network, and therefore impact the response time of P/D tasks over FTT-SE.

8.4.1.2 Asynchronous Messages

The characteristics of asynchronous messages are very similar to the ones presented in Table 8.1. However, one of the main differences from Table 8.1 is that in Table 8.3 the periodicities of some applications are larger. The reason behind that change, is that there exist some delays related to the signalling mechanism, that have to be considered in the response time of the applications (see Section 6.2.2).

Table 8.3: Parameters of the asynchronous messages for the simulation in ns-3.

Message	Type	Period	Size	WCML	Origin	Destination
μ_1	Control	11 ECs	64 B	7 μs	CTRL-1	CTRL-2
μ_2	Control	11 ECs	64 B	7 μs	CTRL-1	RSE
μ_3	Navigation	24 ECs	5000 B	406 μs	Head Unit	RSE
μ_4	MM Video	3 ECs	2800 B	227 μs	Multimedia	RSE
μ_5	MM Audio	6 ECs	2800 B	227 μs	Multimedia	RSE
μ_6	TV Video	6 ECs	2800 B	227 μs	TV	Head-Unit
μ_7	TV Audio	9 ECs	2800 B	227 μs	TV	Head-Unit

The characteristics of the asynchronous messages are as follows. There are two control messages μ_1 and μ_2 with origin in the CTRL-1 ECU producing 64 bytes message with a WCML of 7 μs and a period of 11 ECs. In the case of message μ_1, the destination is the CTRL-2 node and μ_2 has as destination the Rear Seat Entertainment (RSE) node. The navigation message μ_3, has an origin in the Head-Unit (HU) and is sent to the RSE of 5000 bytes (i.e. with a WCML of 406 μs) with a period of 24 ECs. The multimedia streams τ_4 and τ_5 send messages of 2800 bytes (equivalent to a WCML of 227 μs) which have an origin in the Multimedia node and their destination on the RSE node. The Video message has a period of 3 ECs and the Audio message has a period of 6 ECs. Messages μ_6 (Video) and μ_7 (Audio) produce data for the TV node. Both applications send messages of 2800 bytes (equivalent to a WCML of 227 μs). The origin of messages τ_6 and τ_7 is the TV node and it is send to the application in the HU node. τ_6 and τ_7 have a periodicity of 6 ECs and 9 ECs, respectively. The characteristics of the asynchronous messages are shown in Table 8.3.

Similarly to synchronous messages, Table 8.3 shows the parameters of the FTT-SE network for the simulation of the asynchronous messages.

Figure 8.7 shows the results of the experiments. Messages μ_1, μ_2, μ_3, μ_4, μ_5, μ_6 and μ_7 have an average response time of 1642 μs, 1640 μs, 4006 μs, 2078 μs, 2957 μs, 2103 μs and 2478 μs, respectively. The average response time of applications μ_1 and μ_2 is inferior to 2 ECs (i.e., 2000 μs), for applications μ_4, μ_5, μ_6 and μ_7 the average response time is below 3 ECs (i.e., 3000 μs) and for application μ_3 it is inferior to 5 ECs (i.e., 5000 μs).

When comparing Figure 8.5 and Figure 8.7, it is possible to notice that in Figure 8.7 none of the response times of the applications is inferior to 1 EC. The reason is because the signalling mechanism can take up to 2 ECs for initiating transmission for the case of asynchronous applications (see Section 6.3). In relation with the results presented in Figure 8.5, for the case of asynchronous the variations are very similar but including the signalling mechanism overhead.

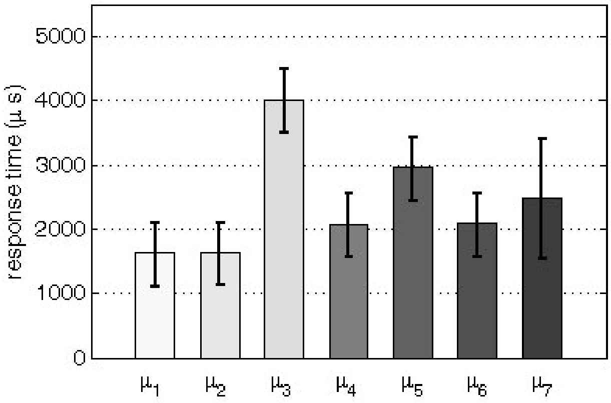

Figure 8.7: Average response times for sequential asynchronous messages: EC of $1000\,\mu$s, SW of $540\,\mu$s, AS of $360\,\mu$s.

8.4.2 Evaluating the Execution of P/D Tasks and Sequential Application

Figure 8.8 shows a system architecture that is used for the experiments of P/D tasks presented in this section. The architecture is composed of 3 switches and 7 ECUs (one acting as a master node of the FTT-SE network). It is assumed that all links in the network are *full-duplex* and they have a capacity of 100 Mb/s. The Maximum Transmission Unit (MTU) of the network is fixed to 1400 Bytes.

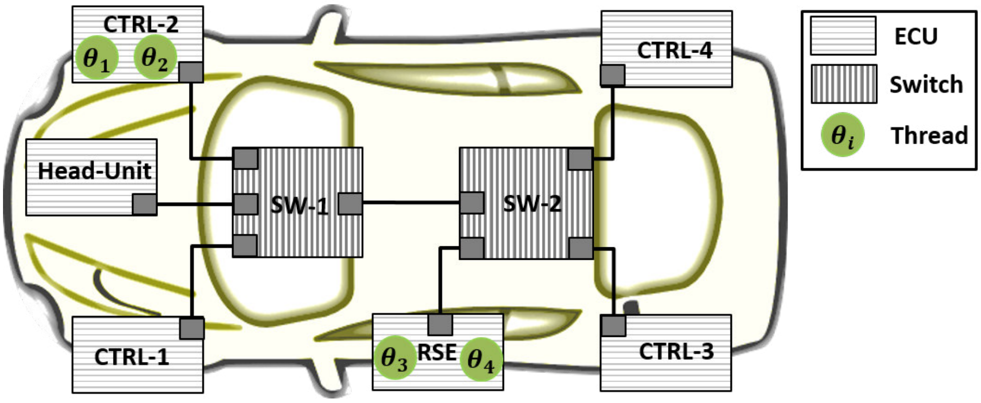

Figure 8.8: Simulated automotive architecture for the execution of P/D tasks and sequential applications.

The characteristics of the FTT-SE network for these experiments are: the EC is fixed to have a duration of $1500\,\mu$s, $100\,\mu$s are reserved for the SIG window, 50% of $1400\,\mu$s (i.e. $700\,\mu$s) are reserved for the transmission of synchronous messages (in the SW) and 50% of $1400\,\mu$s (i.e. $700\,\mu$s) are reserved for the transmission of asynchronous messages (in the AW). Table 8.4 shows the parameters of the FTT-SE network for this simulation.

Table 8.4: Characteristics of the FTT-SE network for the simulation of P/D tasks and sequential synchronous applications in ns-3.

Elementary Cycle (EC)	$1500\,\mu$s
Signalling Window (SIG)	$100\,\mu$s
Synchronous Window (SW)	50% of $1400\,\mu$s($700\,\mu$s)
Asynchronous Window (AW)	50% of $1400\,\mu$s($700\,\mu$s)

The experiments related to P/D tasks correspond to the execution pattern introduced in Section 3.4. A P/D task starts its execution by performing a D-Fork operation which triggers a set of messages that are transmitted over the network, activating the execution of threads in remote processors and returning their partial results to the node that triggered the process by executing a D-Join operation. This execution pattern inside a P/D segment is called Distributed Execution Path (DEP). The P/D tasks simulate vehicular applications that execute in a parallel and distributed manner. Also, sequential applications are executing in the system. Sequential applications consist of a message that is transmitted through the network and the execution of a thread that is triggered by the arrival of the message in a remote processor. Therefore, in contrast with Section 8.4.1 , in this section we denote the applications win τ_i instead of using μ_i.

In here, three types of applications are considered: (i) control, which is a sequential application; (ii) multimedia video (MM video), which is a P/D task; and (iii) multimedia audio (MM audio), which is a sequential application. The three control applications τ_1, τ_2 and τ_3 produce messages of 350 bytes, with a WCML of $30\,\mu$s which activates the execution of a remote threads with a WCET of $80\,\mu$s. τ_1, τ_2 and τ_3 have a period of 5 ECs and have an origin in the Head-Unit ECU and a destination in the ECUs CTRL-1, CTRL-2 and CTRL-3, respectively.

Application τ_4, is a P/D task application. The MM Video P/D application τ_4 has an WCET for Fork and Join execution of $150\,\mu$s. In this section, the parallel workload ($6400\,\mu$s) of the application τ_4 is divided in 8 P/D threads which are sent to CTRL-1 to CTRL-4 (2 threads to each CRTL node) and in 16 P/D threads are alo sent to CTRL-1 to CTRL-4 (4 threads to each CRTL node), with the objective of analysing what it is the impact on the load of both the WCML, and the WCET of the threads executing remotely. The WCET of the remote execution (WCET (rem)) is of $800\,\mu$s when the parallel load is divided between 8 threads and $400\,\mu$s when the load is divided between 16 threads. τ_4 has a period of 10 ECs. Also, τ_4 sends messages of 1400 bytes with a WCML of $114\,\mu$s with origin in the Multimedia node and destination in the nodes CTRL-1-4. MM Audio applications is sequential application which produces a message of 2000 bytes,

Table 8.5: Characteristics of the applications (sequential and P/D tasks) for the simulation in ns-3.

Application	Type	Period	WCET (F/J)	WCET (Rem)	Size	WCML	Origin	Destination
τ_1	Control	5 ECs	-	$80\,\mu$s	350 B	$30\,\mu$s	Head-Unit	CRTL-1
τ_2	Control	5 ECs	-	$80\,\mu$s	350 B	$30\,\mu$s	Head-Unit	CRTL-2
τ_3	Control	5 ECs	-	$80\,\mu$s	350 B	$30\,\mu$s	Head-Unit	CRTL-3
τ_4 (8 threads)	MM Video	10 ECs	$150\,\mu$s	$800\,\mu$s	1400 B	$114\,\mu$s	Multimedia	CRTL-1-4
τ_4 (16 threads)	MM Video	10 ECs	$150\,\mu$s	$400\,\mu$s	700 B	$57\,\mu$s	Multimedia	CRTL-1-4
τ_5	MM Audio	30 ECs	-	$350\,\mu$s	2000 B	$163\,\mu$s	Multimedia	CRTL-1

with a WCML of $163\,\mu$s and a period of 5 ECs. The WCET for its remote execution is of $350\,\mu$s. Applications τ_1 has an origin in the Multimedia ECU and destination in the CTRL-1 ECU. Table 8.5 shows the characteristics of the P/D tasks and sequential applications for this simulation.

8.4.2.1 Synchronous P/D Tasks and Sequential Applications

Figure 8.9 and Figure 8.11 show the response times for applications shown in Table 8.5, when τ_4 is divided in 8 and 16 threads, respectively. Messages are transmitted using *synchronous* messages and FTT-SE parameters are shown in Table 8.4.

Figure 8.9 shows the results of the experiments when τ_4 is divided in 8 threads. Application τ_1, τ_2, τ_3, τ_4, and τ_5 have an average response time of $228\,\mu$s, $288\,\mu$s, $319\,\mu$s, $8303\,\mu$s, and $2481\,\mu$s, respectively. It is possible to observe that applications τ_1, τ_2 and τ_3 are sent within 1 EC ($1500\,\mu$s), τ_4 is sent within the 6 ECs ($9000\,\mu$s) and τ_5 is sent within the 2 ECs ($3000\,\mu$s). It is important to remember that in synchronous systems, it is necessary to know the exact instants at which threads and messages are activated, therefore, it is necessary to compute the sum of the WCRT of the first message $\mu_{i,2j-1,k}$ and the P/D thread $\theta_{i,2j,k}$, and use it as an offset for the activation of $\mu_{i,2j,k}$.

Figure 8.10 shows the response times for the Distributed Execution Paths (DEPs) of τ_4 when it is divided in 8 threads. DEPs: $DP_{4,2,1}$, $DP_{4,2,2}$, $DP_{4,2,3}$, $DP_{4,2,4}$, $DP_{4,2,5}$, $DP_{4,2,6}$, $DP_{4,2,7}$ and $DP_{4,2,8}$ have an average response time of $3575\,\mu$s, $5076\,\mu$s, $4961\,\mu$s, $6461\,\mu$s, $5190\,\mu$s, $8188\,\mu$s, $6688\,\mu$s, and $8303\,\mu$s, respectively. It is possible to observe that the maximum observed average response time for the 8 DEPs is the one of $DP_{4,2,8} = 8303\,\mu$s, therefore, $DP_{4,2,8}$ is the DEP that dictates the response time of the P/D task τ_4. Therefore, $DP_{4,2,8}$ is the same as the response time of τ_4 in Figure 8.9 .

Figure 8.11 shows the results of the experiments when τ_4 is divided in 16 threads. Application τ_1, τ_2, τ_3, τ_4, and τ_5 have an average response time of $228\,\mu$s, $288\,\mu$s, $319\,\mu$s, $8017\,\mu$s, and $3564\,\mu$s, respectively. It is possible to observe that applications τ_1, τ_2 and τ_3 are sent within 1 EC ($1500\,\mu$s), τ_4 is sent within the 6 ECs ($9000\,\mu$s) and τ_5 is sent within the 3 ECs ($4500\,\mu$s).

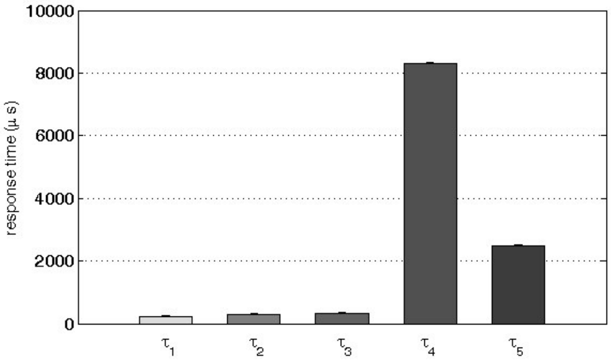

Figure 8.9: Average response times for applications shown in Table 8.5, τ_4 is divided in 8 threads, transmitted using *synchronous* messages and FTT-SE characteristics: EC of 1500 μs, SW of 700 μs, AS of 700 μs.

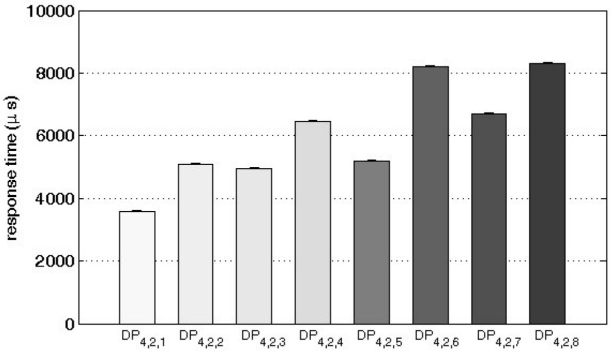

Figure 8.10: Average response times of the Distributed Execution Paths of τ_4 when divided in 8 threads, for the values presented in Table 8.5, transmitted using *synchronous* messages and FTT-SE characteristics: EC of 1500 μs, SW of 700 μs, AS of 700 μs.

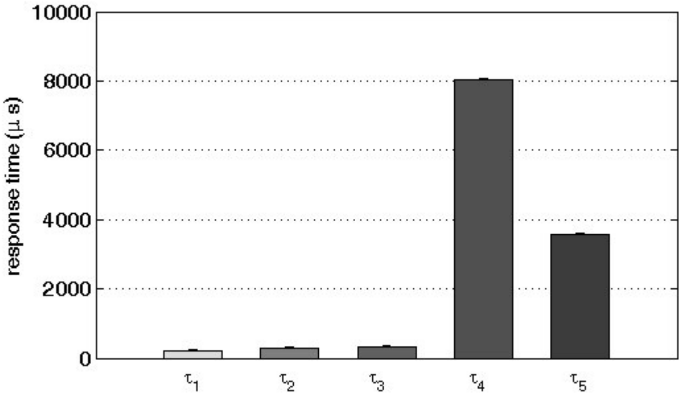

Figure 8.11: Average response times for applications shown in Table 8.5, τ_4 is divided in 16 threads, transmitted using *synchronous* messages and FTT-SE characteristics: EC of 1500 μs, SW of 700 μs, AS of 700 μs.

Comparing Figure 8.9 and Figure 8.11 , one can notice that the response time of τ_4 in 8.11 is reduced from 8303 μs to 8017 μs and for τ_5 it increases from 2481 μs to 3564 μs. The reason is because since the WCML of the messages related to τ_4 is half when divided in 16 threads (when compared to when is divided among 8 threads), it is possible to fit those messages in earlier ECs (since they have higher priority than the ones belonging to τ_5). As a consequence, messages of lower priority belonging to τ_5 are postponed to be scheduled in other ECs.

8.4.2.2 Asynchronous P/D Tasks and Sequential Applications

Figure 8.12 and Figure 8.13 show the response times for applications shown in Table 8.5, when τ_4 is divided in 8 and 16 threads, respectively. Similarly to Section 8.4.2.1, the response times for τ_4 in both cases is taken from the maximum response times of their respective DEPs. Messages of a P/D task are transmitted using *asynchronous* messages and FTT-SE characteristics are as shown in Table 8.4.

Figure 8.12 shows the results of the experiments when τ_4 is divided in 8 threads. Application τ_1, τ_2, τ_3, τ_4, and τ_5 have an average response time of 2481 μs, 2549 μs, 2570 μs, 9021 μs, and 4773 μs, respectively. It is possible to observe that applications τ_1, τ_2 and τ_3 are sent within 2 EC (3000 μs), τ_4 is sent within the 7 ECs (10500 μs) and τ_5 is sent within 4 ECs (6000 μs). For the case of asynchronous applications, the messages/threads are activated whenever the previous thread/message completes its execution/transmission. This activation is made through the signalling mechanism. When comparing synchronous and asynchronous P/D tasks, one can observe that the response time of the asynchronous messages increases up to 2 ECs due to the signalling mechanism.

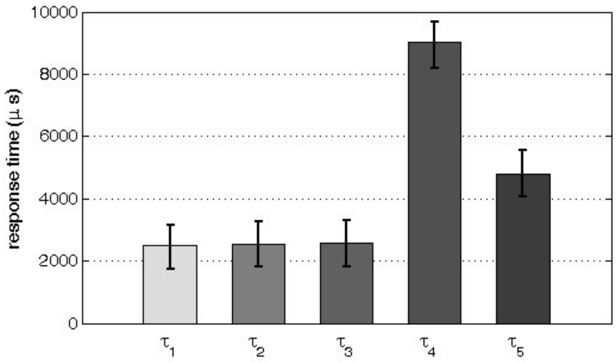

Figure 8.12: Average response times for applications shown in Table 8.5, τ_4 is divided in 8 threads, transmitted using *asynchronous* messages and FTT-SE characteristics: EC of 1500 μs, SW of 700 μs, AS of 700 μs.

Figure 8.13 shows the results of the experiments when τ_4 is divided in 16 threads. Application τ_1, τ_2, τ_3, τ_4, and τ_5 have an average response time of 2481 μs, 2489 μs, 2559 μs, 8838 μs, and 4830 μs, respectively. It is possible to observe that applications τ_1, τ_2 and τ_3 are sent within 2 EC (3000 μs), τ_4 is sent within the 6 ECs (9000 μs) and τ_5 is sent within the 4 ECs (6000 μs).

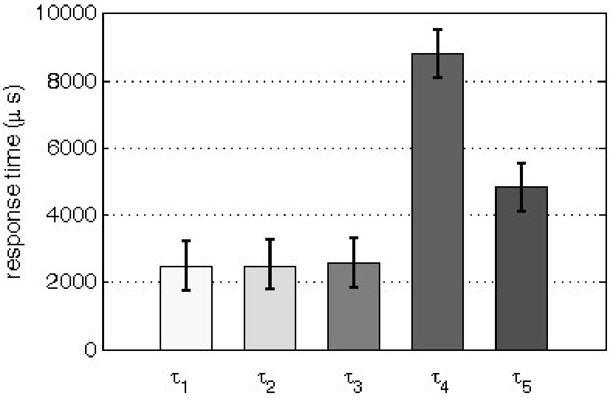

Figure 8.13: Average response times for applications shown in Table 8.5, τ_4 is divided in 16 threads, transmitted using *asynchronous* messages and FTT-SE characteristics: EC of 1500 μs, SW of 700 μs, AS of 700 μs.

By comparing Figure 8.12 and Figure 8.13 one can notice that in a similar situation to the synchronous case, the response time of τ_4 in 8.13 is reduced from 9021 μs to 8838 μs and for τ_5 it increases from 4773 μs to 4830 μs. Again, the reason is because since the WCML of the messages belonging to τ_4 is smaller (half), when compared to when is divided among 8 threads, therefore, it is possible to fit those messages in earlier ECs (since they have higher priority than the ones belonging to τ_5). Contrarily, the messages of lower priority belonging to τ_5 are postponed to be scheduled in other ECs.

8.4.3 Evaluating the Average Response Time Reduction by Applying the Parallel/Distributed Approach

This section presents the evaluation of the average response time for P/D tasks and its behaviour (e.g. increases or decreases), whenever is parallelised and distributed in different number of nodes in the architecture.

Similarly to Section 8.4.2, in this section we consider the architecture depicted in Figure 8.8 and the same network characteristics (*full-duplex* links in the network with a capacity of 100 Mb/s and a MTU equal to 1400 Bytes). Asynchronous messages are used for transmission. The parameters of the FTT-SE network are described in Table 8.4

Figure 8.14 shows the response times for applications shown in Table 8.5, these applications are the same as the ones presented in Section 8.4.2. However, only the response time of the P/D task τ_4 is depicted and the load of τ_4 is divided in 2, 3, 4, 8, 16 threads. Threads (and the respective messages) that τ_4 generates have an origin in the Multimedia node. Whenever 2 threads are used, the destination for remote execution are CRTL-1 and CTRL-2, when 3 threads are generated the destination nodes are CRTL-1, CTRL-2 and CTRL-3, and whenever 4, 8 and 16 threads are used the destination of the nodes for remote execution are CRTL-1, CTRL-2, CTRL-3 and CTRL-4 as in Section 8.4.2.1. This is summarised in Table 8.6.

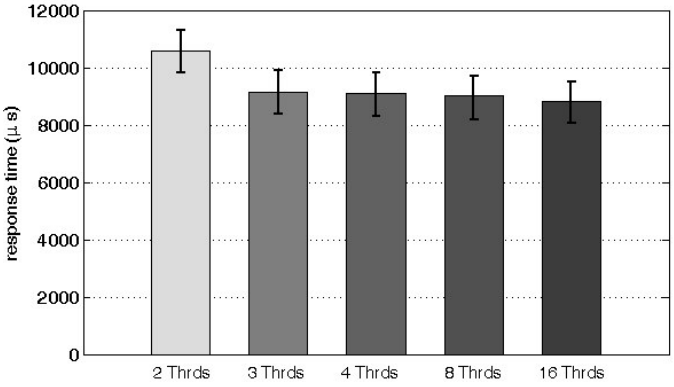

Figure 8.14: Average response times for applications shown in Table 8.5, the characteristics of the network are described in Table 8.4, only the response time of τ_4 is depicted and the load of τ_4 is divided in 2, 3, 4, 8, 16 threads as shown is Table 8.6.

Table 8.6: Characteristics of the applications (sequential and P/D tasks) for the simulation in ns-3.

Application	Type	Period	WCET (F/J)	WCET (Rem)	Size	WCML	Origin	Destination
τ_4 (2 threads)	MM Video	10 ECs	150μs	3200μs	5600 B	456μs	Multimedia	CRTL-1-2
τ_4 (3 threads)	MM Video	10 ECs	150μs	2133μs	3733 B	304μs	Multimedia	CRTL-1-3
τ_4 (4 threads)	MM Video	10 ECs	150μs	1600μs	2800 B	228μs	Multimedia	CRTL-1-4
τ_4 (8 threads)	MM Video	10 ECs	150μs	800μs	1400 B	114μs	Multimedia	CRTL-1-4
τ_4 (16 threads)	MM Video	10 ECs	150μs	400μs	700 B	57μs	Multimedia	CRTL-1-4

It is possible to see that there is a decreasing in the response time of application τ_4 whenever more threads are used. There is a significant decrease from the use of 2 threads to the use of 3 threads, since there are more physical resources (ECUs) being used. It is interesting to notice that although the amount of used ECUs does not augment for a number of threads superior to 4, there is a slight improvement on the response time of the applications which is due to the better utilization of the reserved bandwidth in the network (see Section 8.4.2.1).

8.5 Comparing Experimental Results and Simulation Results

In order to demonstrate the feasibility of using such an execution pattern in a real-time environment, the Parallel/Distributed Real-Time (PDRT) library has been implemented. Also, in this section we compare the simulation results obtained with the simulator described in Section 8.3 versus the results obtained with the PDRT library and versus the results of the WCRT analysis presented in this thesis.

The PDRT library makes the workload distribution between a set of distributed nodes. The PDRT library implements a *for* loop with parallel distributed real-time behaviour as the one described in (Garibay-Martínez et al., 2013a). The PDRT distributes the load in a *for* loop by following a similar behaviour as the example in Figure 6.6, that is, it evenly distributes the iterations in the *for* loop between the nodes in the system (the DST algorithm is not used.). The PDRT library implements the execution of multiple copies of code (one per each *Slave* node in the systems, in a similar manner as in the MPI programming model) which are distributed by a distribution server (e.g., New Technology File System (NTFS)). The data code distribution is made off-line in such a way that it does not affect the real-time behaviour of the code. Each copy of the code, implements its own functionality based on processes IDs (e.g., master and slave roles). The real-time communications are implemented (the main requirements described on Section 8.3) using the module of the FTT-SE for Linux which can be found in (Marau, 2015).

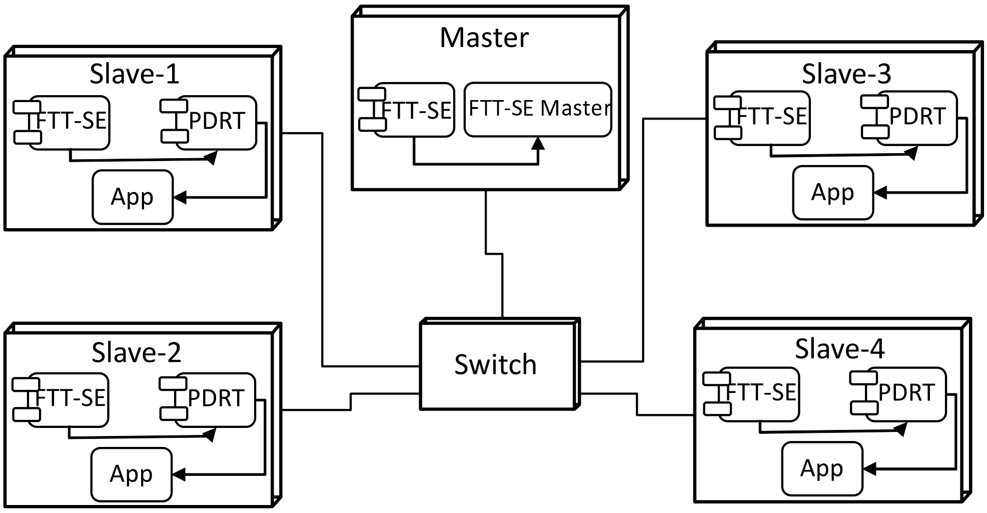

Figure 8.15: Experimental evaluation architecture.

Figure 8.15 shows the deployment diagram of the configuration used for the experimental evaluation of the PDRT library. The deployment diagram is based on three different processing elements with the following characteristics:

1. Master Node (Desktop CPU):
    - Operating system: Debian 7.8 kernel 3.2.0-4-rt-686-pae with PREEMPT RT 3.2.68-1+deb7u1 i686 patch.
    - Processor: AMD Sempron 2800+, 1600 MHz.
    - RAM: 1 GB.
    - Applications: *Master* application. The objective of the Master application is to arbitrate and grant access for transmission to the slave nodes (Marau, 2015).
2. 4 Slave Nodes (Laptop HP ProBook 6460b):
    - Operating system: Debian 7.8 kernel 3.2.0-4-rt-686-pae with PREEMPT RT 3.2.68-1+deb7u1 i686 patch.
    - Processor: Intel Celeron B840 1895 MHz.
    - RAM: 2 GB.
    - Applications: *Slave* application. In the context of the P/D task model, the Applications (App) make use of the PDRT library to generate the threads and messages for execution (Marau, 2015).
3. Ethernet Switch (TPLINK TL-SF1008D)
    - Speed: 100 Mbps.
    - Number of ports: 8.

In order to provide more determinism for the experiments presented in this section, the Advanced Configuration and Power Interface (ACPI) and the Advanced Power Management (APM) features are disabled. Also, the multi-core capabilities of the Slave nodes are disabled, therefore, the architecture is composed by uni-processor nodes only.

The experiment consists on executing two applications (P/D tasks) τ_1 and τ_2. Application τ_1 has a sequential WCET of 200 ms and a period of 150 ms, application τ_2 has a sequential WCET of 300 ms and a period of 300 ms. Both applications start its execution at the Slave-1 node and create 4 threads for execution (i.e., $\theta_{1,2,1-4}$ and $\theta_{2,2,1-4}$). One of the threads is kept for local execution and the other 3 threads are executed in the slave nodes 2-4. Each one of the created threads ($\theta_{1,2,1-4}$ and $\theta_{2,2,1-4}$) have a WCET as shown in Table 8.7. The FTT-SE network is being operated on event-triggered mode. Due to the OS-related overheads, Linux-based FTT-SE applications use an EC duration of 10 ms Marau (2015). Therefore, the resolution of the applications in this experiments is in ms.

Table 8.7: Characteristics of the P/D tasks for comparison of the experimental and simulation evaluation.

App	Type	Period	WCET(Seq.)	WCET(F/J)	WCET(Remote)	WCML	Invok. Node	Rem. Node
τ_1	-	150 ms	200 ms	-	-	-	-	-
-	$\mu_{1,j,k}$	150 ms	-	-	-	112 μs	Slave-1	-
-	$\theta_{1,2,k}$	150 ms	-	38 ms	50 ms	-	Slave-1	Slave 2-4
τ_2	-	300 ms	300 ms	-	-	-	-	-
-	$\mu_{2,j,k}$	300 ms	-	-	-	112 μs	Slave-1	-
-	$\theta_{2,2,k}$	300 ms	-	38 ms	75 ms	-	Slave-1	Slave 2-4

The characteristics of the P/D tasks for comparison of the experimental and simulation evaluation are shown in Table 8.7.

By observing Table 8.7, it is easy to notice that the task set composed of τ_1 and τ_2 would not be schedulable without a parallel distributed execution. This, shows the value of our approach for real-time distributed environments.

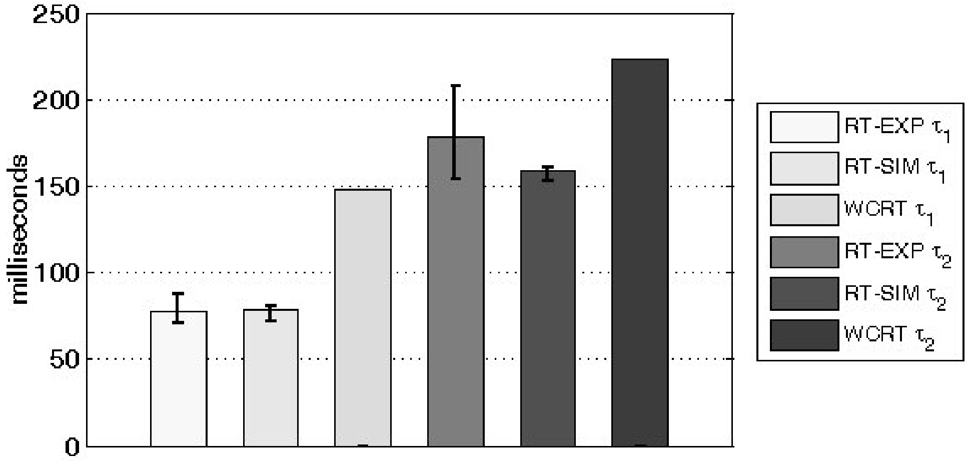

Figure 8.16: Response Time of Experiments (RT-EXP) of P/D tasks τ_1 and τ_2, Response Time of Simulations (RT-EXP) of P/D tasks τ_1 and τ_2 and WCRT of P/D tasks τ_1 and τ_2.

Figure 8.16 shows the response times which are obtained by executing 1000 experiments for applications τ_1 and τ_2 when executed concurrently. $\mathsf{RT}\tau_1$ (equal to 77.84 ms) represents the average observed response time of application τ_1, $\mathsf{WCRT}\tau_1$ (148 ms) represents the WCRT estimation of τ_2 using the analysis presented in Chapter 6, similarly, $\mathsf{RT}\tau_2 = 177.82$ ms represents the average observed response time of τ_2, and $\mathsf{WCRT}\tau_2$ (equal to 223 ms) represents the WCRT of τ_2. Also, note that for $\mathsf{RT}\tau_1$, the error bars represent the minimum observed response time $\mathsf{RT}^{\min}(\tau_1)$ (equal to 70.33 ms) and maximum observed response time $\mathsf{RT}^{\max}(\tau_1)$ (equal to 87.74 ms), similarly, for $\mathsf{RT}\tau_2$ the error bars represent the minimum $\mathsf{RT}^{\min}(\tau_2)$ (equal to 154.22 ms) and maximum $\mathsf{RT}^{\max}(\tau_2)$ (equal to 207.24 ms) observed response times.

In Figure 8.16 one can observe that the WCRT computed with the analysis presented in Chapter 6 is always an upper bound of the observed response time of applications τ_1 and τ_2. These experiments show that our holistic analysis is valid for this example and valuable for designers when developing parallel and distributed systems with real-time constraints.

8.6 Summary

This chapter presented the simulations and a experimental evaluation of the concepts presented in previous chapters. These results allows to confirm that the solutions proposed in this dissertation can be implemented in real systems.

The following publication has been derived from the research related to this chapter:

- Fábio Oliveira, **Ricardo Garibay-Martínez**, Tiago Cerqueira, Michele Albano, and Luis Lino Ferreira. A module for the ftt-se protocol in ns-3. Demo session in Workshop on ns-3 (WNS3 2015), 2015.

Chapter 9

Conclusions and Future Work

9.1 Research Context and Research Contributions

This dissertation proposed an extension *multi-threaded parallel real-time task models* that considers the cases in which there exists a *parallel distributed execution* pattern. Such execution pattern provides more capabilities and processing power, which is transformed on execution flexibility for distributed real-time applications. Furthermore, in some applications the use of parallel computations is the only possibility in which the applications can comply with their time constraints.

Modern cars are a good example of time-constrained distributed systems composed of tens of computing nodes interconnected by various types of communication networks that require executing computational intensive applications such as infotainment or a driver assistance applications. Furthermore, other type of applications such as avionic applications, industrial environments, smart city applications, etc., can take advantage of such execution patterns.

The dissertation focused on a particular distributed multi-threaded parallel model; the Fork-Join Parallel/Distributed Real-Time Task Model (P/D Tasks). On the P/D task model, threads start by executing sequentially in a local processor and then fork (distributed-fork) to be executed in parallel in remote processors, when the parallel execution has completed, the results are aggregated by performing a join (distributed-join) operation.

This dissertation presented a framework for the development of parallel and distributed real-time embedded systems that enables high demanding and time-constrained software applications to be distributed cooperatively by both local and distributed processors.

The dissertation addressed three inter-related objectives. The first objective is related to the definition of an *execution model* based on a realistic programming framework:

*Propose an **execution model** for **parallel** and **distributed real-time embedded** platforms and introduce an **easy to use programming framework** for the development of such systems.*

On that matter, Chapter 3 introduced the Fork-Join Parallel/Distributed Real-Time Task Model (P/D task model) which is derived from OpenMP and MPI programming models. A timing model for OpenMP/MPI programs was derived by individually studying the behaviour of typical OpenMP and MPI programs. However, such a model is able to model any program which is based on a distributed fork-join paradigm.

The second objective is related to the *scheduling and allocation of tasks and messages* onto the elements of the distributed system:

*Propose a set of **scheduling algorithms and heuristics** for the allocation of parallel and distributed applications (P/D tasks) onto a distributed system composed of **identical embedded nodes**.*

In relation of that objective, Chapter 4 presented the Partitioned/Distributed-Deadline Monotonic Scheduling (P/D-DMS) algorithm for P/D tasks. The P/D-DMS algorithm is shown to have a resource augmentation bound of 4. Also, it presented the Distributed Stretch Transformation (DST) algorithm which tries to keep as many threads as possible to be executed locally. It is shown that the use of the DST considerably reduces the interference on the processors and the network when scheduling P/D tasks.

Chapter 5 presents the Distributed using Optimal Priority Assignment (DOPA) and the Parallel-DOPA (P-DOPA) heuristics, which partitions a set of sequential and P/D tasks, respectively, and assign their priority by using the Optimal Priority Assignment Algorithm (OPA) (Audsley, 1991). It was confirmed that the use of OPA increases in average the number of task sets that were successfully scheduled when compared with Deadline Monotonic (DM) that it is normally used in other approaches.

Chapter 7 introduces a set of formulations for modelling the allocation of P/D tasks in a distributed multi-core architectures by using a constraint programming approach. The constraint programming formulation is guaranteed to find a feasible allocation, if one exists, in contrast to other approaches based on heuristics.

The third objective is related to providing a technique that confirms if P/D task deadlines are met. This is a problem of providing a *holistic analysis* :

*Propose a **holistic response time analysis** technique that validates the allocations produced by the scheduling algorithms and heuristics proposed in (ii).*

On relation to this objective, Chapter 6 presented a holistic timing analysis for the computation of the Worst-Case Response Time (WCRT) for P/D tasks when transformed by the DST algorithm. The holistic approach considered both synchronous and asynchronous communication patterns. Also, the Flexible Time Triggered - Switched Ethernet (FTT-SE) protocol is considered. Furthermore, it was shown that computation of the WCRT can be improved with respect to traditional approaches by considering a pipeline effect that occurs on those systems.

Finally, the holistic analysis and proposed algorithms in this dissertation are validated through simulations and an experimental evaluation in Chapter 8.

9.2 Future Work

As a future work, it is considered lifting some of the limiting assumptions of the thesis. These extensions can consider both main elements composing a real-time distributed system:

i. a *set of real-time software applications*, and;

ii. a *distributed computing platform*.

Related to the set of software real-time applications, perhaps one of the most limiting assumptions is the consideration of the fork-join paradigm only. A possible extension of this work, it would be to considering a more general approach to model real-time applications; Distributed Acyclic Graphs (DAGs). The consideration of DAGs will allow the modelling of a variety of different execution patterns (precedence constraints) in the software components, for example, nested parallelism that is not considered in this thesis.

Some other assumptions that can be lifted are the assumptions of considering identical WCET for P/D thread $\theta_{i,2j,k}$ within the same P/D segment $\theta_{i,2j}$ and that the number of threads in a P/D segment is smaller or equal than the number of processing nodes (this assumptions are considered in Chapters 4 and 5). these last restrictions allowed the computation of the resource augmentation for the DST algorithm. However, it would be interesting to investigate extensions of those approaches by considering a more general approach.

Related to the distributed computing platform. As explained through this dissertation, the heterogeneity of technologies used in modern cars reflects the variety of requirements from the integrated software components in terms of processing performance and network bandwidth.

Modern automotive applications are an important example of the combination of different architectures: modern vehicles may integrate up to a few ECUs to run hundreds of different functionalities, all connected using various network technologies (e.g., CAN, LIN, FlexRay, FTT-SE, etc.).

In that regard, future investigations may include extensions (i.e., the design of algorithms, heuristics and analysis tools) to consider the different processing capacities of the processing nodes and different combinations of network technologies.

Related to the processing capacity of processing nodes it can be considered:

- *Identical* multi-processor nodes;
- *Uniform* multi-processor nodes.

Related to the heterogeneity of the network technologies. An extension that is being considered is the combination of a high bandwidth real-time network (e.g. FTT-SE), used as a backbone in the system, and several links to sensors that do not have large bandwidth requirements (e.g., CAN).

I believe that the extensions of this dissertation mentioned above, can lead to interesting scientific contributions.

Author's List of Publications

The following publications have been derived from work directly related to this dissertation:

Journals

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, and L.M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems. J Comput. Syst. Sci. (2015). http://dx.doi.org/10.1016/j.jcss.2015.05.005. Impact Factor: **1.091**.
- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. An Improved Holistic Analysis for Fork-Join Parallel Distributed Real-Time Tasks using the FTT-SE Protocol. Selected as candidate paper for a Special Section on *IEEE Transactions of Industrial Informatics* (revision process: second review). Impact Factor: **8.785**.

Conferences and Workshops

- **R. Garibay-Martínez**, L.L. Ferreira, and L.M. Pinho. A framework for the development of parallel and distributed real-time embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 39–46, Sept 2012. doi: 10.1109/SEAA.2012.60.
- **R. Garibay-Martínez**, L.L. Ferreira, C. Maia, and L.M. Pinho. Towards transparent parallel/distributed support for real-time embedded applications. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 114–117, June 2013. doi: 10.1109/SIES.2013.6601483.
- **R. Garibay-Martínez**, G. Nelissen, L. L. Ferreira, and L. M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems. In Marisol García-Valls and Tommaso Cucinotta, editors, *Second International Workshop on Real-time and distributed computing in emerging applications*. Universidad Carlos III de Madrid, 2013b.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, and L.M. Pinho. On the scheduling of fork-join parallel/distributed real-time tasks. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 31–40, June 2014b. doi: 10.1109/SIES.2014.6871184.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Towards holistic analysis for fork-join parallel/distributed real-time tasks. In *Work in Progress Session (ECRTS), 2014 26th Euromicro Conference on Real-Time Systems*, pages 21–24, July 2014a.

- **R. Garibay-Martínez**, G. Nelissen, L. L. Ferreira, and Luís Miguel Pinho. Allocation of parallel real-time tasks in distributed multi-core architectures supported by an ftt-se network. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015, volume 9017 of Lecture Notes in Computer Science*, pages 224–235. Springer International Publishing, 2015.

- **R. Garibay-Martínez**, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Holistic analysis for fork-join distributed tasks supported by the ftt-se protocol. In *Factory Communication Systems (WFCS), 2015 11th World Conference on*, May 2015.

Posters and Demos

- **R. Garibay-Martínez**, L.L. Ferreira, and L.M. Pinho. A framework for the development of parallel and distributed real-time embedded systems. In *Design Tools and Architectures for Multi-Core Embedded Computing Platforms (PARMA-DITAM).* Workshop in conjunction with the 7th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC 2012), 24, Jan, 2012. Paris, France.

- Fábio Oliveira, **Ricardo Garibay-Martínez**, Tiago Cerqueira, Michele Albano, and Luis Lino Ferreira. A module for the ftt-se protocol in ns-3. Demo session in Workshop on ns-3 (WNS3 2015), 2015.

Bibliography

M. Ashjaei, M. Behnam, T. Nolte, and L. Almeida. Performance analysis of master-slave multi-hop switched ethernet networks. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 280–289, June 2013. doi: 10.1109/SIES.2013.6601501.

M. Ashjaei, P. Pedreiras, M. Behnam, R.J. Bril, L. Almeida, and T. Nolte. Response time analysis of multi-hop hartes ethernet switch networks. In *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, pages 1–10, May 2014. doi: 10.1109/WFCS.2014.6837579.

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

N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

Neil C Audsley, Alan Burns, Mike F Richardson, and Andy J Wellings. Real-time scheduling: the deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*. Citeseer, 1991.

Ekain Azketa, Juan P Uribe, J Javier Gutiérrez, Marga Marcos, and Luıs Almeida. Permutational genetic algorithm for the optimized mapping and scheduling of tasks and messages in distributed real-time systems. In *10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, Dec 1990. doi: 10.1109/REAL.1990.128746.

H. Bauer, J. Scharbarg, and C. Fraboul. Worst-case end-to-end delay analysis of an avionics afdx network. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1220–1224, March 2010. doi: 10.1109/DATE.2010.5456993.

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.

Cilk. The cilk project, 2013. URL http://supertech.csail.mit.edu/cilk.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

R.I Davis and A Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *30th IEEE Real-Time Systems Symposium*, pages 398–409, Dec 2009. doi: 10.1109/RTSS.2009.31.

Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978814. URL http://doi.acm.org/10.1145/1978802.1978814.

Robert I Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Schedulability analysis for controller area network (can) with fifo queues priority queues and gateways. *Real-Time Systems*, 49(1):73–116, 2013.

RobertI. Davis, Alan Burns, ReinderJ. Bril, and JohanJ. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35 (3):239–272, 2007. ISSN 0922-6443. doi: 10.1007/s11241-007-9012-7. URL http://dx.doi.org/10.1007/s11241-007-9012-7.

M.L. Dertouzos and A.K. Mok. Multiprocessor misc scheduling of hard-real-time tasks. *Software Engineering, IEEE Transactions on*, 15(12):1497–1506, Dec 1989. ISSN 0098-5589. doi: 10.1109/32.58762.

Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11, 2010.

Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned scheduling of parallel real-time tasks on multiprocessor systems. *ACM SIGBED Review*, 8(3):28–31, 2011.

Nathan Fisher, Sanjoy Baruah, and Theodore P Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *18th Euromicro Conference on Real-Time Systems*, pages 10–pp. IEEE, 2006.

JJ Gutiérrez García and M González Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *Third Workshop on Parallel and Distributed Real-Time Systems*, pages 124–132. IEEE, 1995.

R. Garibay-Martínez, L.L. Ferreira, and L.M. Pinho. A framework for the development of parallel and distributed real-time embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 39–46, Sept 2012. doi: 10.1109/SEAA.2012.60.

R. Garibay-Martínez, L.L. Ferreira, C. Maia, and L.M. Pinho. Towards transparent parallel/distributed support for real-time embedded applications. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 114–117, June 2013a. doi: 10.1109/SIES.2013.6601483.

R. Garibay-Martínez, G. Nelissen, L. L. Ferreira, and L. M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems. In Marisol García-Valls and Tommaso Cucinotta, editors, *Second International Workshop on Real-time and distributed computing in emerging applications*. Universidad Carlos III de Madrid, 2013b.

R. Garibay-Martínez, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Towards holistic analysis for fork-join parallel/distributed real-time tasks. In *Work in Progress Session Real-Time Systems (ECRTS), 2014 26th EUROMICRO Conference on*, pages 21–24, July 2014a.

R. Garibay-Martínez, G. Nelissen, L.L. Ferreira, and L.M. Pinho. On the scheduling of fork-join parallel/distributed real-time tasks. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 31–40, June 2014b. doi: 10.1109/SIES.2014.6871184.

R. Garibay-Martínez, L.L. Ferreira, C. Maia, and L.M. Pinho. Task partitioning and priority assignment for hard real-time distributed systems. *J. Comput. Syst. Sci.*, 2015a. URL http://dx.doi.org/10.1016/j.jcss.2015.05.005.

R. Garibay-Martínez, G. Nelissen, L.L. Ferreira, P. Pedreiras, and L.M. Pinho. Holistic analysis for fork-join distributed tasks supported by the ftt-se protocol. In *Factory Communication Systems (WFCS), 2015 11th World Conference on*, To be presented in May 2015b.

Ricardo Garibay-Martínez, Geoffrey Nelissen, LuisLino Ferreira, and Luís Miguel Pinho. Allocation of parallel real-time tasks in distributed multi-core architectures supported by an ftt-se network. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015*, volume 9017 of *Lecture Notes in Computer Science*, pages 224–235. Springer International Publishing, 2015. ISBN 978-3-319-16085-6. doi: 10.1007/978-3-319-16086-3_18. URL http://dx.doi.org/10.1007/978-3-319-16086-3_18.

Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling, 1996. URL https://hal.inria.fr/inria-00073732. Projet REFLECS.

J. Goncalves, L.L. Ferreira, L.M. Pinho, and G. Silva. Handling mobility on a qos-aware service-based framework for mobile systems. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 97–104, Dec 2010. doi: 10.1109/EUC.2010.24.

J.J. Gutierrez Garcia, J.C.P. Gutierrez, and M. Gonzalez Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 15–24, 2000. doi: 10.1109/EMRTS.2000.853988.

J.Javier Gutiérrez, J.Carlos Palencia, and Michael González Harbour. Holistic schedulability analysis for multipacket messages in afdx networks. *Real-Time Systems*, 50 (2):230–269, 2014. ISSN 0922-6443. doi: 10.1007/s11241-013-9192-2. URL http://dx.doi.org/10.1007/s11241-013-9192-2.

Intel. Intel cilk plus, January 2013. URL http://software.intel.com/en-us/intel-cilk-plus/.

Mathai Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

A. Kanevsky, A. Skjellum, and A. Rounbehler. Mpi/rt-an emerging standard for high-performance real-time systems. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 3, pages 157–166 vol.3, 1998. doi: 10.1109/HICSS.1998.656130.

Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and*

Parallel Computing, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88139-1. doi: 10.1007/978-3-540-88140-7_24. URL http://dx.doi.org/10.1007/978-3-540-88140-7_24.

Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997. ISBN 0792398947.

Hermann Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (tte) design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33, May 2005. doi: 10.1109/ISORC.2005.56.

K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 259–268, Nov 2010. doi: 10.1109/RTSS.2010.42.

Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

Hyung-Taek Lim, L. Volker, and D. Herrscher. Challenges in a future ip/ethernet-based in-car network for real-time applications. In *DAC'11*, pages 7–12, June 2011.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL http://doi.acm.org/10.1145/321738.321743.

J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 13–22, June 2004. doi: 10.1109/EMRTS.2004.1310992.

R. Marau. *Real-time communications over switched ethernet supporting dynamic qos management*. U. Aveiro, 2009.

R. Marau. Implementation of the ftt-se protocol for linux. http://paginas.fe.up.pt/~ftt/sections/Repository/index.html, January 2015.

R. Marau, L. Almeida, and P. Pedreiras. Enhancing real-time communication over cots ethernet switches. In *Factory Communication Systems, 2006 IEEE International Workshop on*, pages 295–302, 2006. doi: 10.1109/WFCS.2006.1704170.

R. Marau, M. Behnam, Z. Iqbal, P. Silva, L. Almeida, and P. Portugal. Controlling multi-switch networks for prompt reconfiguration. In *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*, pages 233–242, May 2012. doi: 10.1109/WFCS.2012.6242571.

Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004. ISBN 0321228111.

Alexander Metzner and Christian Herde. Rtsat–an optimal and efficient approach to the task allocation problem in distributed architectures. In *27th IEEE International Real-Time Systems Symposium*, pages 147–158. IEEE, 2006.

MPI-Forum. Mpi: A message-passing interface standard version 2.2, April 2012. URL http://www.mpi-forum.org/docs/docs.hmtl/.

ns 3. ns-3 manual, May 2015. URL https://www.nsnam.org/docs/manual/html/organization.html/.

F. Oliveira. *Implementação da rede Flexible Time-triggered para Switched Ethernet no Simulador NS-3*. IPP ISEP, 2015.

Fábio Oliveira, Ricardo Garibay-Martínez, Tiago Cerqueira, Michele Albano, and Luis Lino Ferreira. A module for the ftt-se protocol in ns-3. 2015.

OpenMP-Arch-Rev-Board. Openmp application program interface v3.1 july 2011, April 2012. URL http://www.openmp.org/wp/openmp-specifications/.

J.C. Palencia and M.G. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 3–12, July 2003. doi: 10.1109/EMRTS.2003.1212721.

José Carlos Palencia and Michael Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *The 19th IEEE Real-Time Systems Symposium, 1998. Proceedings*, pages 26–37. IEEE, 1998.

José Carlos Palencia and Michael Gonzalez Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *The 20th IEEE Real-Time Systems Symposium, 1999. Proceedings*, pages 328–339. IEEE, 1999.

P. Pedreiras, P. Gai, L. Almeida, and G.C. Buttazzo. Ftt-ethernet: a flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems. *Industrial Informatics, IEEE Transactions on*, 1(3):162–172, Aug 2005. ISSN 1551-3203. doi: 10.1109/TII.2005.852068.

Paulo Pedreiras and Almeida Luis. The flexible time-triggered (ftt) paradigm: an approach to qos management in distributed real-time systems. In *International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

Manar Qamhieh, Frédéric Fauberteau, Serge Midonnet, et al. Performance analysis for segment stretch transformation of parallel real-time tasks. In *5th Junior Researcher Workshop on Real-Time Computing*, pages 29–32, 2011.

K. Ramamritham, J.A. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):184–194, Apr 1990. ISSN 1045-9219. doi: 10.1109/71.80146.

Michael Richard, Pascal Richard, and Francis Cottet. Allocating and scheduling tasks in multiple fieldbus real-time systems. In *IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 137–144. IEEE, 2003.

Rietveld-Code-Review-Tool. Issue 187880044: full duplex extensions for csma. https://codereview.appspot.com/187880044/, June 2015.

A. Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 217–226, Nov 2011. doi: 10.1109/RTSS.2011.27.

Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49 (4):404–435, 2013.

Lui Sha, Tarek Abdelzaher, Karl-Erik Arzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, November 2004. ISSN 0922-6443. doi: 10.1023/B:TIME.0000045315.61234.1e. URL http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e.

Marco Spuri. Holistic Analysis for Deadline Scheduled Real-Time Distributed Systems, 1996. URL https://hal.inria.fr/inria-00073818. Projet REFLECS.

R. Stafford. Random vectors with fixed sum. In *[misc]: http://www.mathworks.com/matlabcentral/fileexchange/9700*, 2004.

POSIX Standard. Posix threads programming, 2012. URL https://computing.llnl.gov/tutorials/pthreads.

J.A. Stankovic. misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, Oct 1988. ISSN 0018-9162. doi: 10.1109/2.7053.

Andrew S Tanenbaum. *Distributed operating systems*. Pearson Education India, 1995.

K. Tindell, A. Burns, and A.J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995. ISSN 0922-6443. doi: 10.1007/BF01088855. URL http://dx.doi.org/10.1007/BF01088855.

Ken Tindell and Alan Burns. Guaranteeing message latencies on control area network (can). In *Proceedings of the 1st International CAN Conference*, 1994.

Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, April 1994. ISSN 0165-6074. doi: 10.1016/0165-6074(94)90080-9. URL http://dx.doi.org/10.1016/0165-6074(94)90080-9.

Ken W Tindell, Alan Burns, and Andy J. Wellings. Allocating hard real-time tasks: an np-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

Ian Vessey and Glenn Skinner. Implementing berkeley sockets in system v release 4. In *PROCEEDINGS OF THE WINTER 1990 USENIX CONFERENCE*, pages 177–193, 1990.

Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *28th IEEE International Real-Time Systems Symposium*, pages 161–170. IEEE, 2007.

Qi Zhu, Yang Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli. Optimizing the software architecture for extensibility in hard real-time distributed systems. *Industrial Informatics, IEEE Transactions on*, 6(4):621–636, Nov 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2053938.

Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Trans. Embed. Comput. Syst.*, 11(4):85:1–85:30, January 2013. ISSN 1539-9087. doi: 10.1145/2362336.2362352. URL http://doi.acm.org/10.1145/2362336.2362352.