Design of Mixed-Criticality Applications on Distributed Real-Time Systems

Tamas-Selicean, Domitian

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Design of Mixed-Criticality Applications on Distributed Real-Time Systems

Domiţian Tămaş–Selicean

Kongens Lyngby 2014
PhD-2014-329

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253031, Fax +45 45881399
compute@compute.dtu.dk
www.compute.dtu.dk
PhD-2014-329

Summary

A mixed-criticality system implements applications of different safety-criticality levels onto the same platform. In such cases, the certification standards require that applications of different criticality levels are protected so they cannot influence each other. Otherwise, all tasks have to be developed and certified according to the highest criticality level, dramatically increasing the development costs. In this thesis we consider mixed-criticality real-time applications implemented on distributed partitioned architectures.

Partitioned architectures use temporal and spatial separation mechanisms to ensure that applications of different criticality levels do not interfere with each other. With temporal partitioning, each application is allowed to run only within predefined time slots, allocated on each processor. The sequence of time slots for all the applications on a processor are grouped within a Major Frame, which is repeated periodically. Each partition can have its own scheduling policy; we have considered non-preemptive static cyclic scheduling and fixed-priority preemptive scheduling policies. We assume that the communication network implements the TTEthernet protocol, which supports Time-Triggered (TT) messages transmitted based on static schedule tables, Rate Constrained (RC) messages with bounded end-to-end delay, and Best-Effort (BE) messages, for which no timing guarantees are provided. TTEthernet offers spatial separation for mixed-criticality messages through the concept of virtual links, and temporal separation, enforced through schedule tables for TT messages and bandwidth allocation for RC messages.

The objective of this thesis is to develop methods and tools for distributed mixed-criticality real-time systems. At the processor level, we are interested to determine (i) the mapping of tasks to processors, (ii) the assignment of tasks to partitions, (iii) the decomposition of tasks into redundant lower criticality tasks, (iv) the sequence and size of the partition time slots on each processor and (v) the schedule tables, such that all the applications are schedulable and the development and certification costs are minimized. We have proposed Simulated Annealing and Tabu Search metaheuristics to solve these

optimization problems. The proposed algorithms have been evaluated using several benchmarks.

At the communication network level, we are interested in the design optimization of TTEthernet networks used to transmit mixed-criticality messages. Given the set of TT and RC messages, and the topology of the network, we are interested to optimize (i) the packing of messages in frames, (ii) the assignment of frames to virtual links, (iii) the routing of virtual links and (iv) the TT static schedules, such that all frames are schedulable and the worst-case end-to-end delay of the RC messages is minimized. We have proposed a Tabu Search-based metaheuristic for this optimization problem. The proposed algorithm has been evaluated using several benchmarks.

The optimization approaches have also been evaluated using realistic aerospace case studies. In this context, we have shown how to extend the proposed optimization frameworks to also take into account quality of service constraints. For TTEthernet networks, we have also proposed a topology selection method to reduce the cost of the architecture.

Summary (Danish)

I et blandet sikkerhedskritisk system implementeres applikationer med forskellige sikkerhedskritikalitetsniveauer på den samme platform. I sådanne tilfælde kræver certificeringsstandarderne, at applikationer fra forskellige sikkerhedskritisk niveauer er beskyttede, så de ikke kan påvirke hinanden. Alternativet ville være at alle opgaver udvikles og certificeres i henhold til det højeste sikkerhedskritisk niveau, hvilket ville øge udviklingsomkostningerne dramatisk. I denne afhandling betragter vi blandet sikkerhedskritisk realtidsapplikationer implementeret på distribuerede partitionerede arkitekturer.

Partitionerede arkitekturer benytter tidsmæssige og rumlige separationsmekanismer for at sikre, at applikationer fra de forskellige sikkerhedskritisk niveauer ikke forstyrrer hinanden. I den tidsmæssige opdeling får hver applikation kun lov til at køre inden for fastlagte tidsintervaller på hver processor. De enkelte tidsintervaller for alle applikationerne på en processor, er grupperet i en bestemt rækkefølge i en Major Frame, der gentages med jævne mellemrum. Hver partition kan have sin egen planlægnings politik; vi har betragtet ikke-forebyggende statisk cyklisk planlægning og fast prioriterede planlægningspolitikker. Vi antager, at kommunikationsnetværket implementerer TTEthernet-protokollen, som understøtter tidsudløste (TT) meddelelser, der udsendes ud fra statiske planlægningstabeller samt rate begrænsede (RC) meddelelser, med afgrænset ende-til-ende forsinkelse, ligesom den understøtter bedste forsøgs (BE) beskeder, for hvilke der ikke gives timing-garantier. TTEthernet tilbyder rumlig separation af blandet sikkerhedskritisk beskeder via virtuelle forbindelser, og tidsmæssig separation, og de gennemføres ved tidsplanstabeller for TT beskeder og båndbreddetildeling for RC-meddelelser.

Formålet med denne afhandling er at udvikle metoder og værktøjer til distribuerede blandet sikkerhedskritisk realtidssystemer. På processorniveau, er vi interesseret i gøre rede for (i) en kortlægning af opgaver til processorer, (ii) en tildeling af opgaver til partitioner, (iii) en nedbrydning af opgaver i redundant mindre kritiske opgaver, (iv) sekvensen og størrelse af partition-tidsintervaller på hver processor og (v) tidsplanstabeller, således at alle applikationer kan planlægges og at udviklings- og certifice-

ringsomkostninger minimeres. Vi har foreslået Simulated Annealing and Tabu Search metaheuristikker til at løse disse optimeringsproblemer. De foreslåede algoritmer er blevet evalueret ved hjælp af flere benchmarks.

På kommunikations- og netværksniveau, er vi interesserede i designoptimering af TTEthernet netværk, der anvendes til at overføre blandet sikkerhedskritisk beskeder. Givet et sæt af TT- og RC-beskeder, og topologien af netværket, er vi interesserede i at optimere (i) pakning af meddelelser i frames, (ii) tildeling af frames til virtuelle links, (iii) routing af virtuelle links og (iv) TT statiske tabeller, således at alle frames kan planlægges og den værst tænkelige ende-til-ende forsinkelse af RC-beskeder vil blive minimeret. Vi har foreslået en Tabu Search-baseret metaheuristik til dette optimeringsproblem. Den foreslåede algoritme er blevet evalueret ved hjælp af flere benchmarks.

Optimeringsmetoderne er desuden blevet evalueret ved brug af realistiske rumfartcasestudier. I denne sammenhæng har vi vist, hvordan man kan udvide det foreslåede optimerings-framework til også at tage hensyn til service og kvalitetsbegrænsninger. For TTEthernet netværk har vi også foreslået en topologi udvælgelses metode, der reducerer omkostningerne i arkitekturen.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science, Technical University of Denmark in fulfillment of the requirements for acquiring the Ph.D. degree in computer engineering.

The thesis deals with methods and tools for the optimization of mixed-criticality real-time embedded systems, to support the system engineers in the early life cycles phases, where the impact of design decisions is greatest.

The work has been supervised by Associate Professor Paul Pop and co-supervised by Professor Jan Madsen.

Lyngby, 31 January 2014

Domiţian Tămaş–Selicean

Papers Included in the Thesis

- Domiţian Tămaş–Selicean and Paul Pop. Optimization of Time-Partitions for Mixed-Criticality Real-Time Distributed Embedded Systems. *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 2011. Published.
- Domiţian Tămaş–Selicean and Paul Pop. Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures. *Proceedings of the Real-Time Systems Symposium*, pp. 24–33, 2011. Published.
- Domiţian Tămaş–Selicean and Paul Pop. Task Mapping and Partition Allocation for Mixed-Criticality Real-Time Systems. *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pp. 282–283, 2011. Published.
- Sorin O. Marinescu, Domiţian Tămaş–Selicean and Paul Pop. Timing Analysis of Mixed-Criticality Hard Real-Time Applications Implemented on Distributed Partitioned Architectures. *Proceedings of the International Conference on Emerging Technologies and Factory Automation*, pp. 1–4, 2012. Published.
- Domiţian Tămaş–Selicean, Paul Pop and Wilfried Steiner. Synthesis of Communication Schedules for TTEthernet-Based Mixed-Criticality Systems. *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pp. 473–482, 2012. Published.
- Domiţian Tămaş–Selicean, Sorin O. Marinescu and Paul Pop. Analysis and Optimization of Mixed-Criticality Applications on Partitioned Distributed Architectures. *Proceedings of the IET System Safety Conference*, pp. 1–6, 2012. Published.

- Domiţian Tămaş–Selicean, Didier Keymeulen, Dan Berisford, Robert Carlson, Kevin Hand, Paul Pop, Winthrop Wadsworth and Ralph Levy. Fourier Transform Spectrometer Controller for Partitioned Architectures. *Proceedings of the Aerospace Conference*, pp. 1–11, 2013. Published.
- Domiţian Tămaş–Selicean and Paul Pop. Design Optimization of Mixed-Criticality Real-Time Systems. Under review in *ACM Transactions on Embedded Computing*.
- Domiţian Tămaş–Selicean, Paul Pop and Wilfried Steiner. Design Optimization of TTEthernet-based Distributed Real-Time Systems. Under review in *Real-Time Systems Journal*.

Acknowledgements

I cannot thank Paul Pop enough for his tireless help, continuous involvement and friendship. His guidance and encouragement were a motivating force throughout my PhD. I also want to thank Jan Madsen for our insightful conversations and the opportunities he has created. My Phd studies were not only extremely interesting, but also inspiring due to their supervision, help and support.

I would also like to thank my friends and colleagues at DTU Compute, for creating a fun work environment and making the PhD life more pleasant. Special thanks goes to Karin Tunder for her constant help with the administrative tasks and impromptu Danish lessons, to Mads Ingwar for translating the summary, and to Michael Reibel Boesen for his help and door-opening networking. I appreciate the support provided by the administrative and technical staff.

I am grateful to Wilfried Steiner from TTTech Computertechnik AG, Vienna for our collaboration and his hospitality. His in-depth suggestions have been of big help in this work. Many thanks to the people at the Jet Propulsion Laboratory, NASA for the hospitality, and especially to Didier Keymeulen, for his guidance during my external stay and for making a dream come true. I am grateful to the ARTEMIS Joint Undertaking for funding my PhD project, and to Oticon Fonden and Otto Mønsteds Fond for funding my external research stay.

Last but not least, I am profoundly grateful for their love and patience to my family and friends.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Meaning
ABS	Anti-lock Braking System
ADC	Analog-Digital Converter
AGC	Apollo Guidance Computer
ASAP	As-Soon-As-Possible (ASAP)
ASIL	Automotive Safety Integrity Level
AVB	Audio Video Bridging
BAG	Bandwidth Allocation Gap
BE	Best-Effort
BEO	Beyond Earth Orbit
CA	Certification Authority
CAN	Controller Area Network
CBS	Constant Bandwidth Server
CEV	Crew Exploration Vehicle
ChA ChB	Channel A, Channel B
ChI	Channel I
CIRIS	Compositional InfraRed Imaging Spectrometer
CLG	Candidate List Generation
COCOMO	Constructive Cost Model
CM	Crew Module
CPU	Central Processing Unit
cRIO	CompactRIO
DAC	Digital-Analog Converter
DAL	Design Assurance Level
DC	Development Cost
DOTTS	Design Optimization of TTEthernet-based Systems

DSE	Design Space Exploration
ECU	Electronic Control Unit
EDF	Earliest Deadline First
ES	End System
ESA	European Space Agency
ESes	End Systems
ESM	European Service Module
ET	Event-Triggered
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FPS	Fixed-Priority Scheduling
FTIR	Fourier Transform Infrared Spectrometer
FTS	Fourier Transform Spectrometer
FU	Filtering Unit
IAP	Interferogram Acquisition Process
IC	Integrated Circuit
ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
ISS	International Space Station
JPL	Jet Propulsion Laboratory
LAS	Launch Abort System
LCM	Least-Common Multiplier
LS	List Scheduling
MCDO	Mixed-Criticality Design Optimization
MCU	Motor Control Unit
MESUR	Mars Environment Survey Pathfinder
MF	Major Frame
MMU	Memory Management Unit
MO	Mapping Optimization
MO+PO	Optimization approach where MO is performed separately from PO
MPCV	Multi-Purpose Crew Vehicle
MPO	Simultaneous Mapping and Partitioning Optimization
NASA	National Aeronautics and Space Administration
NC	Non-Critical
NIC	Network Interface Card
NRE	Non-Recursive Engineering
NS	Network Switch
NSes	Network Switches
OPD	Optical Path Difference
PA	Partitioned Architecture

PC	Personal Computer
PCIP	Priority-and-Criticality Inversion Protocol
PCCP	Priority-and-Criticality Ceiling Protocol
PE	Processing Element
PFO	Packing and Fragmenting Optimization
PO	Partitioning Optimization
QoS	Quality-of-Service
RC	Rate-Constrained
RM	Rate Monotonic
RO	Routing Optimization
ROM	Read-only Memory
RTOS	Real-Time Operating System
SA	Simulated Annealing
SC	Safety-Critical
SCA	Spacecraft Adapter
SCS	Static Cyclic Scheduling
SDI	Stepper Drive Interface
SFI	Software Fault Isolation
SIL	Safety-Integrity Level
SM	Service Module
SMT	Satisfiability Modulo Theory
SNR	Signal-to-Noise Ratio
SO	Scheduling Optimization
SS	Straightforward Solution
SWaP	Size, Weight and Power
SysML	Systems Modeling Language
TDMA	Time-Division Multiple Access
TI	Initial Temperature
TL	Temperature Length
TP	Traffic Policing task
TPO	Time-Partitions Optimization
TR	Traffic Regulator task
TS	Tabu Search
TT	Time-Triggered
TTP	Time-Triggered Protocol
VL	Virtual Link
WCET	Worst-Case Execution Time
WCD	Worst-Case end-to-end Delay
ZPD	Zero Optical Path Difference

Notations

Notation	Meaning
Γ	Set of all applications in the system
$\mathcal{A}_i \in \Gamma$	An application
$\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$	Application graph for application $\mathcal{A}_i$
$\mathcal{V}_i$	Set of all the nodes in the application graph
$\mathcal{E}_i$	Set of all the edges in the application graph
$\mathcal{V}$	Set of all the tasks in the system
$\mathcal{E}$	Set of edges in the protection requirements graph
$\tau_j \in \mathcal{V}_i$	One task of the application graph
$M : \mathcal{V}_i \rightarrow \mathcal{N}$	The mapping of tasks to processing elements
M°	Initial mapping of tasks to processing elements
$\mathcal{L}$	The library of SIL decompositions
$D(\tau_i) : \mathcal{V} \rightarrow \mathcal{D}_i$	SIL decomposition function
D°	Initial decomposition of tasks, tasks are not decomposed
$\mathcal{D}_i$	Set of SIL decomposition options
$\mathcal{N}$	Set of all the PEs in the architecture
N_i	A processing element
$e_{jk} \in \mathcal{E}_i$	Edge in the application graph, indicates that the output of τ_j is the input of τ_k
m_i	Message in the application graph
$D_{\mathcal{G}_i}$	Deadline of application graph $\mathcal{G}_i$
$T_{\mathcal{G}_i}$	Period of application graph $\mathcal{G}_i$
$C_i^{N_j}$	WCET for τ_i on each PE N_j where τ_i is considered for mapping

R_i	Response time of application $\mathcal{A}_i$
D_i	Deadline of application $\mathcal{A}_i$ or of task τ_i
T_i	Period of task τ_i
$Cost_{TPO}$	Cost function used by TPO
$Cost_{MCDO}$	Cost function used by MCDO
δ	Degree of schedulability
δ_{sched}	Percentage improvement in the degree of schedulability
δ_{SC}	Degree of schedulability for SC applications
δ_{NC}	Degree of schedulability for NC tasks
w_{SC}	Weight for SC application, used by $Cost_{TPO}$
w_{NC}	Weight for NC tasks, used by $Cost_{TPO}$
Γ_{QoS}	Subset of the soft real-time applications
$QoS(\mathcal{A}_i)$	The quality of service for the soft real-time application $\mathcal{A}_i$
$SIL(\tau_i)$	The SIL of task τ_i
$DC(\tau_i, SIL\ j)$	The development cost of τ_i and SIL j
$DC(\mathcal{A}_i)$	The development cost of the application $\mathcal{A}_i$
$DC(\Gamma)$	The development cost for the set of all the applications
δ_{DC}	Increase in the development costs
$\Pi(\mathcal{V}, \mathcal{E})$	Protection requirements graph
$sr_{ij} \in \mathcal{E}$	An edge in Π between τ_i and τ_j means that the two tasks are not allowed to share a partition
$\phi : \mathcal{V} \rightarrow \mathcal{P}$	Assignment of tasks to partitions
ϕ°	Initial assignment of tasks to partitions
$\mathcal{P}$	Set of partitions
$\mathcal{P}^\circ$	Initial set of partitions
P_j	Partition
$\mathcal{P}_{ij}$	Set of partition slices of P_j on N_i
p_{ij}^k	The k^{th} partition slice of partition P_j on N_i
T_{cycle}	System cycle
T_{MF}	Period of the major
t_O	Partition switch overhead
$\mathcal{S}$	Schedule tables
Ψ	Implementation
L	Tabu list, or tabu history
l	Tabu tenure
$\mathcal{C}$	Candidate list
L_{ready}	Sorted priority list used by List Scheduling
ε	Cooling ratio, used by SA

$\mathcal{G}_C(\mathcal{V}_C, \mathcal{E}_C)$	TTEthernet cluster
$\mathcal{V}_C = \mathcal{ES} \cup \mathcal{NS}$	Set of all the end systems and network switches in the cluster
$\mathcal{ES}$	Set of all the end systems in the cluster
ES_i^{src}	The source end system for frame f_i
$\mathcal{ES}_i^{dest}$	The set of destination end systems for frame f_i
$\mathcal{NS}$	Set of all the network switches in the cluster
$\mathcal{NS}^\star$	Subset of network switches that are fixed in the cluster and can not be removed
ES_i	An end system
NS_i	A network switch
$\mathcal{E}$	Set of physical links
$\mathcal{DL}$	Set of dataflow links in the cluster
$\mathcal{DL}^\star$	Subset of dataflow links that are fixed in the cluster and can not be removed
dl_i	A dataflow link
$\mathcal{DP}$	Set of dataflow paths in the cluster
dp_i	A dataflow path
$\mathcal{VL}$	Set of virtual links in the cluster
vl_i	A virtual link
$BAG(vl_i)$	The BAG of vl_i
$\mathcal{B}$	The set of BAG for all VLs
$\mathcal{B}^\circ$	The initial set of BAG for all VLs
$\mathcal{R}_{VL}(vl_i)$	The routing of virtual link vl_i
$\mathcal{R}^\circ$	The initial routing of virtual links
$\mathcal{M} = \mathcal{M}^{TT} \cup \mathcal{M}^{RC} \cup \mathcal{M}^{BE}$	Set of all the messages in the cluster
$\mathcal{M}^{TT}$	Set of the TT messages in the cluster
$\mathcal{M}^{RC}$	Set of the RC messages in the cluster
$\mathcal{M}^{BE}$	Set of the BE messages in the cluster
m_i	A message
$m_i.rate$	The rate of message m_i
$m_i.period$	The period of message m_i
$m_i.deadline$	The deadline of message m_i
$m_i.size$	The size of message m_i
$\Phi_m(m_i)$	Fragmenting of message m_i into message fragments
Φ_m°	The initial fragmenting of messages into message fragments
$\mathcal{M}^+$	The set of message fragments and messages which were not fragmented
$\mathcal{K}$	The packing of messages

$\mathcal{K}^{\circ}$	The initial packing of messages
$\mathcal{F}$	Set of all the frames in the cluster
$\mathcal{F}^{TT}$	Set of the TT frames in the cluster
$\mathcal{F}^{RC}$	Set of the RC frames in the cluster
$\mathcal{F}^{BE}$	Set of the BE frames in the cluster
f_i	A frame
$f_i.deadline$	Deadline of f_i
$f_i.size$	Size of f_i
$f_i.offset$	Offset, i.e., send time relative to the start of the period of frame f_i
$f_{i,x}$	x^{th} instance of frame f_i
$f_{i,x}.jitter$	Jitter for $f_{i,x}$
$f_{i,x}^{dl_j}$	The x^{th} instance of f_i on dataflow link l_j
$pred(f_{i,x}^{dl_j})$	The set of predecessor frame instances of $f_{i,x}$ on l_j
$succ(f_{i,x}^{dl_j})$	The set of successor frame instances of $f_{i,x}$ on l_j
R_{f_i}	Worst-case delay of f_i
$\mathcal{M}_F(f_i)$	The assignment of frame to virtual links
$\mathcal{M}_F^{\circ}$	The initial assignment of frames to virtual links
$C_j^{[\nu_m,\nu_n]}$	Transmission duration of f_j on dataflow link $[\nu_m,\nu_n]$
$B_{1,Tx}$	A transmission buffer
$Q_{1,Tx}$	A transmission queue in an ES
Q_{Tx}	An outgoing queue in a NS
$Q_{1,Rx}$	Receiving queue in a ES
TT_R	Receiver task for TT frames
TT_S	Sender task for TT frames
RC_S	Scheduler task for RC frames
$\mathcal{S}$	Complete set of local schedules in a cluster
$\mathcal{S}^{\circ}$	Initial set of local schedules in a cluster
$\mathcal{S}_R$	Receive schedule
$\mathcal{S}_S$	Send schedule
$BW(vl_i)$	The maximum bandwidth required by vl_i
$Cost_{DOTTS}$	Cost function used by DOTTS
δ_{TT}	Degree of schedulability for the TT frames
δ_{RC}	Degree of schedulability for the RC frames
w_{TT}	Weight for the TT frames, used to compute the cost function
w_{RC}	Weight for the RC frames, used to compute the cost function

Best	The best-so-far solution in the Tabu Search algorithm
Current	The current solution in the Tabu Search algorithm
Next	The solution selected as the next solution in the Tabu Search algorithm
tabu(*Next*)	The tabu of the move that generated the *Next* solution
$hr_{TT}^{l_x}$	The TT frame with the highest rate on dataflow link l_x
$lg_{TT}^{l_x}$	The largest TT frame on dataflow link l_x
$BW_{Avail}(l_j)$	The available bandwidth on dataflow link l_j
$BW_{Req}(l_j)$	The required bandwidth on dataflow link l_j
$BW_{\%}^{BW}$	The percentage of BE messages that have their bandwidth requirements met
MESUR-HC	High-criticality MESUR tasks
MESUR-LC	Loc-criticality MESUR tasks
w_i	Busy window
ϕ_i	The earliest activation of task τ_i relative to the occurrence of the triggering event
φ_i	The time interval between the critical instant and earliest time of the first activation of τ_i
B_i	The maximum blocking time for τ_i
t_c	Critical instant
$p_{0,i}$	The index of the first pending instance of τ_i
n_{ia}	The number of pending τ_i jobs at t_c
$\text{hp}(\tau_i)$	Higher priority tasks from the same application as τ_i
$W_i(\tau_i, w_i)$	Interference from $\text{hp}(\tau_i)$
$hp_a(\tau_i)$	Higher priority tasks from other applications than τ_i
$W_a^*(\tau_i, w_i)$	Worst-case interference from $hp_a(\tau_i)$
Δ_{NC}	Percentage increase in the degree of schedulability for non-critical tasks
Δ_{SC}	Percentage increase in the degree of schedulability for safety-critical tasks
Υ	Implementation at the communication level
C_i	Transmission time for frame f_i
M_f	Fixed mirror
M_m	Mobile mirror
br	Recombined beam
x_o	Initial mirror position

x_{max}	Final mirror position
ν	Wavenumber
$\Delta(\nu)$	Spectrometer resolution
S_{OE}	Optical encoder signals
S_{MA}	Motor alignment signal
T_L	Temperature low
T_H	Temperature high
$S(T,\nu)$	Measured spectrum at temperature T and wavenumber ν
$B(T,\nu)$	Spectral radiance at the surface of the blackbody for temperature T and wavenumber ν
$T_{Calibrated}$	Calibrated transmitted spectrum
$T(\nu)$	Transmittance at wavenumber ν
$T_{Sample}(\nu)$	Sample transmitted spectrum at wavenumber ν
$T_{Background}(\nu)$	Background transmitted spectrum at wavenumber ν
$A(\nu)$	Absorbance at wavenumber ν
$N(\nu)$	Spectral noise at wavenumber ν
$N_{rms}(\nu)$	Root-mean-square of the spectral noise at wavenumber ν

CHAPTER 1

Introduction

The Apollo Guidance Computer (AGC) [180] that landed the Apollo 11 on the moon, in July 1969, was powered by a 16-bit processor running at 2.048 MHz. AGC, with an estimated cost of 25 mil. USD in 1969, had a random access memory (RAM) of 2048 words and a read-only memory (ROM) of 36K. In contrast, almost 45 years later, in 2013, the top smartphones contain processors a thousand times more powerful than the AGC. For example, the Apple iPhone 5s [21], for less than a 1000 USD, contains a 64-bit dual-core processor at 1.3 GHz, with a minimum of 16 GB of ROM.

Since the invention of integrated circuits (ICs) in the 1950s, the number of transistors on ICs is doubling every two years, following Moore's Law [119]. This doubling increases the processing power of the processors and reduces costs, among others, enabling the proliferation of computing devices into our everyday life. Most people think only of desktop and portable computers, when they refer to computing devices. But only 2% [73] of the manufactured microprocessors are used in such general purpose computers. The rest of 98% are used in embedded systems.

An *embedded system* is a computer-based system embedded in a larger system that it controls, repeatedly carrying out a particular function and not designed to be programmed by the end user in the same way that a personal computer (PC) is [88, 173]. That is, any special purpose computer-based system, other than a general purpose computer, is an embedded system [173].

Few people consider that their refrigerator, mobile phone, car, toothbrush, pacemaker or hearing aid contain computing devices. Embedded system are all around us. According to Ebert and Jobes [73], in 2008 there were around 30 embedded microprocessors for each person in the developed countries. Current research estimates that by as early as 2020, there will be 100 processors per person [128], with more than 50 billions connected devices [14]. Embedded systems "enable an every-day object to become a smart object able to communicate with other smart objects" [11], improving our lives.

The Embedded Systems Institute [16] classifies embedded systems as: (1) *infrastructure systems*, i.e., public or industrial systems with a priority on efficient resource management and control of safety-critical actions; (2) *professional systems*, i.e., systems used for work-related purposes (e.g.,medical equipment); (3) *automotive systems*; (4) *consumer electronics*, e.g., domestic appliances or mobile devices; (5) *avionics* and (6) *defense systems*.

1.1 Mixed-Criticality Systems

Many embedded systems are *real-time systems*, in which "the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced" [102]. Examples of real-time systems are a vehicle's cruise control and anti-lock braking system (ABS), and digital audio players. One key characteristic of any real-time system is its *deadline*, which is the latest time instant when the system must complete its execution, or a result must be produced. Depending on the consequences of missing the deadline, real-time systems can be *soft* or *hard*. *Soft real-time systems* can miss the deadlines once in a while, as the system will still function, but with degraded service. Example of such systems are home entertainment systems and mobile phones. In *hard real-time systems*, missing a deadline will lead to the failure of the system. ABS is an example of hard real-time system.

A *safety-critical system* is a system whose failure might endanger human life or the environment. Examples of safety-critical systems are aircraft control systems and pacemakers. *Safety-Integrity Levels* (SILs) are assigned to safety-related functions to capture the required level of risk reduction, and will dictate the development processes and certification procedures that have to be followed [94], [96], [140]. A *mixed-criticality* system is "an integrated suite of hardware, operating system and middleware services and application software that supports the execution of safety-critical, mission-critical, and non-critical software within a single, secure computing platform" [35]. The embedded systems in a vehicle form a mixed-criticality system, as they implement safety-critical applications (e.g., ABS) and non-critical applications (e.g., diagnostics software). SILs are standard specific, but in this thesis we consider that there are four

SIL levels, ranging from SIL 4 (most critical) to SIL 1 (least critical). Certification standards require that safety functions of different criticality levels are *protected* (or, *isolated*), so they cannot influence each other. For example, without protection, a lower-criticality task could corrupt the memory of a higher-criticality task.

In this thesis, we are interested in the design optimization of mixed-criticality real-time applications. We consider heterogenous distributed platforms, consisting of several processing elements (PEs) interconnected using the TTEthernet communication protocol [28]. We assume that the PEs provide mechanisms similar to "Integrated Modular Avionics" (IMA) [3, 141] to enforce enough separation for the mixed-criticality applications. At the network level, TTEthernet offers separation for messages of different time- and safety-criticality.

More and more functionalities are implemented using embedded systems. A good example is the automotive industry, where such embedded systems are called "electronic control units" or ECUs. One of the first ECUs mass-produced and installed in a car was the Bosch D-Jetronic fuel injection system, in 1967 [71, 50, 44]. After 40 years, premium-class cars contain between 70 to 100 ECUs, with software and electronics representing 34–40% of the cost of the car [59]. The software running in ECUs has increased by 25% between 2011 and 2013 [52], and is expected that more than 80 percent of future car innovations, like the "connected car" or self-driving cars, will come from such electronic systems [59].

Unfortunately, this increase in microprocessor implemented functionality translates into an increase in the interactions between the components and in the complexity of the whole system. Embedded systems developers and engineers identify system complexity and system integration as top technical challenges [52, 18, 106], with one study [106] observing that the issue of complexity rose significantly in avionics projects, from 31% to 51.9% between 2012 and 2013. The cost to develop and implement embedded systems, relative to the final product price, varies from area to area. These costs represent 36% in the automotive area, 22% in industrial automation, 37% in the telecommunication area, 41% in the consumer electronics and 33% in medical devices [6]. Thus, new design methods are needed to tackle this complexity increase and help engineer productivity (see Section 1.3).

1.2 Partitioned Architectures

To better understand the increase of complexity we will use as an example the evolution of embedded systems from "federated" to "integrated" and "partitioned" architectures. Many embedded applications are implemented as distributed systems, due to various constraints, e.g, modularity or safety. In a distributed system, the hardware compo-

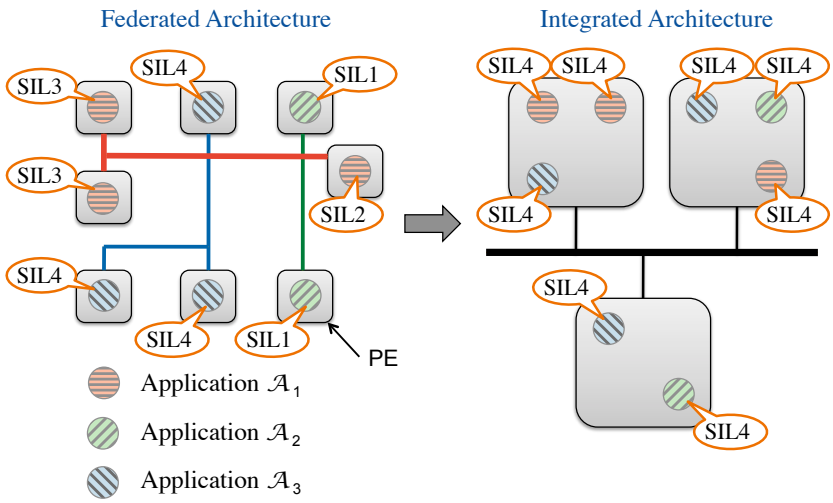


Figure 1.1: Federated versus integrated architectures

nents, called *nodes* or PEs, are interconnected in a network. If all the nodes are of the same type, the system is *homogeneous*. Otherwise, the system is called *heterogeneous*.

In "federated architectures", each node implements at most one function, and the applications are very loosely coupled [141]. On the left side of Fig. 1.1 we have three applications of different criticality levels implemented on a distributed architecture, in a federated manner. Each PE implements one task. If an application is composed of several tasks, the PEs implementing tasks of the same application are connected using a point-to-point communication protocol. Such an architecture facilitates fault containment: unless there is functionality dependency between two applications, a faulty task in an application will not affect tasks in other applications. This approach allows the system engineers to integrate into their system nodes from different suppliers. As more functionality was implemented using embedded systems, the number of such nodes increased (e.g., over 100 in a premium-class car), together with the associated wiring, cost and SWaP, i.e., size, weight and power.

An approach to counter this increase is implementing the system using an "integrated architecture", by integrating several functions onto the same node [103]. Fig. 1.1 presents this progression from federated to integrated architectures. Implementing the applications using an integrated architecture (right side of Fig. 1.1), results in several tasks of possibly different applications implemented on the same PE, thus reducing the SWaP of the system. A disadvantage of integrated architectures is the complexity increase of the system: new possible interactions will arise among applications of different criticality implemented on the same platform. In this case, the certification

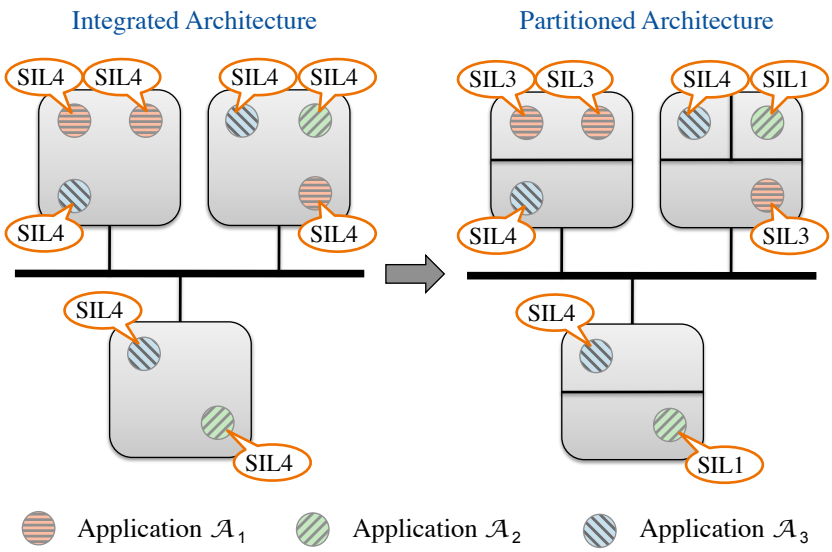


Figure 1.2: Integrated versus partitioned architectures

standards offer two solutions. One solution is to certify all the software in the platform to the highest level, dramatically increasing the certification costs. This is the case in Fig. 1.1, where all the tasks have to be certified to the highest SIL among them, which is SIL4. The other solution is to demonstrate that applications of different criticality levels are separated, so they cannot influence each other.

To provide separation, embedded system engineers are relying on "partitioned architectures", which provide partitioning mechanisms at the platform level. Partitioning provides "fault containment equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines" [141]. Fig. 1.2 compares integrated architectures (left side) with partitioned architectures (right side), in the context of the example in Fig. 1.1. In case of partitioned architectures, the separation between applications in Fig. 1.2 is symbolized using the black lines in the PEs. Compared to the integrated approach, the partitioned architecture offers enough separation between tasks of different SILs, so the tasks can be developed and certified according to their initial SIL (shown in the left side of Fig. 1.1), reducing the costs.

Partitioning has two dimensions: *spatial* and *temporal*. Spatial partitioning ensures that an application in a partition will not interfere with the code and data of another partition [141]. Temporal partitioning ensures that an application's scheduled access to shared resources (e.g, processor or communication bus), cannot be affected by an application in another partition [141].

For example, in the avionics area, the partitioned architecture is called "Integrated Modular Avionics" (IMA) [3], and the platform-level separation mechanisms are provided by implementations of the ARINC 653 standard [17]. With temporal partitioning, each application is allowed to run only within predefined time slots or *partition slices*, allocated on each processor. The sequence of partition slices for all the applications on a processor are grouped within a Major Frame, which is repeated periodically. Similar platform-level separation mechanisms are available in other industries [74, 110, 135]. Each partition can have its own scheduling policy. Depending on the partitioned real-time operating system (RTOS) (i.e., the RTOS that implements partitioning mechanisms), partitions can also run different host operating systems.

Previously, we described partitioning at the processor level. But in a distributed system, the communication protocol must also have mechanisms to enforce partitioning at the bus level. For example, space partitioning is attained in SAFEbus [92] by mapping the messages to unique locations in the inter-module memory, protected by a memory-mapping hardware in the host, and temporal partitioning is achieved in TTP [102] by enforcing a Time-Division Multiple Access scheme.

In this thesis, we assume that the bus-level partitioning mechanisms are provided by a communication protocol such as TTEthernet [28], which provides both spatial and temporal partitioning, and can handle both static and dynamic communication. TTEthernet is a deterministic, synchronized and congestion-free network based on the IEEE 802.3 Ethernet standard and compliant with the ARINC 664p7. TTEthernet is suitable for automotive [156], avionics [161] and space applications [79].

We present in Section 2.2 more details on partitioning at the processor level, and in Section 2.2.2 we talk about partitioning at the communication level, with regards to the TTEthernet protocol. We describe how the TTEthernet protocol works in Section 2.2.3.

1.3 Embedded Systems Design

Embedded systems have to implement the desired functionality and also meet several constraints, or "design metrics". A *design metric* is "a measurable feature of a system's implementation" [173]. Next, we summarize some of the common design metrics. A more detailed description can be found in [173, 86].

We split these design metrics into two categories, depending on the main impact: the "project-related" design metrics impact the management and finances of the project, while "product-related" design metrics impact physical and qualitative characteristics of the product. Examples of product-related design metrics are:

- size, weight and power consumption, collectively referred to as *SWaP*.
- performance: the amount of time necessary to execute the tasks. Most common measures of performance are *latency* (the elapsed time between the start and end of execution) and *throughput* (number of tasks executed per time unit) [173].
- predictability: is a system property that guarantees that the "requirements are met subject to any assumptions made" [155]. Thus, when referring to task scheduling, a real-time system is predictable if one can determine the "evolution of the tasks and guarantee in advance that all critical timing constraints will be met" [56].
- worst-case execution time (WCET) of a given task is the guaranteed "upper bound for the time between the task activation and task termination" [102] and is used in schedulability analyses. There are several tools that determine a task's WCET, e.g., aiT [24, 77]. Wilhelm et al. [181] present an overview of methods and tools to compute the WCET. Determining the WCET in modern processors becomes increasingly difficult, as processors include more features that speed up the common case, but are harder to model. Schoeberl [150] lists the processor design issues that make WCET analysis difficult, and proposes a time-predictable architecture that supports WCET analysis to reduce analysis pessimism.
- worst-case response time (WCRT): of a given task is the longest possible response time, i.e., the time between the task's release time and the task's termination. In the case of offline scheduling, the WCRT of a task is determined by checking the produced schedule. In the case of preemptive scheduling, the WCRT of a task is determined by using scheduling policy-specific response-time analyses.
- dependability: integrates attributes like reliability (continuity of correct service), availability (readiness for correct service), safety (absence of catastrophic consequences on the users and the environment), integrity (absence of improper system state alterations), confidentiality (absence of unauthorized disclosure of information) and maintainability (ability to undergo repairs and modifications) [32].

Examples of project-related design metrics are:

- cost: which can be broken down into *unit cost* and *nonrecurring engineering cost* (NRE). The unit cost is the cost to produce one copy of the product, while NRE is the one-time engineering cost to design and develop the system [173].
- time-to-market: the necessary time to design and produce the system to the point it can be sold. With increasing competition and shortening market windows[1],

[1] The time during which the product would have the highest sales [173].

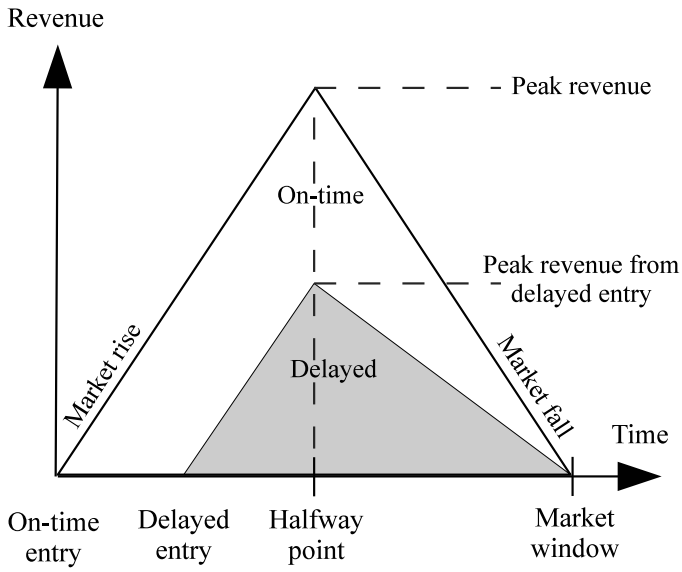


Figure 1.3: Revenue impact of delayed entry

this constraint became one of the most demanding. Fig. 1.3, adapted from [173], presents the revenue impact of delayed market entry. This figure uses a simplified revenue model, where the revenue of a product is represented by the triangle area. Entering the market late will reduce the revenues, and will potentially have a bigger impact than development cost overruns [173].

The design metrics are competing against each other: most often, optimizing one metric will have negative impact on the others. For example, two conflicting metrics are performance and power consumption. Increasing the performance of the system will increase the power consumption of the system, reducing the battery life. Thus, optimizing the system to meet all the design metrics adds to the difficulty of designing embedded systems.

Successfully designing an embedded system within the given time frame is increasingly difficult, taking into account the increasing complexity and the competing design metrics. The 2013 Embedded Market Survey [18] revealed that in 2013, 57% of projects finished late or were cancelled. A survey [52] of the automotive industry states that some of the top system design challenges are integration problems that delay time-to-market (46%) and the difficulty of trade-off decisions about system architecture (29%).

In this context, using good practices, processes and methods to engineer systems becomes critical. *Systems engineering* is an "interdisciplinary approach and means to

Systems Engineering stages	Concept development			Engineering development			Post development	
Systems Engineering phases	Needs analysis	Concept exploration	Concept definition	Advanced development	Engineering design	Integration & evaluation	Production	Operation & support

Figure 1.4: A systems engineering life cycle model (from [105])

enable the realization of successful systems" [15]. Another source [54] defines systems engineering as the "discipline that develops, matches, and trades off requirements, functions, and alternate system resources to achieve a cost-effective, life-cycle-balanced product based upon the needs of the stakeholders". Systems engineering, as an approach, considers the whole life cycle of the product, i.e., all the phases of a system, from concept and development to disposal [105].

There are several life cycle models, e.g., ISO/IEC 15288 [97], but in the following we will focus on a derived systems engineering life cycle model developed by Kossiakoff and Sweet [105]. This model is presented in Fig. 1.4, and divides the life cycle intro three stages: a *concept development* stage, an *engineering development* stage, and a *post development* stage, which are further divided into 8 phases. The concept development stage defines the system concept. The engineering development stage translates the system concept into a system design that meets the imposed constraints. The post development stage includes the production, operation, support and decommissioning of the system. More details on this model can be found in [105].

The methods presented in this thesis focus on the early life cycle phases, where the impact of design decisions is greatest. This impact is shown by comparing, for each life cycle stage, the actual expenditures (cost incurred) with the planned costs based on design and engineering decisions (cost committed), see Fig. 1.5 (adapted from [54]). Although at the end of the engineering development stage, when the final system design is produced, only about 20% of the cost is incurred, but 80% of the cost is already committed [54].

Given that 80% of the budget is committed at the end of the engineering stage, what is the cost of correcting design issues? During system development, the system is validated, to ensure that it meets the needs of the customer. Design changes require the system to be validated again. Let us discuss the cost of correcting design issues by considering the impact of revalidating the system. For example, the average time-to-market for a power train unit is 24 months, while the validation of such a system takes 5 months [144]. In the case of the instrument cluster, the validation time is 2 months, representing 16% of the time-to-market (12 months). Thus, bad design decisions will also increase the time-to-market of the system. We have discussed earlier in this section the negative impact on revenue of delayed entry in the market (see Fig. 1.3).

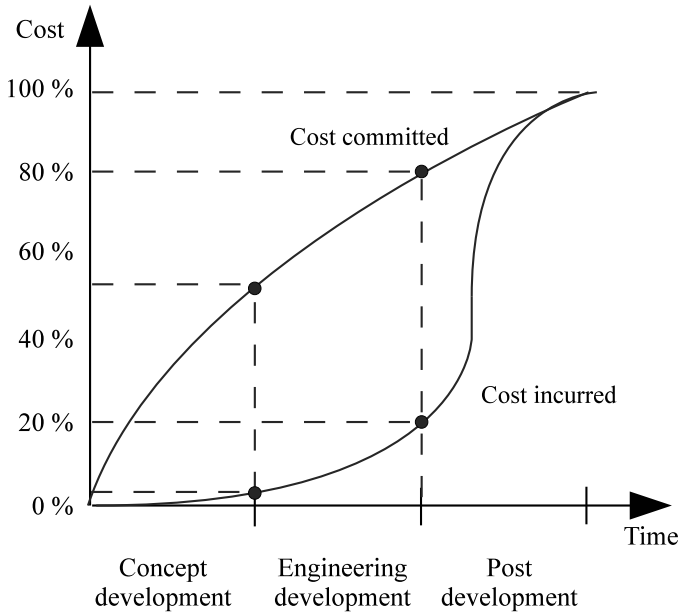


Figure 1.5: Cost committed versus cost incurred during the system life cycle

Thus, more design effort is needed in the early life cycle phases, since the decisions taken during these phases have a high impact and commit a lot of costs. In this thesis, we provide methods and tools to be used during the early design stages, to help the system engineers take better decisions, and thus reduce the costs. The solutions we propose are useable at the processor level and at the communication network level. Section 1.5 summarizes the contributions of this thesis.

1.4 Design Space Exploration

During the concept development stage (see Fig. 1.4), the engineers examine potential system concepts and design alternatives, and select the preferred one [54]. Design engineers select the preferred alternative after performing a *trade-off analysis*, which evaluates several design alternatives in terms of design metrics. The creation and evaluation of alternatives is called "design space exploration" (DSE).

Fig. 1.6 presents the DSE technique. DSE starts from models of the application functionality and system platform. There are many ways to model an application functionality (see [84] for a description of the common modeling formalisms used in embedded systems design). In this thesis we use task graphs [138] (also referred to as "dataflow

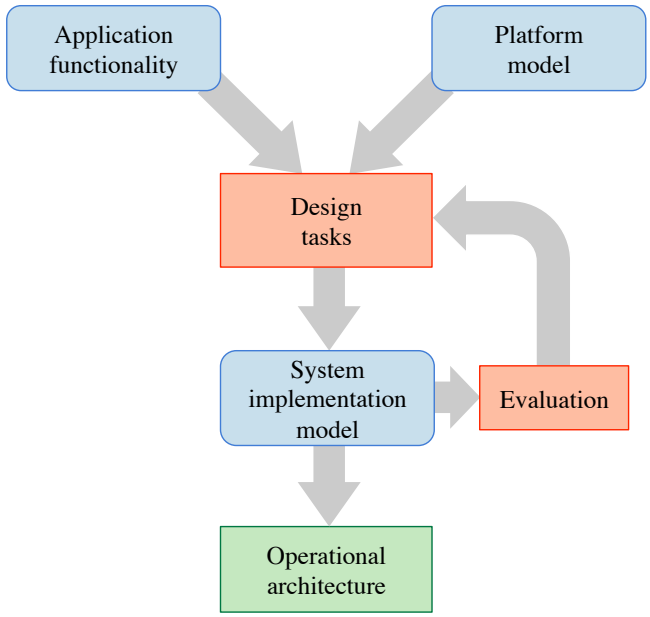


Figure 1.6: Design Space Exploration

process networks" [108]) and the exact model is presented in Section 2.1. For the platform, we consider heterogeneous distributed platforms, consisting of processing elements interconnected using the TTEthernet [28] communication protocol. We assume the platform implements partitioning mechanisms similar to Integrated Modular Avionics (IMA), so that each application can execute only in its own partition. Moreover, partitions can implement different scheduling policies. In this thesis we considered several scheduling policies such as non-preemptive static cyclic scheduling or preemptive fixed-priority scheduling. The details of our platform model are presented in Section 2.2.

DSE is performed in the "Design Tasks" box. These tasks are done during the early lifecycle phases. DSE could be done manually for small designs, but in practice it is typically supported by tools, which perform an automatic DSE. These tools use optimization techniques to search for solutions which optimize a set of design criteria called objectives. There are many optimization approaches. For example, optimizations based on mathematical methods, such as integer linear programming [51], always find the optimal solutions, but are impractical for large problems due to prohibitive search times. In this thesis, we have used metaheuristics such as Tabu Search for automatic DSE. Tabu Search does not guarantee finding the optimal solution, but can often produce good quality solutions that satisfy the requirements, in a reasonable time, as the experimental results will show.

We perform several design tasks, such as mapping of task to processing elements, partition optimization, routing of messages in the communication network and scheduling of tasks. Section 1.5 presents the full list of design tasks we address in this thesis, and they are described in detail in Chapter 3 (at the processor level) and Chapter 4 (at the communication network level).

The solutions are evaluated using objectives. There are multi-objective optimizations such as Evolutionary Algorithms [60], but we have used Tabu Search, where the multiple objectives have been collapsed into a single objective using the weighted sum method [66]. The evaluation of each implementation alternative can be done either analytically or via simulation. Both methods have advantages and disadvantages. Simulations are better suited "to investigate dynamic and sporadic, unforeseeable effects in the system", but they require a working model [86] and in the context of real-time systems it cannot provide guarantees that the timing constraints are satisfied. On the other hand, an analytical approach is usually faster. In this thesis, we used analytical evaluations.

Once the engineer is satisfied with the solution, the DSE can stop. The obtained solution defines "operational architecture". However, if no solution is found, an option is to go back and change the hardware architecture. We discuss DSE-based architecture selection in Section 5.4.2.1, where we show how to perform topology selection for a TTEthernet network.

1.5 Research Contributions

In this thesis, we focus on mixed-criticality applications (both in the safety and time domains), implemented using heterogeneous distributed platforms, consisting of several processing elements interconnected using the TTEthernet [28] protocol. We assume that the platform provides both spatial and temporal partitioning, thus enforcing enough separation for the mixed-criticality applications. We consider that each application is composed of tasks that communicate using messages.

At the processor level, we consider partitioned systems similar to IMA [3, 141]. Similar platform-level separation mechanisms are available in other industries [74, 110, 135]. With temporal partitioning, each application is allowed to run only within predefined time slots or *partition slices*, allocated to each processor. The sequence of partition slices for all the applications on a processor are grouped within a Major Frame, which is repeated periodically. Thus, each partition can have its own scheduling policy. We have considered two scheduling policies using non-preemptive static cyclic scheduling or preemptive fixed-priority scheduling.

At the communication level, messages are transmitted via frames using the TTEthernet protocol. The TTEthernet protocol has three classes of messages: static time-triggered (TT) traffic and dynamic traffic, which is further subdivided into Rate Constrained (RC) traffic that has bounded end-to-end latencies, and Best-Effort (BE) traffic, for which no timing guarantees are provided. TT messages are transmitted based on static schedule tables and have the highest priority. RC messages are transmitted if there are no TT messages, and BE traffic has the lowest priority. TTEthernet offers spatial separation for mixed-criticality messages through the concept of virtual links, and temporal separation, enforced through schedule tables for TT messages and bandwidth allocation for RC messages.

All of the related work (see Section 1.7) assumes that tasks of different criticality share the same processor with little or no separation (i.e., there is no spatial-partitioning). Current certification practice requires separation, and can only remove such a requirement if the two tasks are at the same criticality level. This practically means that all the tasks would have to be developed and certified at the highest criticality level, which is not feasible, due to the prohibitive development and certification costs. Such research assumes that in the future the certification practice will change. For example, a vision of "just-in-time certification" is proposed by [142]. However, the current standards-based certification practice is unlikely to change in the near future. For this reason, our work allows tasks with different criticality levels to share a partition only if the lower-criticality tasks are elevated at the higher-criticality level. We describe our research for the processor level in Chapter 3.

At the processor level, the main contributions of this thesis are:

- Considering a given architecture, where PEs are connected with a simple statically scheduled bus, and a fixed mapping of tasks to PEs, we have proposed an optimization method to determine the sequence and size of the partition slices within the Major Frame on each PE such that all applications are schedulable [166]. For this problem, we have developed a metaheuristic solution based on Simulated Annealing. We have shown that only by optimizing the sequence and length of the time partitions we are able to obtain schedulable implementations.

- Partitioning introduces overheads because it constrains the way tasks can use the PEs, leading to unused slack inside certain partition slices. We have shown that by simultaneously optimizing the mapping and partitioning, these overheads are minimized, increasing thus the chance to find schedulable implementations. We have proposed an optimization strategy based on Tabu Search to solve this design problem. Given a set of applications and the physical architecture, our optimization determines the mapping of tasks to PEs, the assignment of tasks to partitions, and the sequence and size of the partition slices within each Major Frame on each PE, such that all applications are schedulable [167].

- In situations where simultaneously optimizing the mapping and partitioning does not result in schedulable solutions, the system engineers can upgrade the architecture, increasing the unit cost of the system. We have shown that an alternative is to elevate tasks to higher SILs to allow partition sharing. Thus, the engineer can keep the same architecture, at the expense of increasing the NRE costs. We have extended the original task graph model to allow applications to contain tasks of different SILs. Moreover, we have proposed a *development cost model* to capture the development costs associated to a given SIL, and a *separation requirements graph*, specifying which tasks are not allowed to share the same partition. This design problem can be formulated as: given the set of applications (including the SIL-dependent development costs and the separation requirements graph) and the architecture, we are interested to determine the mapping of tasks to processors, the assignment of tasks to partitions, the partitioning on each PE, such that all applications are schedulable, and the development costs are minimized. We have proposed a Tabu Search-based approach to solve this optimization problem [165].

- Certification standards allow a task of higher SIL to be decomposed as several redundant tasks of lower SILs, i.e., "task decomposition". We have shown how using task decomposition can reduce the NRE costs and can facilitate partition sharing. The disadvantage is the introduction of more tasks, potentially impairing schedulability. The design problem in this case is determining a decomposition of tasks such that the timing requirements are satisfied and the development costs are minimized. We proposed a Tabu Search-based optimization strategy to solve this problem [168].

- The previously mentioned optimization strategies target hard real-time applications. We have shown in Chapter 5, using a real-life case study [163], how to apply these optimizations to systems also containing soft real-time applications. We have proposed a new cost function, capturing the quality of service for the soft real-time applications.

- We proposed a response time analysis for tasks scheduled using fixed-priority preemptive scheduling policy in partitioned architectures [114]. Our proposed analysis is based on the work presented in [136].

We have proposed design strategies for the optimization of TT schedules. Compared to the related work (see Section 1.7), our optimizations do not restrict the space inserted into the TT schedules to evenly-spaced periodic slots. Moreover, our strategies are able to take into account the RC end-to-end delays during the design space exploration, and not only as a post-synthesis check. None of the existing related work addresses packing/fragmenting and routing for TTEthernet (see Section 1.7). We describe our research for the network communication level in Chapter 4.

At the communication network level, the main contributions of this thesis are:

- Given the set of mixed-criticality messages in the system and the topology of the virtual links on which the messages are transmitted, we have proposed an optimization to synthesize offline the static schedules for the TT messages, such that the deadlines for the TT and RC messages are satisfied, and the end-to-end delay of the RC traffic is minimized. We have implemented a Tabu Search meta-heuristic to solve this optimization problem. We have shown that by considering the RC traffic when scheduling the TT frames, the impact of the TT schedule on the latency of the RC frames can be greatly reduced [169].
- In a similar context, we have shown that by carefully deciding the fragmenting and packing of messages to frames, we can improve the schedulability of messages. We have also shown the importance of optimizing the routing of virtual links. We proposed a design optimization strategy that, for a given network topology and a given set of TT and RC messages, optimizes the fragmenting and packing of messages to frames, the routing of virtual links, the bandwidth for each RC virtual link and the TT schedules, such that the deadlines for the TT and RC frames are satisfied, and the worst-case end-to-end delay of RC frames is minimized [170].
- We have proposed in Section 5.4.2.1 an architecture selection method to reduce the costs of the system. Starting from an initial topology, our approach performs topology selection by iteratively reducing the number of physical links and network switches, and searching for solutions that satisfy the constraints of the RC and TT frames.
- Although the main focus of this thesis, at the communication network level, is on TT and RC frames, in Section 5.4.2.2 we have extended our optimizations to take into account BE traffic as well. We have shown that our proposed strategy is flexible, and with minimal changes, it can tackle different optimization problems.

1.6 Thesis Overview

The thesis is structured into six chapters as follows:

Chapter 2 introduces the application and architecture platform models used in this thesis. This chapter also presents IMA and shows how the TTEthernet protocol works.

Chapter 3 presents the optimization strategies at the processor-level. The chapter starts presenting several motivational examples that explain the design problems

we focus on. The first strategy is a Simulated Annealing-based metaheuristic that, for a given set of applications and a fixed mapping of tasks to PEs, optimizes the partitioning (i.e., the sequence and size of the partition slices) on each PE, such that all applications are schedulable. The second strategy, implemented using Tabu Search, takes into account also the issue of development and certification costs. Given the set of applications, the library of possible SIL decompositions, the separation requirements between tasks and the architecture platform, the second strategy optimizes the SIL decomposition, the mapping, partitioning and assignment of tasks to partitions, such that all applications meet their deadline and the development costs are minimized. The chapter also describes our proposed response-time analysis for tasks scheduled using the fixed-priority preemptive scheduling policy in partitioned systems. We evaluated these strategies using several synthetic and real-life benchmarks.

Chapter 4 describes our optimization strategy at the communication-level, considering the TTEthernet protocol. Given the topology of the TTEthernet cluster and the set of TT and RC messages, the strategy optimizes the fragmenting and packing of messages to frames, the assignment of frames to virtual links, the routing of virtual links, the bandwidth for each RC virtual link and the TT schedules, such that the deadlines for the TT and RC frames are satisfied. This optimization strategy is implemented as a Tabu Search metaheuristic. We have evaluated the proposed optimization using several synthetic and real-life benchmarks. The evaluation confirmed that in many cases, obtaining schedulable solutions is not possible only by performing TT schedule optimization, but requires performing all the optimizations simultaneously.

Chapter 5 discusses the issues related to implementing mixed-criticality applications on partitioned architectures in the context of a given application area, in this case the aerospace area. If in the previous chapters we focused on hard real-time applications, in this chapter we show that handling also soft real-time and best-effort applications is important, using several realistic applications. The applications at the processor-level consist of the Mars Pathfinder lander software (see Section 5.2.1) and the controller for a spectrometer (see Section 5.2.2). At the processor level, we extended the optimization strategy described in Section 3.2.2 to take into consideration soft real-time constraints. At the communication network level, the realistic application is the Orion Crew Exploration Vehicle, described in Section 5.3. We extend the optimization strategy presented in Chapter 4 to take into account BE traffic. Moreover, we also propose a topology selection method, extending the strategy from Chapter 4, to reduce the cost of the system.

Chapter 6 concludes this thesis and describes future work ideas.

1.7 Related Work

Processor-Level. There is a large amount of research on hard real-time systems [102, 56], including task mapping to heterogeneous architectures [53]. There are two approaches to handling tasks, depending on the triggering mechanism to initiate a processing or communication activity. In the *Event-Triggered* (ET) approach, activities are initiated whenever a particular event is noted. In the *Time-Triggered* (TT) approach, activities are initiated at predetermined points in time. Researchers have addressed systems with mixed *time*-criticality requirements, showing how Time Triggered (TT)/Event Triggered (ET) tasks or hard/soft real-time tasks can be integrated onto the same platform. There has been a long debate in the real-time and embedded systems communities concerning the advantages of each approach [29, 102, 186]. However, there is little research work on the integration of mixed *safety*-criticality applications onto the same platform.

In the context of mixed TT/ET systems, Pop et al. [136] have shown how the static schedules can be optimized such that both the TT applications (scheduled using non-preemptive static-cyclic scheduling) and the ET applications (scheduled using fixed-priority preemptive scheduling) are schedulable. Their approach could be extended to constrain the TT schedules to follow a given partitioning. They have later addressed the problem of mapping and partitioning, but in their context partitioning means deciding which tasks should be TT and which ET [129]. While in [129, 136] TT and ET tasks share the same processor, the work in [134] considers that TT and ET tasks are implemented on different *clusters*. In this context, partitioning means deciding in which cluster (TT or ET) to place a task.

Researchers have shown how to integrate mixed hard/soft real-time tasks onto the same platform. The order of tasks is decided by quasi-static scheduling in [62] (several schedules are determined offline, and are activated online depending on when tasks finish executing), such that the hard tasks meet their deadlines and the total "utility" of soft tasks is maximized. This work has been extended by Izosimov et al. [98] to handle transient faults, by switching online to backup recovery schedules. Soft real-time tasks can be integrated in fixed-priority preemptive scheduling using the Constant Bandwidth Server (CBS) [23], where the server is a hard task providing a desired level of service to soft tasks. Thus, the CBS-servers provide a time-partitioning between hard and soft tasks. The optimization of CBS-server capacity in the context of mixed hard and soft real-time tasks has been addressed by Saraswat et al. [146], such that the hard tasks are schedulable and the quality of service for the soft tasks is maximized.

Researchers also addressed the problem of mixed-criticality systems. A recent review of the research in the area of mixed-criticality systems was written by Burns and Davis [55].

Lee et al. [109] consider an IMA-based system where all tasks are scheduled using FPS. The time-partition optimization problem is formulated as a static cyclic scheduling problem, where the partitions are statically scheduled such that the FPS tasks are schedulable. A similar approach to IMA is used in the DEOS operating system [46], with the difference that FPS is used for scheduling both the partitions (which are normally scheduled using SCS) and the tasks. Binns [46] has proposed several slack-stealing approaches, where the unused time in one partition is given to the other partitions, thus the partitions are implicitly adjusted online.

There are several works where mixed-criticality tasks are addressed, mostly targeting uniprocessor systems. Vestal [175] was the first to extend the task model to include criticality-level dependent Worst-Case Execution Times (WCETs). He proposes two fixed-priority preemptive scheduling algorithms based on period transformation [154] and on Audsley's algorithm [31], respectively. Dorin et al. [72] prove that Vestal's algorithm based on Audsley's priority assignment algorithm is optimal. Baruah and Vestal [41] extend the work from [175] and propose for sporadic tasks sets a hybrid-priority scheduling policy [38], which includes features of the Earliest Deadline First (EDF) policy as well. Baruah et al. [40] propose a task model that can capture mixed-criticality functions, together with an associated schedulability analysis.

Several papers [111, 39, 36] base their work on the assumption that the Certification Authorities (CAs) consider a more pessimistic WCET for the tasks than the system engineer, leading to inefficient usage of computing resources. Thus, for each task they take into account two WCETs: a HI WCET, pessimistic, considered by the CA, and a less pessimistic LO WCET expected by the system engineer. The CAs require, indeed, the use of guaranteed upper bounds, i.e., WCET, for the tasks when designing the system. Li and Baruah [111] propose an algorithm for on-line priority-based scheduling of mixed-criticality sporadic tasks on uniprocessors. If during a busy period, a high criticality task exhibits HI WCET behavior, that is, its execution time is larger than the LO WCET assumed by the engineer, the low criticality tasks are discarded, in order to ensure the necessary CPU time to all the high criticality tasks. Baruah et al. [36] introduce a novel scheduling algorithm using fixed-priority scheduling on uniprocessor mixed-criticality systems which takes into account the criticality level of each task, evaluate three possible priority assignment schemes, and propose an associated response time analysis.

Baruah and Fohler [39] offer a solution using Time-Triggered scheduling to the problem in [40, 111]. They propose a "mode change" approach: using the algorithm from [111], they compute offline two schedules, a "certification mode" schedule considering the HI WCET behavior, and a "engineer mode" schedule considering the LO WCET behavior. In case a task overruns its LO WCET, a mode change is triggered, and the system runs using the "certification mode" schedule.

Similar to the work cited before, de Niz et al. [65] also base their work on two operation modes, which they refer to as "normal mode" and "critical mode". In [65], they discuss the issue of "criticality inversion", similar to the classical priority inversion problem, and propose a "zero-slack scheduling" policy for such a context. This scheduling algorithm works on top of preemptive priority-based schedulers, uses both the priority and criticality of a task, and prevents interference from lower-criticality to higher-criticality. Lakshmanan, de Niz and Rajkumar [107] extend this work with two protocols: the Priority-and-Criticality Inheritance Protocol (PCIP) and the Priority-and-Criticality Ceiling Protocol (PCCP). With PCIP, a task holding a lock to a shared resource inherits the highest criticality and the highest priority of the tasks waiting for the resource. The PCCP assigns to each lock (resource) the highest priority and the highest criticality of all the tasks, and thus prevents deadlocks.

Mollison et al. [118] propose a scheduling policy for mixed-criticality tasks on multicore systems. In their approach, the high criticality tasks are considered *slack generators*, as the WCET predictions are deemed overly pessimistic, and the authors assume that these tasks will "use only a small fraction of the execution time budgeted for them". Employing slack shifting, this approach considers the low criticality tasks *slack consumers* and are allowed to execute during the slack if it will not have a negative impact on the timing of the high criticality tasks. Herman et al. [89] experimentally evaluated [118] to asses OS-related implementation issues, e.g., the impact of scheduling overhead, and to evaluate "the robustness of mixed-criticality analysis" presented in [118]. Li and Baruah [112] propose a scheduling algorithm for mixed-criticality sporadic tasks implemented on homogeneous multiprocessor platforms, were task migration is permitted. Their algorithm is based on the EDF-VD [42] uniprocessor scheduling algorithm and fpEDF [43] global scheduling algorithm.

Kelly et al. [101] use a SIL-dependent WCET for the tasks, and propose bin-packing algorithm for mapping of tasks to processor. Furthermore, they focus on fixed-priority preemptive tasks, and compare the rate monotonic priority assignment algorithm [113] with Audsley's algorithm [31], concluding that "in general Audsley's algorithm offers a significant advantage over RM assignment".

Goswami et al. [85] present an Integer Linear Programming (ILP) algorithm for the schedule synthesis of mixed-criticality tasks on TT platforms. They consider two criticality types, safety-critical (for control applications) and time-critical, and synthesize the schedules for the tasks and the bus to optimize control performance, while meeting the deadlines for the time-critical applications. Schneider et al. [148] address a similar problem, in the context of fixed priority scheduling. They propose an offline priority assignment algorithm to optimize the quality of control for the control applications, while satisfying the timing constraints for the time-critical applications.

Communication Network. The ET/TT duality is also reflected at the level of the communication infrastructure, where communication activities can be triggered either dynamically, in response to an event, as with the Controller Area Network (CAN) bus [4], or statically, at predetermined moments in time, as in the case of Time-Division Multiple Access (TDMA) protocols such as the Time-Triggered Protocol (TTP) [102]. The trend is towards bus protocols that support both static and dynamic communication [10, 28].

The problem of the optimization of time-partitions has been addressed at the bus level. Researchers have shown how a Time-Division Multiple Access bus such as the TTP [130] and a mixed TT/ET bus such as FlexRay [137] can be optimized to decrease the end-to-end delays. This optimization problem was also addressed by Schoeberl et al. [151] in the context of a TDMA network-on-chip designed at the Technical University of Denmark. The optimization implies deciding on the sequence and length of the communication slots.

Due to an increase in the complexity of control applications and embedded systems, several companies are offering real-time communication solutions based on Ethernet. Ethernet, with its low costs and high speeds (100 Mbps and 1 Gbps, and soon 10 Gbps), is known to be unsuitable for real-time and safety-critical applications [7]. For example, in half-duplex implementations, frame collision is unavoidable, leading to unbounded transmission times. Decotignie [68] presents the requirements for a real-time network and how Ethernet can be improved to comply with these requirements. There are several academic and industrial Ethernet-based solutions: ARINC 664p7, TTEthernet [28], EtherCAT [19], IEEE Audio Video Bridging (AVB), etc. Audio Video Bridging is a collection of technical specifications [12, 9, 8, 13] that target synchronized communication with low jitter and low latency on Ethernet networks. The work in [147, 64, 99] describe and compare the previously mentioned Ethernet-based communication protocols.

For full-duplex switched Ethernet networks with priority operations, Schneider et al. [149] have proposed a compositional timing analysis based on real-time calculus [58]. For ARINC 664p7 systems, researchers [120] have proposed a new real-time switching algorithm that guarantees an upper bound on the switching period. Having such an upper bound simplifies the worst-case delay analysis. For TTEthernet, Steiner [159] proposes an approach for the synthesis of static TT schedules, where he ignored the RC traffic and used a Satisfiability Modulo Theory (SMT)-solver to find a solution which satisfies an imposed set of constraints. The same author has proposed an SMT-solver approach to introduce periodic evenly-spaced slots into the static schedules to help reduce RC delays in [157]. Suethanuwong [162] proposes a scheduling approach of the TT traffic, ignoring RC traffic, that introduces equally distributed available time slots for BE traffic.

Researchers have also addressed the issue of frame packing [133, 143]. Frame packing is one of the fundamental features for some communication protocols. For example, EtherCAT [19] is a master/slave protocol, where the master packs several "datagrams" (i.e., messages) into a single frame, regardless of the destination, and sends the frame to all the slaves. Recent work has also addressed the ARINC 664p7 protocol. Ayed et al. [33] propose a packing strategy for multi-cluster networks, where the critical avionics subsystems are based on CAN buses, and are interconnected via ARINC 664p7. This strategy, meant to minimize the CAN bandwidth through the ARINC 664p7 network, performs packing at the CAN-ARINC 664p7 gateway based on a timer. Messages are not packed based on destinations, but on availability. As a consequence, all the messages packed in a frame are delivered to all the possible destinations. Also for ARINC 664p7, Al Sheikh et al. [25] propose a packing strategy for messages with the same source and destinations, with the goal of minimizing the reserved bandwidth. Mikolasek et al. [116] proposed a segmentation algorithm for the standard Ethernet messages in Time-Triggered Ethernet [104], an academic Ethernet-based protocol that supports standard Ethernet traffic and time-triggered messages. This algorithm will fragment the standard Ethernet messages into smaller frames that can be transmitted between two time-triggered frames, reducing transmission preemption and increasing throughput. Routing in networks is a very well researched topic. Researchers have also addressed routing in safety-critical systems [90, 127]. For ARINC 664p7, Al Sheikh et al. [25] propose a load-balancing routing strategy.

CHAPTER 2

System Model

In this chapter we present the system models we used in the thesis. A *model* is a "reduced representation" [102] of an object, focusing on key aspects of the object, while ignoring all the others. Models are obtained through a process of abstraction, and help us deal with complexity. The intended use of the model dictates what are the key aspects of the modeled object.

First, we present the application and architecture models at the processor level, and we shortly talk about the certification process and about partitioning (Section 2.1 and Section 2.2.1). Second, we present the models at the communication level (see Section 2.2.2). As previously mentioned, we assume the communication is done using TTEthernet. Section 2.2.3 presents how the TTEthernet protocol works.

2.1 Application Model

The set of all applications in the system is denoted with Γ. We model an application as a directed, acyclic graph $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \Gamma$. The graph is polar, which means that there is a *source node*, i.e., a node that has no predecessors, and a *sink node*, i.e., a node that has no successors. Each node $\tau_j \in \mathcal{V}_i$ represents one task. The mapping is denoted by the function $M : \mathcal{V}_i \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of processing elements (PEs) in the

architecture. For each task τ_i we know the worst-case execution time (WCET) $C_i^{N_j}$ on each processing element N_j where τ_i is considered for mapping.

An edge $e_{jk} \in \mathcal{E}_i$ from τ_j to τ_k indicates that the output of τ_j is the input of τ_k. A task becomes ready after all its inputs have arrived, and it issues its outputs when it terminates. Communication between tasks mapped to different PEs is performed by message passing over the bus. We assume that the message sizes $m_i.size$ of each message m_i are known. Section 2.2.2.1 describes how we model applications at the communication level.

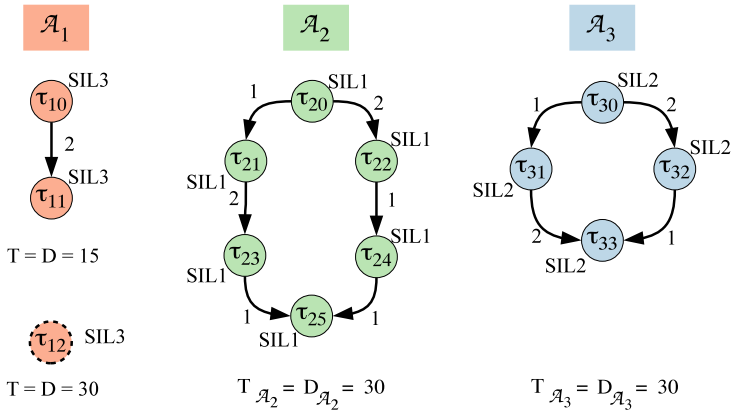
The applications are scheduled using non-preemptive static cyclic scheduling (SCS) or preemptive fixed-priority scheduling (FPS). For the SCS applications, a deadline $D_{\mathcal{G}_i} \leq T_{\mathcal{G}_i}$, where $T_{\mathcal{G}_i}$ is the period of $\mathcal{G}_i$, is imposed on each application graph $\mathcal{G}_i$. Regarding FPS tasks, we use the model from [136], which considers arbitrary deadlines and release times, and also takes into account dependencies. The tasks' priorities are specified by the engineer. Moreover, for each task τ_i, we assume that we know its period T_i and deadline D_i.

An example of mixed-criticality system composed of three SCS applications is presented in Fig. 2.1a. The periods and deadlines are presented under the application graphs. The WCETs of tasks are given in Fig. 2.1b for two PEs, N_1 and N_2. An "x" in the table means that the task is not considered for mapping on the respective PE. The size of the messages is depicted on the graph edges. Fig. 2.1c captures the development costs for each task. These costs are discussed in Section 2.1.3.

If dependent tasks are of different periods, they are combined into a merged graph capturing all activations for the hyper-period (LCM of all periods). Release times of some tasks as well as multiple deadlines can be easily modeled by inserting dummy nodes between certain tasks and the source or the sink node respectively. These dummy nodes represent tasks with a certain execution time but which are not allocated to any PE. Thus, by meeting the global deadline, all the local deadlines and release times are guaranteed [132].

2.1.1 Safety Integrity Levels

As mentioned, a *safety-critical* system should not endanger human life or the environment. A *hazard* is a situation in which there is actual or potential danger to people or to the environment. *Risk* is a combination of the frequency or probability of a specified hazardous event, and its consequence. If, after performing an initial hazard and risk analysis, a system is deemed safety-related, it has to be certified [160]. Certification is a "conformity of assessment" performed by a third party, e.g, an independent organization or a national authority, namely a "certification authority".



(a) Example mixed-criticality applications

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_{10}	τ_{11}	τ_{12}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}	τ_{30}	τ_{31}	τ_{32}	τ_{33}
N_1	x	3	x	2	3	x	2	6	2	4	6	4	x
N_2	4	5	3	3	x	6	3	9	6	9	10	5	4

(b) WCET and mapping restrictions

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_{10}	τ_{11}	τ_{12}	τ_{21}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}	τ_{30}	τ_{31}	τ_{32}	τ_{33}
SIL 1	x	x	x	2	3	3	4	3	4	x	x	x	x
SIL 2	x	x	x	4	5	4	8	7	7	5	5	8	9
SIL 3	13	14	12	9	8	8	11	13	12	11	9	15	15
SIL 4	29	20	21	16	14	15	19	22	23	20	18	25	26

(c) Development costs (kEuro)

Figure 2.1: Application model example

The current certification practice is "standards-based" [142], and requires that the product and the development processes fulfill the requirements and satisfy the objectives of a certain certification standard, depending on the application area. For example, IEC 61508 [94] is used in industrial applications, ISO 26262 [96] is for the automotive area, whereas DO 178B [140] refers to software for airborne systems.

During the engineering of a safety-critical system, the hazards are identified and their severity is analyzed, the risks are assessed and the appropriate risk control measures are introduced to reduce the risk to an acceptable level. A Safety-Integrity Level (SIL) captures the required level of risk reduction. SIL allocation is typically a manual process, which is done after performing hazard and risk analysis [160], although a few

researchers have proposed automatic approaches for SIL allocation [124]. SILs differ slightly among areas. For example, the avionics area uses five "Design Assurance Levels" (DAL), from DAL E (lest critical) to DAL A (most critical), while ISO 26262 specifies for the automotive area four "Automotive Safety Integrity Levels" (ASIL), from ASIL A (least critical) to ASIL D (most critical). However, the approach presented in this thesis is applicable to all safety-critical areas, regardless of the standard. SILs are assigned to tasks, from SIL 4 (most critical) to SIL 0 (non-critical).

Thus, we introduce the function $SIL : \mathcal{V}_i \rightarrow \{SIL\ k\}$, where $k \in \{0..4\}$, to capture the SIL of a task. The tasks of an application may have different SILs. The SILs for the example in Fig. 2.1a are presented next to the tasks.

2.1.2 Task Decomposition

During the early stages of the design of safety-critical systems, a SIL is allocated to each safety function. Safety functions are later implemented as software or hardware, or a combination of both. Let us consider a safety function of SIL i, to be implemented as software tasks. The certification standards allow several options. For example, the safety-function could be implemented as one task of SIL i or, using redundancy to increase dependability, as several redundant tasks of a lower SIL, e.g., SIL i-1. Decomposing a safety function of a higher SIL into several redundant tasks of lower SILs can reduce the development and certification costs, and could be the right choice in a particular context. For software redundancy, the standards recommend the use diversity, i.e., different implementations of the same functionality. This is because a fault (bug) in a software task will lead to a correlated failure in all of the tasks sharing the same implementation, unless software diversity is used. Often, one of the redundant tasks will implement a simpler (and maybe less accurate) algorithm as alternative diverse implementation.

Certification standards refer to this process as "SIL decomposition" and provide recommendations on the possible decompositions. For example, ISO/DIS 26262[1], Part 9, Section 5, provides the guide shown in Table 2.1 for SIL decomposition. Such a decomposition guide amounts to a "SIL algebra" [125], i.e., the SIL of the safety function is the sum of the SILs of the redundant tasks.

We assume that the safety functions are implemented as software tasks running on a distributed architecture. Let us consider a tasks τ_A which has to fulfill a safety requirement of SIL 3. According to Table 2.1, we can decompose task τ_A into two redundant tasks, e.g., τ_B with SIL 2 and τ_C of SIL 1. Task τ_B can be further decomposed into two SIL 1 tasks.

[1] As mentioned, ISO 26262 uses the concept of Automotive SIL, or ASIL. To simplify the discussion, we consider ASIL D to be SIL 4 and ASIL A to be SIL 1.

Table 2.1: ISO/DIS 26262 SIL decomposition schemes

SIL	Can be decomposed as
SIL 4	SIL 3 + SIL 1 or SIL 2 + SIL 2 or SIL 4
SIL 3	SIL 2 + SIL 1 or SIL 3
SIL 2	SIL 1 + SIL 1 or SIL 2
SIL 1	SIL 1

Furthermore, we assume that, for those tasks which are considered for decomposition, the engineer will specify a library $\mathcal{L}$ of possible decompositions based on the standard considered, similar to the library in Table 2.1. In this thesis, we are also interested to determine a decomposition of tasks such that the timing requirements are satisfied and the development costs are minimized. This design problem is presented in Section 3.1.4. There is very little work on automatic SIL allocation [124] and decomposition [125, 34]. Parker et al. [125] have proposed a Genetic Algorithm for SIL decomposition and Azevedo et al. [34] have proposed a Tabu Search metaheuristic. Both works aim at a SIL decomposition which reduces the development costs, and are interested in deriving a fault-tolerant architecture. The safety of the resulted architecture is evaluated using Fault-Tree Analysis [174].

Fig. 3.4a shows a two decomposition options for task τ_{11} of SIL 4 from application $\mathcal{A}_1$ in Fig. 3.3a. We define the decomposition function $D(\tau_i)$, $D(\tau_i) : \mathcal{V}_i' \rightarrow \mathcal{D}_i$, where $\mathcal{D}_i$ is a set of decomposition options, specified in the decomposition library $\mathcal{L}$. There are two decomposition options in Fig. 3.4a: D_1 in two tasks of SIL 2, namely τ_{11b} and τ_{11c}, and D_2 into tasks τ_{11f} of SIL 3 and τ_{11g} of SIL 1. Fig. 3.4a also shows how τ_{11}, once decomposed, is connected to the graph of application $\mathcal{A}_1$ from Fig. 3.3a. We assume that a decomposed task will be connected to the original application graph via two "connecting" tasks; one task which is distributing the input to the redundant decomposed tasks (τ_{11a} in Fig. 3.4a) and one task which is collecting the outputs (τ_{11d}). The SIL of the connecting tasks are given by the engineer based on the communication constraints as discussed in Section 2.1.4 and on the requirements from the standards.

The redundant decomposed tasks of lower SIL may have a lower WCET than the original task of higher SIL. Researchers [39] have considered the aspect of criticality-dependent WCET. Our model can also take into account a SIL-dependent WCET for each task. However, for simplicity, we do not consider this aspect in our examples, except in the case of SIL decomposition, as discussed next. One reason for a lower WCET could be that the redundant tasks may implement only part of the functionality of the original task, with some functionality implemented in the "connecting" tasks. Also, as mentioned in the case of software diversity, one of the tasks may use a simplified algorithm (which, for example, produces less accurate results), that will have a lower WCET. Yet another possibility for the WCET decrease could be the requirements on compiler optimization imposed by the standard. Such optimizations, for example,

are not allowed for SIL 4, but are allowed for lower SILs. Although compiler optimizations would affect mainly the average execution times, removing them could lead to a WCET increase in case of SIL 4 tasks.

The WCETs for our example task τ_{11} are shown in Fig. 3.4b. Note that in the decomposition option D_1, the sum of the WCETs of the tasks along the longest path of the task graph replacing τ_{11}, i.e., $\tau_{11a} \rightarrow \tau_{11c} \rightarrow \tau_{11d}$, has a value of 6 on N_1, compared to the WCET of 5 for the original τ_{11} on N_1.

2.1.3 Development Cost Model

The SIL assigned to a task will dictate the development processes and certification procedures that have to be followed. Standards provide checklists of objectives required to be fulfilled for each SIL. Depending on the SIL, the standard may also impose that some objectives to be satisfied with independence, to ensure an unbiased evaluation and to avoid misinterpretation of the requirements [140]. For example, for the verification process, independence is achieved by using tools and personnel other than those used throughout the development process.

SIL 0 functions are non-critical and do not impact the safety of the systems, thus are not covered by the standards. In the case of SIL 1, the processes are similar to those covered by quality management standards such as ISO 9001 [95]. SIL 2 involves more reviewing and testing. SIL 3 is significantly more difficult, and requires "semi-formal" methods. SIL 4 often mandates formal methods, increasing further the development costs.

The assessment of conformity to the checklist of objectives has to be performed by independent assessors. For SIL 1 is enough to have an independent person, whereas for SIL 2 an independent department is required. In the case of SIL 3 and SIL 4, an independent organization has to be used. Moreover, the number of objectives that have to be satisfied with independence is also growing. For example, in the case of DO-178B, the main difference between DAL A and DAL B is the number of objectives to be satisfied with independence: 25 out of 66 objectives are required for DAL A to be satisfied with independence, while for DAL B it is only 14 out of 65.

Software development cost estimation is a widely researched topic, and is beyond the scope of this thesis. The reader is directed to [100, 47] for reviews on this topic. One of the most influential software cost models is the Constructive Cost Model (COCOMO) [48]. Researchers have shown how to take into account the development costs during the design process of embedded systems [67].

The development of safety-critical systems is a highly structured and systematic process dictated by standards. These standards increase the development costs due to additional processes for software development and testing, qualification activities involved in compliance and increased process complexity, shown also by an IBM Rational study [93]. Additional development and certification costs may arise from using development and verification tools, e.g. for binary patching or to achieve partitioning using compiler and linker mechanisms, since these tools need to be *verified* or *qualified*. For DO-178B [140] this means to demonstrate that the tool satisfies the same objectives as the processes it automates, reduces or replaces, thus increasing the development and certification costs. Furthermore, [140] requires that the tool to be qualified only for use on the specific system where the tool is intended to be used, while the "use of the tool for other systems may need further qualification".

Because of the systematic nature of the development processes dictated by the standards, we assume that the engineer will be able to estimate the development effort required for a task. Hence, we define the development cost (DC) function $DC(\tau_i, SIL\ j)$ to capture the cost to develop and certify a task τ_i to safety integrity level $SIL\ j$. Fig. 2.1c shows an example of the development costs for each of the tasks in Fig. 2.1a. Knowing the DC for each task, we can compute this cost at the application level. The DC of application $\mathcal{A}_i$, denoted with $DC(\mathcal{A}_i)$, is the sum of the development costs of each task in the application. Similarly, we define the DC for the set of all the applications, $DC(\Gamma)$, as the sum of the costs for each application. An example certification cost estimation in person-days for an Air Traffic Control radio platform is presented in [139].

2.1.4 Protection Requirements

When several tasks of different SILs share the same processing element, the standards require that they are developed at the highest SIL among the SILs of the tasks, which is very expensive. Unless, the standards state, it can be shown that the implementation of the tasks is "sufficiently independent", i.e., there is both spatial and temporal separation among the tasks. Hence, tasks of different SILs have to be protected from each other. Otherwise, for example, a lower-criticality task could corrupt the code or data area of a higher-criticality task [57], or block the higher-criticality task from accessing the CPU, leading thus to a failure.

Protection also imposes constraints on the type of communication that is allowed. Thus, within an application, a task can only receive an input from a task of the same criticality level or higher than its own. In addition, we assume that there is no communication between two applications. Such constraints on communication may be relaxed by implementing a validation middleware, such as the one proposed by [179], which checks and upgrades the data from lower levels to be used at the higher integrity levels.

Standard practice in certain areas may place additional protection requirements. For example, it may be recommended that two tasks of SIL 4 from different applications should be protected, although they are at the same SIL level. To capture such requirements, and any additional protection requirements desired by the engineer, we define the protection requirements graph $\Pi(\mathcal{V}, \mathcal{E})$ as a bidirectional graph, where $\mathcal{V}$ represents the set of all tasks, and $\mathcal{E}$ is a set of edges. An edge $sr_{ij} \in \mathcal{E}$ is a separation requirement, which means that tasks τ_i and τ_j are not allowed to share a partition. The communication restrictions are captured implicitly by the structure of the application graphs $\mathcal{G}_i \in \Gamma$.

2.2 Architecture Model

We consider architectures composed of a set $\mathcal{N}$ of PEs which communicate via a TTEthernet network. Section 2.2.1 presents our partitioning model at the PE-level and Section 2.2.2 presents our communication-level models. The TTEthernet protocol is presented Section 2.2.3.

2.2.1 Partitioning at PE-Level

If two tasks are of different SILs, or if they have to be separated according to the protection requirements graph Π, we consider that the protection is achieved through a temporal- and space-partitioning scheme similar to Integrated Modular Avionics (IMA) [141]. Note that partitioning schemes similar to IMA are available in several application areas [135], not only in the avionics area. Space partitioning uses mechanisms such as a Memory Management Unit (MMU) to ensure that, for example, applications running on different partitions cannot corrupt the memory for the other applications. Temporal partitioning ensures the access of each application to the CPU, according to a predetermined partition table. A detailed discussion about partitioning is available in [141].

We denote the assignment of tasks to partitions using the function $\phi : \mathcal{V} \rightarrow \mathcal{P}$, where $\mathcal{V}$ is the set of tasks in the system and $\mathcal{P}$ is the set of partitions. On a processing element N_i, a partition $P_j \in \mathcal{P}$ is defined as the sequence $\mathcal{P}_{ij}$ of k partition slices p_{ij}^k, $k \geq 1$. A partition slice p_{ij}^k is a predetermined time interval in which the tasks mapped to N_i and allocated to the partition P_j are allowed to use N_i.

All the slices on a processor are grouped within a time interval called “Major Frame” (MF), i.e., in a *partition table*. This partition table is repeated periodically. The period T_{MF} of the Major Frame is given by the engineer and is the same on each PE. Several

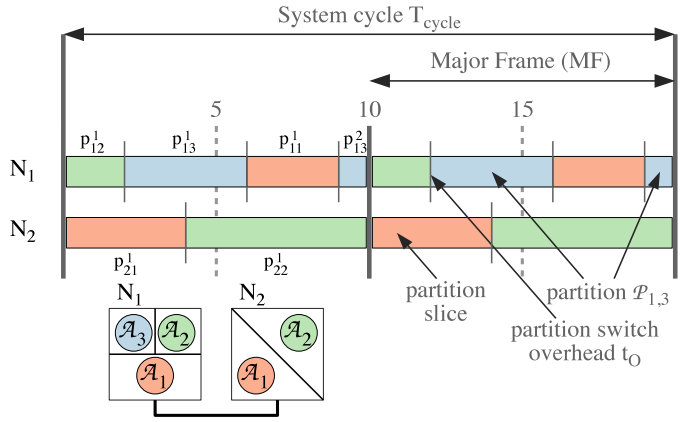


Figure 2.2: Partitioned architecture

MFs are combined together in a system cycle that is repeated periodically, with a period T_{cycle}. Within a T_{cycle}, the sequence and length of the partition slices are the same across MFs (on a given PE), but the contents of the slices (i.e., the tasks assigned to the slices) can differ.

Fig. 2.2 presents the partitions for 3 applications of different SILs, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, implemented on an architecture of 2 PEs, N_1 and N_2, with $T_{MF} = 10$ and $T_{cycle} = 2 \times T_{MF} = 20$. Using the partitions in the figure, the tasks of $\mathcal{A}_3$, for example, can execute only in partition P_3 on PE N_1 (the light blue rectangles in the timeline on N_1), composed of the sequence $\mathcal{P}_{1,3}$ of partition slices p^1_{13} and p^2_{13}. In this example, we assume that all the tasks of $\mathcal{A}_3$ have the same SIL. However, as mentioned in Section 2.1.1, the tasks of an application may have different SILs. Such tasks have to be placed in separate partitions.

The schedule tables $\mathcal{S}$ for the applications have to be constructed such that they take into account the partitions $\mathcal{P}$. Note that a task can extend its execution over several partition slices and MFs. When a task does not complete during a partition slice, its execution is suspended until its partition is activated again. Such an example is task τ_{31} on N_1 in Fig. 3.2b, which shows the schedule tables for the applications in Fig.2.1a. The time overhead due to partition switching is denoted with t_O, and our optimization approaches described in Chapter 3 take into account the partition switching overheads.

2.2.1.1 Elevation and Software-Based Protection

As mentioned, tasks of different SILs have to be placed in separate partitions. However, there might be situations when it would be beneficial (e.g., in terms of schedula-

bility) for two tasks of different SILs to share a partition. This can be achieved through *elevation*: increasing the SIL of the lower criticality task to the level of the higher criticality task. Although such elevation to a higher SIL is allowed by the standards, it will increase the development costs for the elevated task. For example, considering the application details from Fig. 2.1, in Fig. 3.2d task τ_{23} shares the partition with tasks τ_{12} on N_2, second MF. As task τ_{23} has a lower SIL than τ_{12}, which is SIL 3, it has to be elevated from SIL 1 to SIL 3. This is shown visually in the schedule by raising task τ_{23} slightly compared to the other tasks which are not elevated.

Such elevation may trigger the elevation of other tasks, depending on the protection requirements graph Π. For example, as mentioned in Section 2.1.4, we assume that a task can only receive inputs from predecessors of the same or higher SIL. This means that elevating a task τ_i to a higher SIL may trigger the elevation of its predecessors. This, in turn, can trigger the elevation of other tasks, if such predecessors will be assigned a higher SIL in another partition slice. This is the case for the tasks in application $\mathcal{A}_2$, shown in Fig. 3.2d using green rectangles. As τ_{22} and τ_{23} share the partition with τ_{12}, they have to be elevated from SIL 1 to SIL 3. This in turn triggers the elevation to SIL 3 of the predecessors of these tasks, namely tasks τ_{20} and τ_{21}, see the $\mathcal{A}_2$ graph in Fig. 2.1a. Furthermore, τ_{20} and τ_{21} share the partition with τ_{24} and τ_{25}, namely partition $\mathcal{P}_2$ on N_1, see Fig. 3.2d. Since tasks τ_{20} and τ_{21} were elevated to SIL 3, and tasks τ_{24} and τ_{25} are SIL 1, we need to elevate tasks τ_{24} and τ_{25} to SIL 3 to allow the sharing of partition $\mathcal{P}_2$. Thus, all the tasks of application $\mathcal{A}_2$ are elevated to SIL 3, increasing the development costs DC for this application from 19 to 61 kEuro (the development costs for these tasks are presented in Fig. 2.1c).

For lower SILs, an alternative to using elevation for partition sharing, is to use *software-based protection* mechanisms. Such protection mechanisms are typically used to provide spatial protection for tasks of SIL 1 and SIL 2. At these levels, spatial protection can also be obtained using methods such as *Software Fault Isolation* (SFI) [141], or compiler and linker mechanisms, which can guarantee separation of code and data for lower SIL tasks [75]. SFI (or sandboxing) [178] is a technique for memory protection and fault isolation, achieved either by binary patching or during compile time. This translates into a run-time overhead between 4%–20%, depending on the type of instructions which are sandboxed, on the CPU architecture and on the implementation [178, 153]. A survey of isolation techniques is compiled by [176].

Our partitioning model can take into account any of the available protection mechanisms. Also, as discussed, some protection mechanisms may introduce additional overheads, such as performance overhead and/or a development and certification cost increase. Our model captures these overheads.

2.2.2 Communication Network Model

A TTEthernet network is composed of a set of clusters. Each cluster consists of End Systems (ESes) interconnected by links and Network Switches (NSes). The links are full duplex, allowing thus communication in both directions, and the networks can be multi-hop. An example cluster is presented in Fig. 2.3, where we have 4 ESes, ES_1 to ES_4, and 3 NSes, NS_1 to NS_3. We address design optimizations performed at the cluster-level. Each ES consists of a PE and a network interface card (NIC). The architecture model at the processor level is presented in Section 2.2.

We model a TTEthernet cluster as an undirected graph $\mathcal{G}_C(\mathcal{V}_C, \mathcal{E}_C)$, where $\mathcal{V}_C = \mathcal{ES} \cup \mathcal{NS}$ is the set of end systems ($\mathcal{ES}$) and network switches ($\mathcal{NS}$) and $\mathcal{E}$ is the set of physical links. For Fig. 2.3, $\mathcal{V}_C = \mathcal{ES} \cup \mathcal{NS} = \{ES_1, ES_2, ES_3, ES_4\} \cup \{NS_1, NS_2, NS_3\}$, and the physical links $\mathcal{E}_C$ are depicted with thick, black, double arrows.

A *dataflow path* $dp_i \in \mathcal{DP}$ is an ordered sequence of dataflow links connecting one sender to one receiver. For example, in Fig. 2.3, dp_1 connects ES_1 to ES_3, while dp_2 connects ES_1 to ES_4 (the dataflow paths are depicted with green arrows). A *dataflow link* $dl_i = [\nu_j, \nu_k] \in \mathcal{DL}$, where $\mathcal{DL}$ is the set of dataflow links in a cluster, is a directed communication connection from ν_j to ν_k, where ν_j and $\nu_k \in \mathcal{V}_C$ can be ESes or NSes. Using this notation, a dataflow path such as dp_1 in Fig. 2.3 can be denoted as [[ES_1, NS_1], [NS_1, NS_2], [NS_2, ES_3]].

The space partitioning between messages of different criticality transmitted over physical links and network switches is achieved through the concept of *virtual link*. Virtual links are defined by ARINC 664p7 [7], which is implemented by the TTEthernet protocol, as a "logical unidirectional connection from one source end system to one or more destination end systems".

Let us assume that in Fig. 2.3 we have two applications, $\mathcal{A}_1$ and $\mathcal{A}_2$. $\mathcal{A}_1$ is a high criticality application consisting of tasks τ_1 to τ_3 mapped on ES_1, ES_3 and ES_4, respectively. $\mathcal{A}_2$ is a non-critical application, with tasks τ_4 and τ_5 mapped on ES_2 and ES_3, respectively. τ_1 sends message m_1 to τ_2 and τ_3. Task τ_4 sends message m_2 to τ_5. With TTEthernet, a message has a single sender and may have multiple receivers. The flow of these messages will intersect in the physical links and switches. Virtual links are used to separate the highly critical message m_1 from the non-critical message m_2. Thus, m_1 is transmitted over virtual link vl_1, which is isolated from virtual link vl_2, on which m_2 is sent, through protocol-level temporal and spatial mechanisms (which are briefly presented in Section 2.2.3).

We denote the set of virtual links in a cluster with $\mathcal{VL}$. A virtual link $vl_i \in \mathcal{VL}$ is a directed tree, with the sender as the root and the receivers as leafs. For example, vl_1, depicted in Fig. 2.3 using dot-dash red arrows, is a tree with the root ES_1

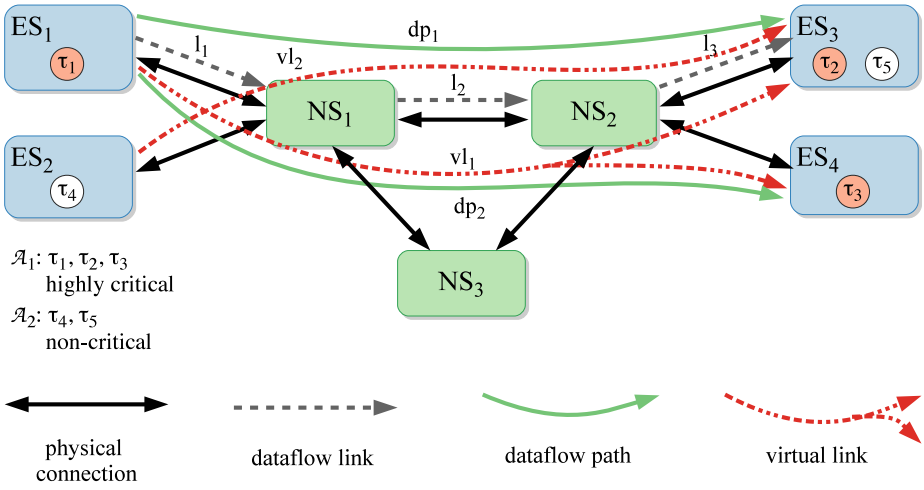


Figure 2.3: TTEthernet cluster example

and the leafs ES_3 and ES_4. Each virtual link is composed of a set of dataflow paths, one such dataflow path for each root-leaf connection. More formally, we denote with $\mathcal{R}_{VL}(vl_i) = \{\forall dp_j \in \mathcal{DP} | dp_j \in vl_i\}$ the routing of virtual link vl_i. For example, in Fig. 2.3, $\mathcal{R}_{VL}(vl_1) = \{dp_1, dp_2\}$.

For a given message, with one sender and multiple receivers, there are several virtual links which can be used for transmission. For example, in Fig. 2.3, message m_2 from ES_2 to ES_3 can be transmitted via vl_2, containing dataflow path $dp_3 = [[ES_2, NS_1], [NS_1, NS_2], [NS_2, ES_3]]$, or via $vl_3 = \{dp_4\}$, with $dp_4 = [[ES_2, NS_1], [NS_1, NS_3], [NS_3, NS_2], [NS_2, ES_3]]$. Deciding each virtual link vl_i for a message m_i is a routing problem: we need to decide which route to take from a set of possible routes. This routing is determined by our optimization approach presented in Chapter 4 and, for real life systems, which can contain tens to hundreds of connected ESes and NSes, this is not a trivial problem.

2.2.2.1 Frames

TTEthernet transmits data using *frames*. The TTEthernet frame format fully complies with the ARINC 664p7 specification [7], and is presented in Fig. 2.4. A complete description of the ARINC 664p7 frame fields can be found in [7]. Messages are transmitted in the payload of frames. A bit pattern specified in the frame header identifies the traffic class of each frame (TT, RC or BE). The total frame header (Ethernet header and ARINC 664p7 header) is of 42 B, while the payload of each frame varies between

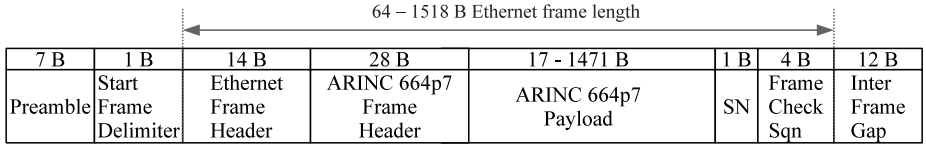


Figure 2.4: Simplified frame format

a minimum of 17 B and a maximum of 1471 B. In case a frame carries data smaller than 17 B, the frame payload will be padded with zeroes to reach the minimum payload of 17 B. Thus, as shown in Fig. 2.4, the total protocol overhead for a frame (including the frame header, preamble, start frame delimiter and interframe gap) varies from 67 B, for data bigger than 17 B, to 83 B for data of 1 B.

The size $m_i.size$ for each message $m_i \in \mathcal{M}$ is given, where $\mathcal{M}$ is the set of all messages. As mentioned, TTEthernet supports three traffic classes: time-triggered (TT), rate constrained (RC) and best effort (BE). We assume that the system engineer has decided the traffic classes for each message. We define the sets $\mathcal{M}^{TT}$, $\mathcal{M}^{RC}$ and $\mathcal{M}^{BE}$, respectively, with $\mathcal{M} = \mathcal{M}^{TT} \cup \mathcal{M}^{RC} \cup \mathcal{M}^{BE}$. In addition, for the TT and RC messages we know their periods / rate and deadlines, $m_i.period$ or $m_i.rate$, and $m_i.deadline$, respectively. Furthermore, we also know the safety-criticality level for each message. Safety-criticality levels, referred to as Safety Integrity Levels, or SILs, are discussed in Section 2.1.1. RC messages are not necessarily periodic, but have a minimum interarrival time. We define the rate of an RC message m_i as $m_i.rate = 1/m_i.period$.

So far, researchers have assumed that each message $m_i \in \mathcal{M}$ is transmitted in the payload of a dedicated frame f_i. This was also the assumption of our earlier work, which focused only on the scheduling of TT frames [169]. However, the payload of a frame can in practice carry several messages. Moreover, messages can also be fragmented into several pieces, each carried by a different frame. We extended our optimization presented in Chapter 4 to also determine the *fragmenting* and the *packing* of messages into frames.

The *fragmenting* of messages into message fragments is denoted with $\Phi_m(m_i) = \{\forall m_j \in \mathcal{M}^+ | m_j \in m_i\}$, where $\mathcal{M}^+$ is the set which contains all the message fragments resulted from fragmenting, and the messages which were not fragmented. The message fragments $m_j \in \Phi_m(m_i)$ inherit the temporal constraints of the original message, and have equal sizes. For example, let us consider $\mathcal{M} = \{m_1, m_2\}$. In this case, $\mathcal{M}^+ = \mathcal{M}$. Message m_1 has a period and deadline of 30 ms, and a size of 300 B. Message m_2 has a deadline and period of 24 ms, and $m_2.size = 1200\ B$. We fragment m_2 into 3 same-sized message fragments, such that $\Phi_m(m_2) = \{m_3, m_4, m_5\}$ and m_3, m_4 and m_5 have the same period and deadline as m_2, but their size is 400 B. After fragmenting m_2, $\mathcal{M}^+ = \{m_1, m_3, m_4, m_5\}$.

The *packing* of messages and message fragments into frames is denoted with $\mathcal{K}: \mathcal{M}^+ \rightarrow \mathcal{F}$, $\mathcal{K}(m_j) = f_i$, where $\mathcal{F}$ is the set of all the frames in the cluster. We define $\mathcal{F}^{TT}$, $\mathcal{F}^{RC}$ and $\mathcal{F}^{BE}$, respectively, with $\mathcal{F} = \mathcal{F}^{TT} \cup \mathcal{F}^{RC} \cup \mathcal{F}^{BE}$. A bit pattern specified in the frame header identifies the traffic class. Each frame is assigned to a virtual link, which specifies among others, the routing for the frame. In TTEthernet, each virtual link has assigned only one frame. The function $\mathcal{M}_F(f_i) = vl_i, \mathcal{M}_F : \mathcal{F} \rightarrow \mathcal{VL}$ captures this assignment of frames to virtual links. Let us consider the example given in Fig. 2.3, with message m_1 sent from ES_1 to ES_3 and ES_4. We assume that m_1 is packed by frame f_1, $\mathcal{K}(m_1) = f_1$. In this case, f_1 is assigned to vl_1, $\mathcal{M}_F(f_1) = vl_1$. Fig. 2.3 shows vl_1 routed along the shortest route.

The properties of the frames are derived based on what messages or message fragments are packed, such that the timing constraints are satisfied. Let us consider the prior example. The packing of message m_1 and message fragment m_3, $m_1, m_3 \in \mathcal{M}^+$, in frame $f_1 \in \mathcal{F}$, is denoted with $\mathcal{K}(m_1) = f_1$ and $\mathcal{K}(m_3) = f_1$, respectively. In this case, the data packed by f_1 has a size of 700 B. Note that, unlike in the case of the EtherCAT [19] protocol, not all fragmenting and packing combinations are valid (e.g., messages packed into a frame must have the same source and destination ESes and must be of the same safety-criticality level). Also, the timing properties of the new frame depends on the timing constraints of messages. Our optimization takes care of these aspects, see Section 3.2 for details. Knowing the size of a frame f_j and the given speed of a dataflow link $[\nu_m, \nu_n]$, we can determine the transmission duration $C_j^{[\nu_m,\nu_n]}$ of f_j on $[\nu_m, \nu_n]$.

2.2.3 The TTEthernet Protocol

Let us illustrate how the TTEthernet protocol works using the example in Fig. 2.5, where we have two end systems, ES_1 and ES_2, and three network switches, NS_1 to NS_3. Task τ_2 on ES_1 sends the TT message m_2 to task τ_4 mapped on ES_2, while task τ_1 on ES_1 sends the RC message m_1 to task τ_3 on ES_2. Let us assume that tasks τ_1 and τ_3 are part of application $\mathcal{A}_1$ and tasks τ_2 and τ_4 belong to application $\mathcal{A}_2$. Furthermore, $\mathcal{A}_1$ and $\mathcal{A}_2$ are of different safety criticality. The separation of the applications is achieved at the CPU-level through partitioning. Thus, tasks τ_1 and τ_3 are placed in partitions $\mathcal{P}_{1,1}$ and $\mathcal{P}_{2,2}$, respectively, while tasks τ_2 and τ_4 are assigned to partitions $\mathcal{P}_{1,2}$ and $\mathcal{P}_{2,1}$, see Fig. 2.5.

Message m_1 is sent within application $\mathcal{A}_1$ and packed in frame f_1. m_2 is sent within $\mathcal{A}_2$ and packed into the frame f_2. The fragmenting and packing of messages is performed at application level. The different criticality frames are separated by assigning them to two different virtual links, vl_1 and vl_2 (not depicted in the figure). Frames f_1 and f_2 have to transit the switch NS_1, which also forwards frames f_3 and f_4, from NS_2 and NS_3, respectively, see Fig. 2.5.

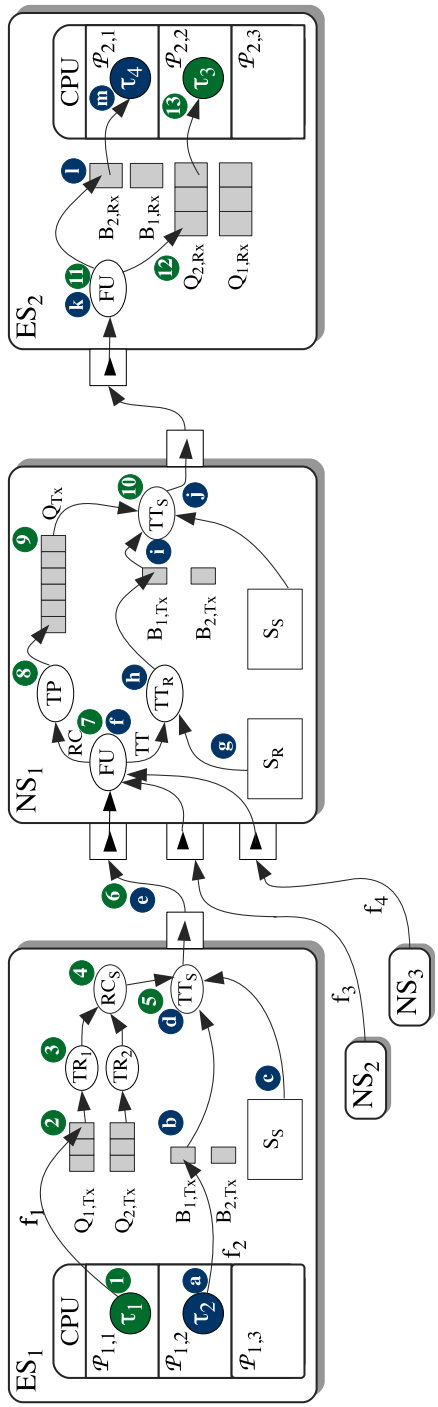


Figure 2.5: TT and RC message transmission example

2.2.3.1 Time-Triggered Transmission

In this section we present how TT frames are transmited by TTEthernet, using the example of the TT message m_2 sent from task τ_2 on ES_1, to task τ_4 on ES_2. We depict each step of the TT transmission on Fig. 2.5 and mark it with a letter from (a) to (m) on a blue background.

Thus, in the first step denoted with (a), task τ_2 packs m_2 into frame f_2 and in the second step (b), f_2 is placed into buffer $B_{1,Tx}$ for transmission. Conceptually, there is one such buffer for every TT frame sent from ES_1. TT communication is done according to static communication schedules determined offline and stored into the ESes and NSes. The complete set of local schedules in a cluster are denoted with $\mathcal{S}$. The schedules $\mathcal{S}$ are derived by our optimization approach. Thus, in step (d), the scheduler task TT_S will send f_2 to NS_1 at the time specified in the send schedule S_S stored in ES_1 (c). There are several approaches to the synchronization of tasks (which could be TT or ET) and TT messages [122]. Often, TT tasks are used in conjunction with TT messages, and the task and message schedules are synchronized such that the task is scheduled to finish before the message is scheduled for transmission.

Next, f_2 is sent on a dataflow link to NS_1 (e). The computational logic of the TTEthernet protocol is implemented in hardware and, conceptually, consists of several hardware tasks working in parallel to implement the protocol services. Such is the case of the Filtering Unit (FU) task, which is invoked every time a frame is received by an NS. The FU checks the integrity and validity of frame f_2 (see step (f)) and forwards it to the TT receiver task TT_R (h), which copies it into the sending buffer $B_{1,Tx}$ for later transmission.

The separation mechanisms implemented by TTEthernet to isolate mixed-criticality frames, such as f_1 and f_2 in our example, are spread across several hardware tasks. In addition, TTEthernet provides fault-tolerance services, such as fault-containment, to the application level. For example, if a task such as τ_2 becomes faulty and sends more messages than scheduled (called a "babbling idiot" failure), the TT sender task TT_S on ES_1 will protect the network as it will only transmit messages as specified in the schedule table S_S.

Also, a TT receiver task TT_R in an NS will rely on a receive schedule S_R (g) stored in the switch to check if a TT frame has arrived within a specified receiving window. This window is determined based on the sending times in the send schedules (schedule S_S on ES_1 for the case of frame f_2), the precision of the clock synchronization mechanism and the "integration policy" used for integrating the TT traffic with the RC and BE traffic (see next subsection for details). TT message frames arriving outside of this receiving window are considered faulty. In order to provide virtual link isolation and fault-containment, a TT receiver task TT_R will drop such faulty frames.

The schedules $\mathcal{S}$ contain the sending times and receiving windows for all the frames transmitted during an application cycle, T_{cycle}. A periodic frame f_i may contain several instances (a *frame instance* is the equivalent of the periodic *job* of a task) within T_{cycle}. We denote the *x-th* instance of frame f_i with $f_{i,x}$. The sending time of a frame f_i relative to the start time of its period is called the *offset*, denoted with $f_i.offset$. In [169] we have assumed a TTEthernet implementation where within an application cycle, the offset of a frame may vary from period to period. In this thesis we consider a realistic implementation, where the sending time offset of a frame is identical in all periods, with the advantage of reducing the size needed to store the schedules.

Let us continue to follow the transmission of f_2 in Fig. 2.5. The frame has arrived in NS_1 and has been placed in $B_{1,Tx}$ (h). Next, f_2 is sent by the TT sender task TT_S in NS_1 to ES_2 at the time specified in the TT send schedule S_S in NS_1. When f_2 arrives at ES_2 (k), the FU task will store the frame into a dedicated receive buffer $B_{2,Rx}$ (l). Finally, when task τ_4 is activated, it will read f_2 from the buffer (m).

2.2.3.2 Rate Constrained Transmission

This section presents how RC traffic is transmitted using the example of frame f_1 sent from τ_1 on ES_1 to τ_3 on ES_2. Similarly to the discussion of TT traffic, we mark each step in Fig. 2.5 using numbers from (1) to (13), on a green background.

Thus, τ_1 packs message m_1 into frame f_1 (1) and inserts it into a queue $Q_{1,Tx}$ (2). Conceptually, there is one such queue for each RC virtual link. RC traffic consists of event-triggered messages. The separation of RC traffic is enforced through "bandwidth allocation". Thus, for each virtual link vl_i carrying an RC frame f_i the engineer decides the Bandwidth Allocation Gap (BAG). A BAG is the minimum time interval between two consecutive instances of an RC frame f_i. The BAG is set in such a way to guarantee that there is enough bandwidth allocated for the transmission of a frame on a virtual link, with $BAG_i \leq 1/f_i.rate$.

The BAG is enforced by the Traffic Regulator (TR) task. Thus, TR_1 in ES_1 in Fig. 2.5 will ensure that each BAG_1 interval will contain at most one instance of f_1 (3). Therefore, even if a frame is sent in bursts by a task, it will leave the TR task within a specified BAG. Thus, the maximum bandwidth used by a virtual link vl_i which transmits an RC frame f_i is $BW(vl_i) = f_i.size/BAG(vl_i)$. The BAG for each RC frame is computed offline, based on the requirements of the messages it packs.

Several messages will be sent from an ES. Let us first discuss how RC messages are *multiplexed*, and then we will discuss the *integration* with the TT traffic. In an ES, the RC scheduler task RC_S (such as the one in ES_1) will multiplex several RC messages (4) coming from the traffic regulator tasks, TR_i, such as TR_1 and TR_2 in ES_1. Fig. 2.6

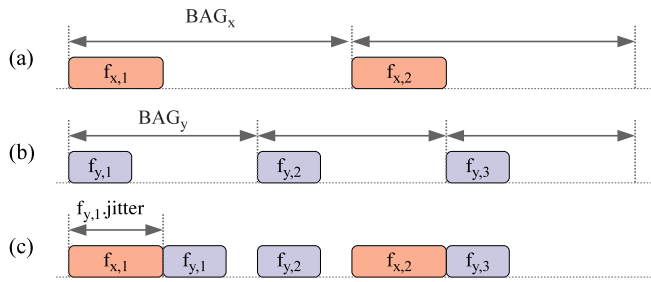


Figure 2.6: Multiplexing two RC frames

depicts how this multiplexing is performed for the frames f_x and f_y with the sizes and BAGs as specified in Fig. 2.6(a) and (b), respectively. Fig. 2.6(c) shows how the two frames will be sent on the outgoing dataflow link $[ES_1, NS_1]$ by the RC_S task. In the case several TRs attempt to transmit messages at the same time, due to the multiplexer, the frames waiting to be transmitted will be affected by jitter. This is the case of f_y in Fig. 2.6(c), which is delayed to allow for the transmission of f_x. Thus, the $f_{y,1}.jitter$ jitter for $f_{y,1}$ equals to the transmission duration of f_x.

RC traffic also has to be integrated with TT traffic, which has higher priority. Thus, RC frames are transmitted only when there is no TT traffic on the dataflow link. Hence, for our example on ES_1, the TT_S task on ES_1 will transmit frame f_1 (5) to NS_1 on the dataflow link $[ES_1, NS_1]$ only when there is no TT traffic (6). With integration, contention situations can occur when a TT frame is scheduled for transmission, but an RC frame is already transmitting.

There are three approaches in to handle such situations [28, 158]: (i) shuffling, (ii) pre-emption and (iii) timely block. (i) With *shuffling*, the higher priority TT frame is delayed until the RC frame finishes the transmission. Thus, in the worst-case scenario, the TT frame will have to wait for the time needed to transmit the largest Ethernet frame, which is 1542 Bytes. In the case (ii) of *pre-emption*, the RC frame is pre-empted, and its transmission is restarted after the TT frame finished transmitting. In the case (iii) of *timely block*, the RC frame is blocked (postponed) from transmission on a dataflow link if a TT frame is scheduled to be sent before the RC frame would complete its transmission. Note that, as discussed in the previous subsection, the integration approaches have an impact on the receiving window of a TT frame, which has to account for the delays due to shuffling, for example.

When the RC frame f_1 arrives at NS_1, the Filtering Unit task (7) will check its validity and integrity. As mentioned, TTEthernet provides services to separate the mixed-criticality frames, such that a faulty transmission of a lower-criticality frame will not impact negatively a higher-criticality frame. Fault-containment at the level of RC vir-

tual links is provided by the Traffic Policing (TP) task, see NS_1 in Fig. 2.5. TP implements an algorithm known as *leaky bucket* [7, 28], which checks the time interval between two consecutive instances on the same virtual link. If this interval is shorter than the specified BAG time corrected by the maximum allowed transmission jitter, the frame instance is dropped. Thus, the TP function prevents a faulty ES to send faulty RC frames (more often than allowed) and thus to disturb the network.

After passing the checks of the TP task (8), f_1 is copied to the outgoing queue Q_{Tx} (9). Throughout this thesis we assume that all the RC frames have the same priority, thus the TT_S (10) will send the RC frames in Q_{Tx} in a FIFO order, but only when there is no scheduled TT traffic. At the receiving ES, after passing the FU (11) checks, f_1 is copied in the receiving $Q_{2,Rx}$ queue (12). Finally, when τ_3 is activated, it will take f_1 (13) from this queue.

CHAPTER 3

Design Optimizations at the Processor-Level

In this chapter we focus on design optimizations at the processor-level, hence we consider a simple statically scheduled bus where the communication takes place according to a static schedule table computed offline. Chapter 4 presents the optimization strategies at the communication level. At the processor-level, we assume the platform implements partitioning mechanisms similar to IMA [141](see Section 2.2.1). We presented the architecture and application models we consider in Chapter 2, and we talked about partitioning and how partitioning constrains the way tasks use the processor in Section 2.2.1.

We begin by presenting the design optimization problems addressed in the chapter. The problems are illustrated using four motivational examples. Then we present the optimization strategies we propose to solve these problems. We conclude the chapter with the experimental evaluations of the proposed optimizations. We used several synthetic and real-life benchmarks for evaluating the algorithms.

3.1 Problem Formulation

The problem we are addressing in this thesis can be formulated as follows: given a set Γ of applications, the criticality level $SIL(\tau_i)$ of each task τ_i, the library of SIL decompositions $\mathcal{L}$, the separation requirements Π between the tasks, an architecture consisting of a set $\mathcal{N}$ of processing elements, the size of the major frame T_{MF} and the application cycle T_{cycle}, we are interested to find an implementation Ψ such that all applications meet their deadlines and the development costs are minimized. Deriving an implementation Ψ means deciding on (1) the SIL decomposition D of the tasks for which the designer has provided alternatives in the library $\mathcal{L}$, (2) the mapping M of tasks to PEs taking into account the mapping restrictions, (3) the set $\mathcal{P}$ of partition slices on each processor, including their order and size, (4) the assignment ϕ of tasks to partitions and (5) the schedule $\mathcal{S}$ for all the tasks and messages in the system.

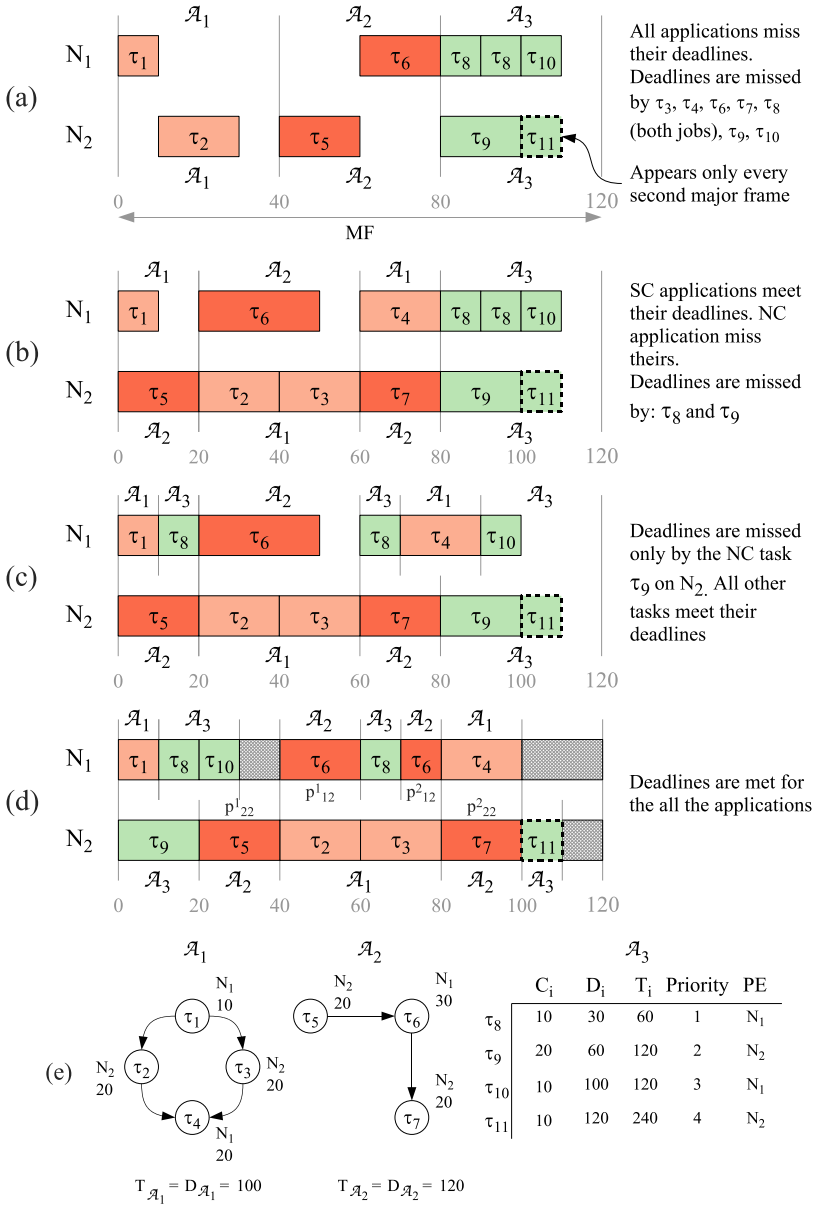
We present four motivational examples that show several aspects of this problem. First, we consider the problem of optimizing the partition time slots, assuming the mapping of tasks to processors is given and fixed. Second, we extend the problem by optimizing the partitioning and the mapping and the same time. In the first two examples we do not consider the issue of cost. Thus, we do not allow partition sharing by tasks of different criticality levels (see Section 2.2.1.1), and we do not consider task decomposition (see Section 2.1.2). Next, we remove these constraints. Third, we optimize the partitioning and mapping, but also allow tasks to share the same partition by elevating the tasks of lower criticality levels to the highest criticality level. This will increase the cost of the system. Fourth, we optimize the mapping, partitioning and partition sharing at the same time, but also consider task decomposition as a way to lower development costs.

Please note that, for simplicity, we ignore the partition switch overhead in the following examples.

3.1.1 Optimization of Time-Partitions

First, we consider the case where the mapping of tasks to processor has already been decided by the system engineer (e.g., due to modularity or physical constraints). At this point, we do not consider the issue of cost, i.e., partition sharing (see Section 2.2.1.1) or task decomposition (see Section 2.1.2). To simplify this example, we consider only two criticality levels: safety-critical and non-critical. We assume the safety-critical applications are scheduled using non-preemptive static cyclic scheduling (SCS), and the non-critical tasks are scheduled using preemptive fixed-priority scheduling (FPS)[1].

[1]For this problem we consider a Universal Communication Model bus [69], which has a dynamic segment used for messages exchanged by FPS tasks.



	C_i	D_i	T_i	Priority	PE
τ_8	10	30	60	1	N_1
τ_9	20	60	120	2	N_2
τ_{10}	10	100	120	3	N_1
τ_{11}	10	120	240	4	N_2

Figure 3.1: Motivational example

Let us illustrate the problem using the example in Fig. 3.1 where we have two SC applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, and one NC application, $\mathcal{A}_3$ (see Fig. 3.1e). For the SC applications, each task has next to it the PE it is mapped to and the worst-case execution time. The period and deadline for the applications are presented under the application graph. The NC tasks are scheduled using FPS and thus have their worst-case execution time C_i, deadline D_i, period T_i, priority and their PE, specified in the table. We have set T_{MF}=T_{cycle}=120 ms. To simplify the discussion, we assume that all NC tasks are released at time 0 and the communication costs are ignored.

Note that, we consider the mapping of tasks to processing elements as fixed, and given as indicated in the figure. Very often, the mapping decision is taken based on the experience and preferences of the engineer, considering aspects such as the functionality implemented by the task and the type of processing elements available, legacy constraints, proximity to sensors and actuators. This could be the reason, for example, that tasks τ_1 and τ_2 of $\mathcal{A}_1$ are mapped to different processing elements. Many tasks, however, do not exhibit certain particular features or requirements which lead to an obvious mapping decision. We address the problem of mapping optimization in Section 3.1.2.

A simple way to do the partitioning is to divide the major frame equally among the 3 applications and to use the same partitioning slices on each PE, as depicted in Fig. 3.1a. The thin light grey lines are the borders for the partitions slices. Above, respectively under, each partition slice is specified the application it is assigned to. In this case, none of the applications meet their deadlines. Tasks τ_3 and τ_4 from $\mathcal{A}_1$ and τ_7 from $\mathcal{A}_2$ do not even fit into the system cycle. Note that the deadlines for the NC tasks are measured from their release time. Task τ_8 is released twice during the MF, at time 0 and 60. Task τ_{11} is released every second MF. The schedulability of a solution is defined using the degree of schedulability, defined in Eq. 3.3. For SCS tasks, the degree of schedulability is the sum of slacks available between the completion time R_i of an application $\mathcal{A}_i$ and its deadline D_i. For SCS applications, R_i is determined by the scheduling in Section 3.4. For the FPS tasks, the degree of schedulability is calculated at the task level. The completion time R_i of a task τ_i is determined using the response time analysis presented in Section 3.5.

The scheduling of SCS tasks is presented in Section 3.4 and the schedulability analysis for the FPS tasks is presented in Section 3.5.

In order to better accommodate the SC applications, we can try to adjust the size of the slices and introduce a new slice for $\mathcal{A}_1$ on N_1 and for $\mathcal{A}_2$ on N_2, as shown in Fig. 3.1b. The SC applications meet their deadlines, but the NC tasks τ_8 and τ_9 miss in this case theirs. Note that although the slices have the same sizes on the two PEs, they are assigned to different applications.

Fig. 3.1c presents a way to make the NC task τ_8 on N_1 meet its deadline. The extra space from the first partition slice associated to $\mathcal{A}_1$ on N_1 is assigned to $\mathcal{A}_3$, and the

second partition slice of $\mathcal{A}_1$ is shifted to the right. In this case, both jobs of τ_8 will meet their deadlines. However the NC task τ_9 on N_2 still has a deadline miss.

With the solution proposed in Fig. 3.1d, the partitioning has been optimized such that all deadlines are met. In addition, we have also created 3 unused partition slices, depicted with a light grey rectangle, which can be used, for example, for future upgrades. This was managed by moving the time partition slice for the NC task τ_9 at the beginning of the partition table on PE N_2 and by splitting the SC task τ_6 on PE N_1 to execute in two different partition slices.

This example shows that the sequence and length of the partition slices has to be carefully optimized in order to find schedulable implementations.

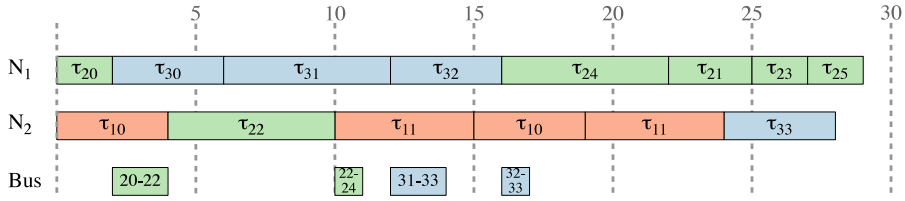
3.1.2 Partition-Aware Mapping Optimization

In the previous subsection we focused on the problem of optimizing the partitioning, while considering that the engineer has previously decided the mapping of tasks to processors. Here we assume the mapping is not given and that it has to be decided by our optimization. Note that here we do not yet consider partition sharing by tasks of different criticality, which is discussed in Section 3.1.3, nor do we consider task decomposition, which is discussed in Section 3.1.4.

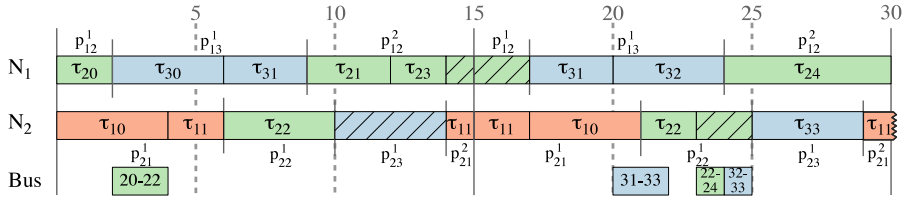
For simplicity, we consider all applications are scheduled using SCS. We have shown in the previous section how FPS can be used inside a partition. Compared to the example in Section 3.1.1, here we consider that tasks in an application can have different criticality levels (see Section 2.1.1).

Let us illustrate the problem using the mixed-criticality applications $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ from Fig. 2.1a, to be implemented on two PEs, N_1 and N_2. We initially do not consider task τ_{12}, i.e., it is not part of application $\mathcal{A}_1$. We have set T_{MF} to 15 time units and $T_{cycle} = 2 \times T_{MF} = 30$. The development cost, considering the model from Section 2.1.3 and the lowest SILs of the tasks from Fig. 2.1a is 73 kEuro.

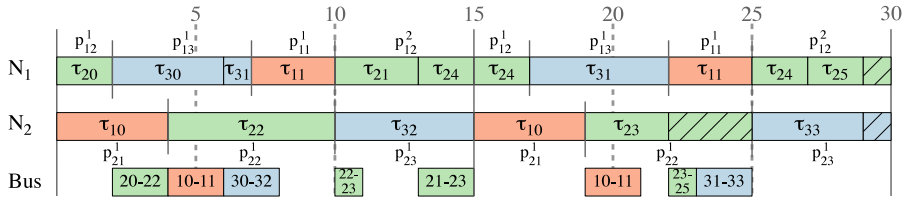
Let us first consider the case when the mapping and partitioning optimizations are performed separately. Thus, Fig. 3.2a presents the mapping and schedules for the case when there is no partitioning, i.e., the tasks do not have to be separated, and they can use the PEs without restrictions. The mapping and scheduling are optimal in terms of schedulability, captured by the "degree of schedulability" metric from Eq. 3.3, which is the sum of the slacks available between the completion time R_i of an application graph $\mathcal{A}_i$ and its deadline D_i. The "degree of schedulability" cost function is presented in Eq. 3.3 in Section 3.3. In Fig. 3.2a we show the schedules on each resource, namely, the PEs N_1 and N_2 and the bus, using a Gantt chart. The messages on the bus are labeled



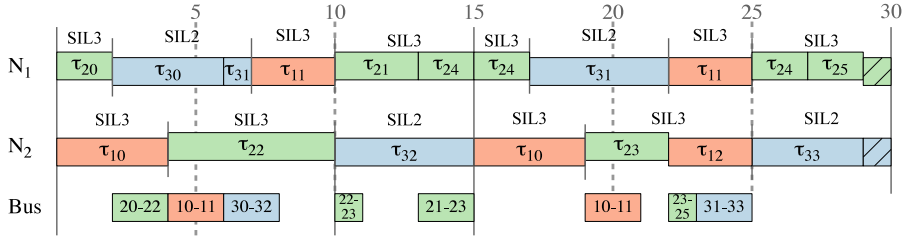
(a) Optimal mapping and schedules, without considering partitions



(b) Partitioning, using the previously obtained mapping. τ_{25} and the second instance of τ_{11} do not fit in the schedule



(c) By remapping tasks τ_{11}, τ_{23} and τ_{32}, and by optimizing the time partitions we manage to successfully schedule all the applications



(d) By elevating τ_{22} and τ_{23} to SIL 3, and thus all the tasks in $\mathcal{A}_2$, we manage to successfully schedule all applications

Figure 3.2: Motivational example

with the indices of the sender and receiver task, e.g., the first message on the bus, "20–22" is sent from task τ_{20} to τ_{22}. The dashed vertical lines are timeline guides to help with the visualization of the schedule, and should not be interpreted as partitions, since we ignore partitions in Fig. 3.2a.

Next, using this optimal mapping, we are interested to obtain the partitions and the schedules, such that, the separations are enforced and the schedule lengths are minimized with the goal of producing a schedulable implementation. Thus, Fig. 3.2b presents the optimal partitions and schedules (in terms of the same cost function from Eq. 3.3), considering the fixed mapping decided in Fig. 3.2a. The continuous line at time 15 represents the major frame boundary, while the shorter continuous lines, such as the one between tasks τ_{20} and τ_{30} represent partition slice boundaries. The partition slices are denoted with the notation p_{ij}^{k} introduced in Section 2.2.1. We mark the unused CPU time of a partition slice with a hatching pattern, as is the case with partition slice p_{12}^{1} on N_1 in the second MF assigned to $\mathcal{A}_2$.

With partitioning, tasks can only execute in their assigned partition. Hence, partitioning may lead to unused slack in the schedule, even in the case of an optimal partitioning and schedule, as depicted in Fig. 3.2b. In this case, although application $\mathcal{A}_3$ is schedulable, task τ_{25} of $\mathcal{A}_2$ and the second instance of task τ_{11} of $\mathcal{A}_1$ do not fit into the schedule, and thus applications $\mathcal{A}_1$ and $\mathcal{A}_2$ are not schedulable.

Our approach presented in Section 3.2.2 is to perform the optimization of mapping and partitioning at the same time, and not separately. By deciding simultaneously the mapping and partitioning we have a better chance of obtaining schedulable implementations. Such a solution is depicted in Fig. 3.2c, where all applications are schedulable. Compared to the solution in Fig. 3.2b, we have changed the mapping of tasks τ_{23} and τ_{32} from N_1 to N_2 and of task τ_{11} from N_2 to N_1, and we have resized the partition slices and changed the schedule accordingly. This example shows that by optimizing the mapping at the same time with partitioning we are able to obtain schedulable implementations.

3.1.3 Partition-Sharing Optimization

However, there might be cases when obtaining schedulable implementations is not possible, even if mapping and partitioning are considered simultaneously. For example, let us consider a similar setup as in the previous section, with the only difference that we add task τ_{12} to application $\mathcal{A}_1$, see Fig. 2.1a. The development costs, considering the addition of τ_{12} at SIL 3 are now 85 kEuro. Due to the addition of τ_{12} we are now unable to obtain a schedulable implementation. Note that, although it may seem that task τ_{12} would fit in-between tasks τ_{23} and τ_{33} in the schedule of N_2 in Fig. 3.2c, τ_{12}, which is SIL 3, cannot use that partition, which is for SIL 1 tasks. Moreover, the particular

partition slice cannot be split, because then it would not fit task τ_{22} in the first major frame.

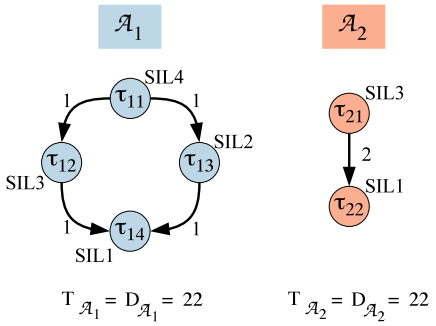
For such situations, we consider the elevation of tasks to allow partition sharing, and we are interested to derive schedulable implementations that minimize the development costs associated to elevation. Thus, in Fig. 3.2d we allow τ_{12} of SIL 3 to share the partition with tasks τ_{22} and τ_{23} of SIL 1, by elevating these two tasks to SIL 3. This will trigger the elevation of the predecessors of τ_{22} and τ_{23}, namely τ_{20} and τ_{21}, to SIL 3. In addition, since τ_{20} and τ_{21} share partitions with tasks τ_{24} and τ_{25}, these will also have to be elevated to SIL 3, leading to a complete elevation of application $\mathcal{A}_2$ from SIL 1 to SIL 3, which, according to the costs from Fig. 2.1c, means an increase in development costs from 85 kEuros to 127 kEuros. The solution in Fig. 3.2d is schedulable, and is optimal in terms of development costs as captured by the cost function from Eq. 3.2 discussed in Section 3.2.2.

Note that, in many application areas, such a development cost increase is preferred to an increase in unit costs. The optimization approach presented in Section 3.2.2 provides to a trade-off analysis tool to the engineer, who can decide what is the best option: to upgrade the platform and increase the unit costs, or to increase the development costs, but keep the same architecture.

3.1.4 Task Decomposition

We have not yet discussed the issue of SIL decomposition. In Section 3.1.3 we have shown how to use elevation to achieve partition sharing, which may lead to increased development costs. Another option is to explore several SIL decompositions for those tasks for which the engineer has specified a SIL decomposition in the decomposition library $\mathcal{L}$ (see Section 2.1.2). Using SIL decomposition will result in more (redundant) tasks of lower SILs. Using several tasks of lower SILs has the advantage of lowering the development costs and may facilitate partition sharing. The disadvantage is the introduction of more tasks, which have to be placed in the schedule table, potentially impairing schedulability.

Let us illustrate these issues using the example in Fig. 3.3. We have two applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, presented in Fig. 3.3a. The two applications are scheduled using SCS. The WCETs for the tasks in the two applications are shown in Fig. 3.3b, while Fig. 3.3c shows the development costs. Let us assume the engineer is considering decomposing task τ_{11} into two options D_1 and D_2 as discussed in Section 2.1.2 and shown in Fig. 3.4a. Fig. 3.4b and Fig. 3.4c present the WCETs and development costs of the tasks resulted from the decomposition. The T_{MF} is set to 11 time units, and T_{cycle} is 22.



(a) Two applications

	$\mathcal{A}_1$				$\mathcal{A}_2$	
	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{21}	τ_{22}
N_1	5	5	4	3	5	3
N_2	4	7	6	4	6	4

(b) WCETs

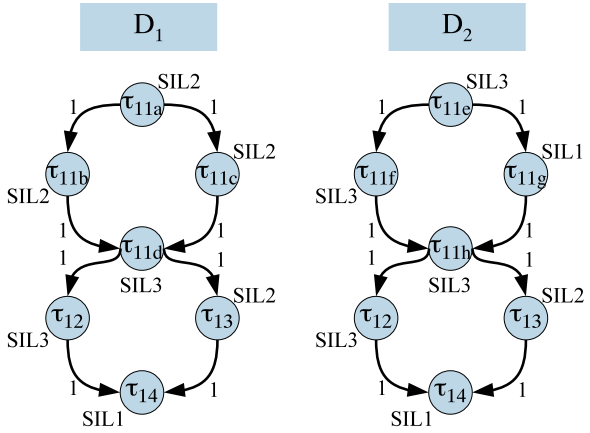
	$\mathcal{A}_1$				$\mathcal{A}_2$	
	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{21}	τ_{22}
SIL 1	x	x	x	10	x	9
SIL 2	x	x	18	20	x	16
SIL 3	x	34	37	37	32	31
SIL 4	60	70	70	70	68	62

(c) Development costs (kEuro)

Figure 3.3: Application model example for SIL decomposition

We first show a solution to this example without considering partition sharing and SIL decomposition. Thus, Fig. 3.5a presents the optimal mapping, partitioning and schedules of tasks, as obtained by running the simultaneous mapping and partitioning optimization discussed in Section 3.1.2. In this case, τ_{22} does not fit into the schedule. Although $\mathcal{A}_2$ has two partition slices on N_1, namely p^1_{12} and p^2_{12}, with a total time of 6 time units out of the 11 time units of the MF, τ_{22}, which is of SIL 1, is allowed to execute only in p^2_{12}, since it cannot share the partition slice p^1_{12} with τ_{21} of SIL 3. Fig. 3.5b shows a solution where we allow partition sharing, but not SIL decomposition. In this case, a schedulable solution was obtained by elevating τ_{22} to SIL 3 to allow τ_{21} and τ_{22} to share the same partition slice, namely p^1_{12}. Due to the elevation of τ_{22}, the development cost of this solution increased to 194 kEuros, compared to 172 kEuros, if all the tasks would have been implemented and certified according to their lowest possible SIL.

We show in Fig. 3.5c the solution when we use SIL decomposition alongside with partition sharing. In Fig. 3.5c we decompose τ_{11} of SIL 4 into two tasks, of SIL 3 and SIL 1, respectively, as specified by the decomposition option D_2, see Fig. 3.4a. Decomposing τ_{11} in this manner increases the cost from 194 kEuros, corresponding to the solution in Fig. 3.5b, to 242 kEuros. This is because to obtain a schedulable solution task τ_{11g} is elevated from SIL 1 to SIL 3 and τ_{13} from SIL 2 to SIL 3 to share the partition slice p^1_{21} with the other tasks of SIL 3. Similarly, τ_{14} is elevated to SIL 3 to share p^1_{11} with τ_{12}. Clearly, this decomposition does not help our design, as it significantly increases the costs. Hence, not all decompositions are improving the design. Fig. 3.5d presents a solution where we use the SIL decomposition specified by D_1, Fig. 3.4a. Thus, τ_{11} of SIL 4 is decomposed into two tasks of SIL 2, namely τ_{11b} and τ_{11c}. Similar to Fig. 3.5b, task τ_{22} is elevated to SIL 3 to share the partition slice



(a) Library $\mathcal{L}$ with two decompositions

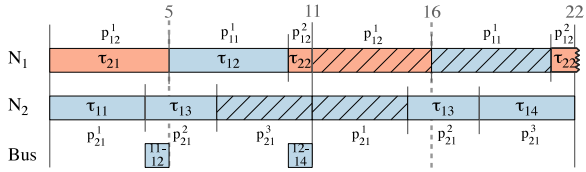
	τ_{11a}	τ_{11b}	τ_{11c}	τ_{11d}	τ_{11e}	τ_{11f}	τ_{11g}	τ_{11h}
N_1	1	3	3	2	1	5	1	2
N_2	1	2	2	2	1	4	1	2

(b) WCETs for the tasks resulted from the SIL decomposition

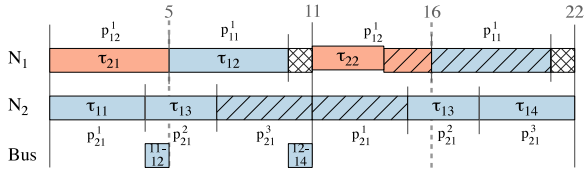
	τ_{11a}	τ_{11b}	τ_{11c}	τ_{11d}	τ_{11e}	τ_{11f}	τ_{11g}	τ_{11h}
SIL 1	x	x	x	x	x	x	9	x
SIL 2	1	14	14	x	x	x	18	x
SIL 3	1	30	29	5	1	33	32	5
SIL 4	1	60	60	10	1	60	60	10

(c) Developments costs for the tasks resulted from SIL decomposition

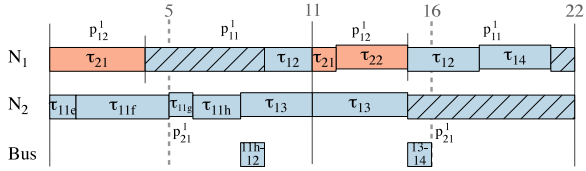
Figure 3.4: Example decomposition for task τ_{11}



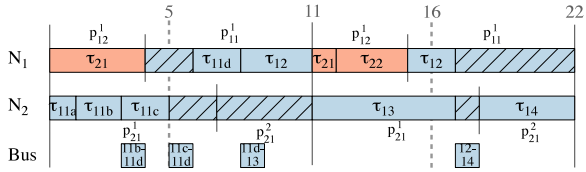
(a) No sharing or decomposition, τ_{22} does not fit into the schedule



(b) By allowing partition sharing, we obtain a schedulable solution



(c) If not selected carefully, SIL decomposition may increase costs



(d) Using an optimized SIL decomposition can lower the development costs

Figure 3.5: SIL decomposition optimization example

p_{12}^1 with τ_{21}. Decomposing in this manner reduces the cost to 160 kEuros, while also ensuring that all deadlines are satisfied.

This example shows that, in order to reduce costs and obtain schedulable solutions, it is important to optimize the SIL decomposition.

3.2 Design Optimization Strategies

Next we propose two optimization strategies for the design problems presented in Section 3.1. The problem of scheduling tasks on multiprocessors is known to be NP-complete [171]. Thus, we propose metaheuristics-based optimization strategies to solve these problems. In Section 3.2.2 we propose a Tabu Search-based (TS) strategy for the problems presented in Sections design optimization problem presented in Section 3.1. However, we first start by presenting in Section 3.2.1 a Simulated Annealing-based (SA) strategy for the problem of time-partition optimization, presented in Section 3.1.1.

3.2.1 Optimization of Time-Partitions

The problem presented in the Section 3.1.1 is NP-complete [171]. Its complexity depends not only on the number of tasks and processors, but also on the number of partition slices on each processor. In order to solve this problem, we will use the optimization strategy Time-Partitioning Optimization (TPO) from Fig. 3.6. TPO takes as input a set of applications Γ, the set of processing elements $\mathcal{N}$ and the mapping of tasks to processors M, and returns the implementation Ψ consisting of the set of partitions slices $\mathcal{P}$ on each processor and the schedules $\mathcal{S}$ for the SC applications. As a reminder, the problem in Section 3.1.1 considers two criticality levels, safety-critical and non-critical, with the SC applications scheduled using SCS and NC tasks scheduling FPS. Moreover, it considers the mapping of tasks to processors as given and fixed. The TPO strategy has 3 steps:

(1) in the first step, we determine an initial set of partition slices $\mathcal{P}^\circ$, line 1 in Fig. 3.6. $\mathcal{P}^\circ$ consists of a simple straight forward partitioning scheme which allocates for each application $\mathcal{A}_j$ a total time on PE N_i proportional to the utilization of the tasks of $\mathcal{A}_j$ mapped to N_i. The partitions P_{ij} thus allocated have the same length and they are distributed with a period equal to the smallest period of a task from $\mathcal{A}_j$ mapped to N_i.

(2) In the second step, we use a Simulated Annealing metaheuristic to determine the set of partition slices $\mathcal{P}$ such that the applications are schedulable and the unused partition

TPO(Γ, $\mathcal{N}$, M)
1 $\mathcal{P}^{\circ}$ = InitialSolution(Γ, $\mathcal{N}$, M)
2 $\mathcal{P}$ = SimulatedAnnealing(Γ, $\mathcal{N}$, M, $\mathcal{P}^{\circ}$)
3 $\mathcal{S}$ = ListScheduling(Γ, $\mathcal{N}$, M, $\mathcal{P}$)
4 **return** $\Psi =< \mathcal{P}, \mathcal{S} >$

Figure 3.6: Time-Partition Optimization

space (potentially used for future upgrades) is maximized. The alternatives provided by Simulated Annealing are evaluated using the cost function from Eq. 3.1.

(3) Finally, given the optimized partitions $\mathcal{P}$ obtained in line 2 in Fig. 3.6, we use a List Scheduling heuristic (presented in Section 3.4) to determine the schedule tables for the SC applications.

Simulated Annealing (SA) is an optimization metaheuristic that tries to minimize the cost function in order to find the global optimum by randomly selecting neighboring solutions of the current solution [22]. The algorithm presented in Fig. 3.7 takes as input the set of application Γ, the architecture $\mathcal{N}$, the mapping M and the initial partitioning $\mathcal{P}^{\circ}$, and returns the best solution found $\mathcal{P}^{best}$, i.e., with the smallest cost function (see line 8 in Fig. 3.7). In order to escape local minima, worse solutions are sometimes accepted with a probability depending on a control parameter called temperature and on the deterioration of the cost function (see lines 10 to 13 in Fig. 3.7). Before we call SA we merge all NC tasks into a single application, since the NC tasks are allowed to share a partition.

The algorithm is inspired by the process of annealing metals, a thermal process in which a metal is heated past its melting point and then carefully cooled down so that the particles arrange themselves with lower internal energy than the initial solution [22]. The cooling rate of the process influences the quality of the result. The cooling schedule of SA is defined by the initial temperature TI, the temperature length TL, the cooling ratio ε and the stopping criterion. The temperature length TL and the cooling ratio ε decide how fast will the temperature drop. We use a time limit as a stopping criterion (line 17).

Cost Function. We have defined our cost function as follows:

$$Cost_{TPO}(\mathcal{P}) = w_{SC} \times \delta_{SC} + w_{NC} \times \delta_{NC} \tag{3.1}$$

where δ_{SC} is the degree of schedulability for SC applications (scheduled using SCS) and δ_{NC} is the degree of schedulability for NC applications (scheduled using FPS). The degree of schedulability is presented in Section 3.3.

SimulatedAnnealing($\Gamma, \mathcal{N}, M, \mathcal{P}^\circ$)

```
1  temperature = initial temperature TI
2  P^now = P^best = P°
3  repeat
4    for i = 1 to temperature length TL do
5      generate a random neighbor solution P' of P^now
6      delta = Cost_TPO(P') - Cost_TPO(P^now)
7      if delta < 0 then
8        P^now = P^best = P'
9      else
10       generate q = random (0, 1)
11       if q < e^(-delta/temperature) then
12         P^now = P'
13       end if
14     end if
15   end for
16   temperature = ε × temperature
17 until stopping criterion is met
18 return P^best
```

Figure 3.7: The Simulated Annealing algorithm

These are summed together into a single value using the weights w_{SC} and w_{NC}. In case an application is not schedulable, its corresponding weight is a very big number, i.e., a "penalty" value. This allows us to explore unfeasible solutions (which correspond to unschedulable applications) in the hope of driving the search towards a feasible region. In case an application $\mathcal{A}_i$ is schedulable, we use for the weight a value given by the engineer, depending on the importance of the application. For example, in our experiments we have used weights for the SC application which are several times greater than those for the NC applications.

Design Transformations. The neighboring solutions of the current solution $\mathcal{P}^{now}$ are generated using design transformations (or "moves") applied to $\mathcal{P}^{now}$. There are 4 types of moves: *resize*, *swap*, *join* and *split*. The moves are applied to a randomly selected partition slice from a randomly chosen PE.

The *resize* move, as its name implies, resizes the selected partition slice. This is done either by increasing the size of the partition slice at the expense of a neighboring partition slice, or by decreasing it and giving the extra space to a neighboring slice. The amount with which the partition can be resized is randomly chosen, but we have imposed an upper limit (half the size of the partition). The *swap* move swaps the chosen partition slice with another randomly chosen partition slice. The *join* move joins two

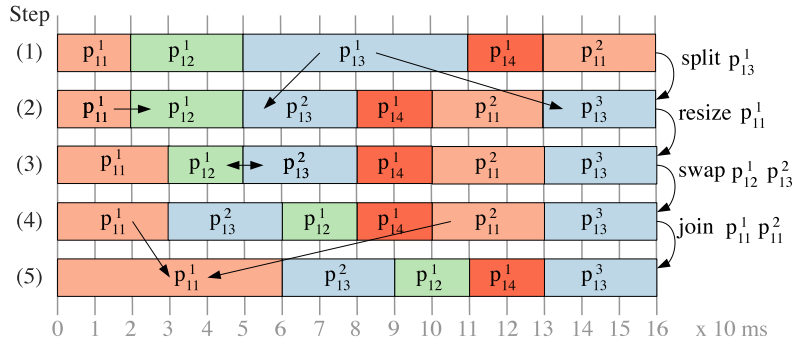


Figure 3.8: Partition slice move examples

partition slices belonging to the same application, while the *split* move splits a partition slice into two, and adds the second slice to the end of the MF. Together with the 4 types of basic moves, we also apply "improved moves". An "improved move" is intended to accelerate the search by performing several basic moves at once.

Fig. 3.8 depicts the basic moves as they are sequentially performed on a single PE, namely N_1. As mentioned, the notation p_{ij}^k means the k^{th} partition slice of the application $\mathcal{A}_j$ on the processing element N_i. There are 4 applications, numbered from 1 to 4, and the first application has 2 partition slices, p_{11}^1 and p_{11}^2. The current solution $\mathcal{P}^{now}$ is presented in Step 1 in Fig. 3.8. The first move is the *split* move, which is performed on the partition slice p_{13}^1 belonging to $\mathcal{A}_3$. The slice is split in two equal parts, and the resulting slice is added to the end of the MF. The second move is a *resize* with 10 ms, which affects p_{11}^1 at the expense of p_{12}^1. The third move is a *swap* of slices p_{12}^1 and p_{13}^2. The result is shown in the 4^{th} step. The last move is a *join* move and as previously mentioned, it can be applied only to partition slices belonging to the same application. For this move we chose the p_{11}^1 and p_{11}^2 slices. Once a move has been performed on the partition set $\mathcal{P}^{now}$, the resulted partition set $\mathcal{P}'$ is evaluated using the cost function from (3.1), which is executed using List Scheduling and Response Time Analysis, presented in Section 3.4 and Section 3.5, respectively.

3.2.2 Tabu Search-Based Design Optimization

The problem of scheduling tasks on multiprocessors is known to be NP-complete [171], while the problem of mapping tasks onto a multiprocessor system is proved to be NP-hard [37]. In order to solve the problem presented in Section 3.1, we will use the "Mixed-Criticality Design Optimization" (MCDO) strategy from Fig. 3.9, which is based on a Tabu Search metaheuristic. MCDO takes as input a set of applications Γ

MCDO(Γ, $\mathcal{N}$, $\mathcal{L}$)
1 $< D^\circ, M^\circ, \mathcal{P}^\circ, \phi^\circ >$ = InitialSolution(Γ, $\mathcal{N}$)
2 $< D, M, \mathcal{P}, \phi >$ = TabuSearch(Γ, $\mathcal{N}$, $\mathcal{L}$, D°, M°, $\mathcal{P}^\circ$, ϕ°)
3 $\mathcal{S}$ = ListScheduling(Γ, $\mathcal{N}$, D, M, $\mathcal{P}$, ϕ)
4 **return** $\Psi =< D, M, \mathcal{P}, \phi, \mathcal{S} >$

Figure 3.9: Mixed-Criticality Design Optimization strategy

(including the SIL information, development costs DC and the separation requirements graph Π), the SIL decomposition library $\mathcal{L}$ and the set of processing elements $\mathcal{N}$, and returns the implementation Ψ consisting of the SIL decomposition D, the mapping M of tasks to PEs, the set of partitions slices $\mathcal{P}$ on each PE, the assignment ϕ of tasks to partitions and the schedules $\mathcal{S}$ for the applications. Our strategy has 3 steps:

(1) In the first step, we consider that tasks are not decomposed (denoted with D°) and we determine an initial task mapping M°, an initial set of partition slices $\mathcal{P}^\circ$ and an initial assignment of tasks to partitions ϕ°, line 1 in Fig. 3.9. The initial mapping M° is done such that the utilization of processors is balanced and the communication on the bus is minimized. $\mathcal{P}^\circ$ consists of a simple straightforward partitioning scheme which allocates for each application $\mathcal{A}_j$ a total time on PE N_i proportional to the utilization of the tasks of $\mathcal{A}_j$ mapped to N_i. The initial assignment ϕ° of tasks to partitions consists of a separate partition for each SIL level in each application, and does not allow partition sharing.

(2) In the second step, we use a Tabu Search metaheuristic to determine the SIL decomposition D, the task mapping M, the set of partition slices $\mathcal{P}$ and the assignment of tasks ϕ to partitions, such that the applications are schedulable and the development costs are minimized.

(3) Finally, given the SIL decomposition D, the task mapping M, the optimized partitions $\mathcal{P}$ and the assignment ϕ of tasks to partitions obtained in line 2 in Fig. 3.9, we use a List Scheduling heuristic (see Section 3.4) to determine the schedule tables $\mathcal{S}$ for the applications.

Tabu Search (TS) [83] is a metaheuristic optimization, which searches for that solution which minimizes the *cost function* (see Eq. 3.2 for our cost function definition). Tabu Search takes as input the set of applications Γ, the set of PEs $\mathcal{N}$, the decomposition library $\mathcal{L}$ and the initial solution, consisting of D°, M°, $\mathcal{P}^\circ$, and ϕ°, and returns at the output the best solution found during the design space exploration, in terms of the cost function.

Tabu Search explores the design space by using design transformations (or "moves") applied to the current solution in order to generate neighboring solutions. To escape lo-

TabuSearch(Γ, $\mathcal{N}$, $\mathcal{L}$, D°, M°, $\mathcal{P}^\circ$, ϕ°)

```
1  Best ← Current ← < D°, M°, P°, φ° >
2  L ← {}
3  while termination condition not reached do
4     remove tabu with the oldest tenure from L if Size(L) = l
5     // generate a subset of neighbors of the current solution
6     C ← GenerateCandidateList(Current, Γ, N)
7     Next ← solution from C that minimizes the cost function
8     if Cost_MCDO(Next) < Cost_MCDO(Best) then
9        // accept Next as Current solution if better than the best-so-far Best
10       Best ← Current ← Next
11       add Next to L
12    else if Cost_MCDO(Next) < Cost_MCDO(Current) and Next ∉ L then
13       // also accept Next as Current solution if better than Current and not tabu
14       Current ← Next
15       add Next to L
16    end if
17    if diversification needed then
18       Current ← Diversify(Current)
19       empty L
20    end if
21    if restart needed then
22       Current ← Best
23       empty L
24    end if
25 end while
26 return Best
```

Figure 3.10: The Tabu Search algorithm

cal minima, TS incorporates an adaptive memory (called "tabu list" or "tabu history"), to prevent the search from revisiting previous solutions, thus avoiding cycling. The size of the tabu list, that is, the number of solutions marked as tabu, is called *tabu tenure*. In case there is no improvement in finding a better solution for a number of iterations, TS uses *diversification*, i.e., visiting previously unexplored regions of the search space. In case the search diversification is unsuccessful, TS will *restart* the search from the best known solution.

Fig. 3.10 presents the Tabu Search algorithm. Line 1 initializes the *Current* and *Best* solutions to the initial solution formed by the tuple $< D^\circ, M^\circ, \mathcal{P}^\circ, \phi^\circ >$. Line 2 initializes the tabu list L to an empty list. The size l of L, i.e., its *tenure*, is set by the user. The Tabu Search algorithm runs until the termination condition is not reached (see line 3). This termination condition can be, for example, a certain number of iterations

or a number of iterations without improvement, considering the cost function [82]. Our implementation stops the search after a predetermined amount of time, set by the user. In case the tabu list L is filled, we remove the oldest tabu from this list (see line 4).

Since it is infeasible to evaluate all the neighboring solutions (see the discussion in this Section, "Candidate List"), we generate a subset of neighbors of the *Current* solution (line 6), called *Candidate List* and we choose from this *Candidate List*, as the possible *Next* solution, the one that minimizes the cost function (line 7). We accept a solution as the *Current* solution from which the exploration continues if: (1) if it has a cost which is better than the best-so-far solution *Best*, lines 8–11 in Fig. 3.10, or (2) if it has a better cost then the *Current* solution and it is not "tabu", lines 12–16. The *Best* and *Current* solutions are updated accordingly, lines 10 and 14, respectively, and the *Next* solution is added to the tabu list L, lines 11 and 15. Note that in the first case we can also accept tabu solutions, which is referred to as "aspiration". In this situation, the already tabu solution in L will be moved to the tail of the list, thus setting its tenure to the size l of the list.

In case the algorithm does not manage to improve the current solution after a number of iterations, it proceeds to a diversification stage (lines 17–20). During this stage, we attempt to drive the search towards an unexplored region of the design space. Thus, in the Diversify function call, we randomly decompose tasks that have decomposition options specified in the library $\mathcal{L}$, and we randomly re-assign a task from each application, while keeping the same partition tables. If after a preset number of diversification stages, the algorithm is still unable to improve the solution, we restart the search from the best known solution so far (lines 21–24). After a diversification or restart occurs, the tabu list L is emptied.

Cost Function. For each alternative solution visited by TS we use the List Scheduling-based heuristic from Section 3.4 to produce the schedule tables $\mathcal{S}$. We define the response time R_i of an application $\mathcal{A}_i$ as the time difference between the finishing time of the sink node and the start time of the application. $DC(\Gamma)$ is the development cost of the set Γ of all applications (see Section 2.1.3). We define the cost function of an implementation ψ as:

$$Cost_{MCDO}(\psi) = \begin{cases} c_1 = \sum_{\mathcal{A}_i \in \Gamma} \max(0, R_i - D_i) & if\, c_1 > 0 \\ c_2 = DC(\Gamma) & if\, c_1 = 0 \end{cases} \tag{3.2}$$

This cost function is modified from the degree of schedulability presented in Section 3.3. If at least one application is not schedulable, there exists one R_i greater than the deadline D_i, and therefore the term c_1 will be positive. However if all the applications are schedulable, this means that each R_i is smaller than D_i, and the term $c_1 = 0$. In this case, we use c_2 as the cost function, since when the applications are schedulable, we are interested to minimize the development cost.

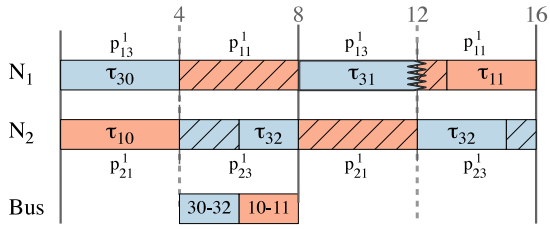
Design Transformations. As previously mentioned, the exploration of the design space is done by applying design transformations (moves) to the current solution *Current*. We use one *re-assignment* move, which changes the assignment of a task to another partition and four types of moves applied to partition slices: *resize*, *swap*, *join* and *split*. The partition slice moves are the same as the one used by the TPO algorithm presented in Section 3.2.1. They are presented in Fig. 3.8 and explained in detail in Section 3.2.1. We also employ SIL decomposition moves, namely *decompose* and *recompose*.

The task *re-assignment* move re-assigns a task to another partition. The partition can be an existing one, or newly created. The partition may be on another PE, thus, implicitly, the re-assignment move will also re-map the task. The re-assignment move respects the separation requirements graph Π, but does not prevent partition sharing by tasks of different SILs. In case the move will lead to sharing, we elevate tasks as required, and update the development costs accordingly. If a re-assignment move results in an empty partition, the partition is deleted and its assigned CPU time is distributed to a randomly chosen partition. As a result, the algorithm creates and deletes partitions and partition slices, as well as resizes them, on the fly as needed, depending on the task re-assignment moves.

The SIL decomposition moves are applied to the tasks which have decomposition alternatives specified in the library $\mathcal{L}$. The *decompose* move selects a random decomposition option from the library. The *recompose* move is applied to a task τ_i, and it reverts the task to its initially proposed model, thus undoing any *decompose* moves that may have affected τ_i. These moves are applied during the diversification phase (line 18 in Fig. 3.10) to randomly selected tasks.

Our algorithm relies on a *tabu list* with tabu-active attributes, that is, it does not remember complete solutions in the list L, but rather attributes of the moves that generated the tabu solutions. In case of the resize and the swap moves, tabu-active attributes are the involved partition slices. For the split and join moves, the tabu-active attribute is the partition the move was performed on. As for the re-assignment move, the attributes are the re-assigned task and the involved partitions.

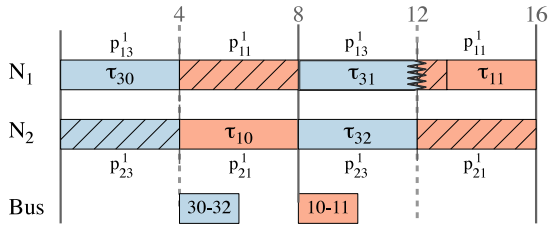
Let us illustrate in Fig. 3.11 how Tabu Search works. We consider applications $\mathcal{A}_1$, with tasks τ_{10} and τ_{11}, and $\mathcal{A}_3$, with tasks τ_{30}–τ_{33}, from Fig. 2.1, with their periods and deadlines equal to 16. The size of the major frame T_{MF} is set to 8 and T_{cycle} is 16. We are interested to implement these applications on an architecture with two PEs, N_1 and N_2. Let us assume that we are running our TS and the current solution, which is also the best-so-far solution, is the one presented in in Fig. 3.11a. The mapping and assignment of tasks in this solution is as follows. $\tau_{10} \in \mathcal{A}_1$ is assigned to partition $\mathcal{P}_{2,1}$ on N_2 (composed of partition slice p^1_{21}), while τ_{11} is assigned to partition $\mathcal{P}_{1,1}$ on N_1 (with partition slice p^1_{11}). In the case of application $\mathcal{A}_3$, τ_{30} and τ_{31} are assigned to partition $\mathcal{P}_{1,3}$ on N_1 (of slice p^1_{13}), while τ_{32} and τ_{33} are assigned to $\mathcal{P}_{2,3}$ on N_2 (with slice p^1_{23}).



Tenure	Tabu
5	*resize* p^1_{21} and p^1_{23} on N_2
4	*join* p^1_{11} and p^2_{11} from $\mathcal{P}_{1,1}$ on N_1
3	*resize* p^1_{11} and p^1_{13} on N_1
2	*swap* p^1_{11} and p^2_{11} on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 7

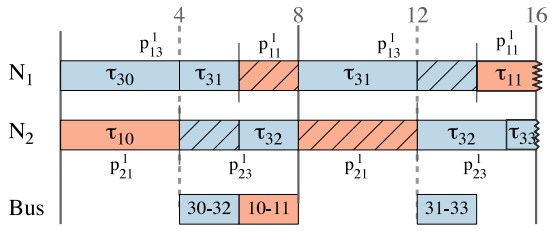
(a) Current solution



Tenure	Tabu
5	*resize* p^1_{21} and p^1_{23} on N_2
4	*join* p^1_{11} and p^2_{11} from $\mathcal{P}_{1,1}$ on N_1
3	*resize* p^1_{11} and p^1_{13} on N_1
2	*swap* p^1_{11} and p^2_{11} on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 7

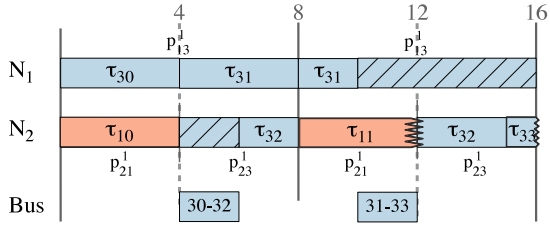
(b) Swap the partitions on N_2, results in a solution which is not better than the current solution



Tenure	Tabu
5	*resize* p^1_{11} and p^1_{13} on N_1
4	*resize* p^1_{21} and p^1_{23} on N_2
3	*join* p^1_{11} and p^2_{11} from $\mathcal{P}_{1,1}$ on N_1
2	*swap* p^1_{11} and p^2_{11} on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 4

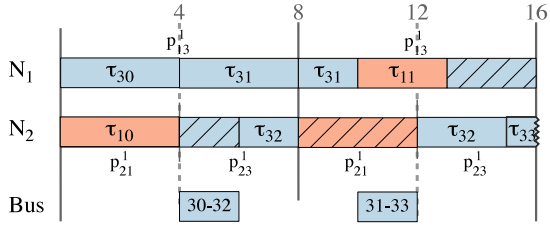
(c) Resize τ_{31}'s partition. Best solution so far, although it is a tabu move



Tenure	Tabu
5	*resize* p^1_{21} and p^1_{23} on N_2
4	*join* p^1_{11} and p^2_{11} from $\mathcal{P}_{1,1}$ on N_1
3	*resize* p^1_{11} and p^1_{13} on N_1
2	*swap* p^1_{11} and p^2_{11} on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 4

(d) Re-assign τ_{11} to N_2. Tabu move and does not improve the solution



Tenure	Tabu
5	*re-assign* τ_{11} *from* $\mathcal{P}_{1,1}$ *to* $\mathcal{P}_{1,3}$
4	*resize* p^1_{21} *and* p^1_{23} *on* N_2
3	*join* p^1_{11} *and* p^2_{11} *from* $\mathcal{P}_{1,1}$ *on* N_1
2	*resize* p^1_{11} and p^1_{13} on N_1
1	*swap* p^1_{11} and p^2_{11} on N_1

Cost = 3

(e) Re-assign τ_{11} to $\mathcal{A}_3$'s partition on N_1. Best solution so far

Figure 3.11: Moves and tabu history

Note that this solution is not schedulable, since tasks τ_{31} and τ_{33} from $\mathcal{A}_3$ do not fit into the schedule. Each of the figures from Fig. 3.11b–3.11e presents a neighboring solution generated from the current solution in Fig. 3.11a, and are intended to illustrate moves performed by TS and how the tabu list is updated. None of these solutions are schedulable, but we can see improvements in the cost function, which will drive the search to a schedulable solution.

Next to each solution we present the value of the cost function associated to the solution. Since none of these solution are schedulable, the value of the cost function is the term c_1 from Eq. 3.2. Furthermore, we also present for each solution the updated tabu list (referred to as L in Fig. 3.10). Fig. 3.11a presents the current tabu list. Fig. 3.11b–3.11e present the updated list, that will be used in case the associated solution is chosen as the as the *Next* solution (see Fig. 3.10). For this example, the tabu tenure l is 5. The tabu most recently added to the list has the highest tenure, while the oldest tabu in the list has the lowest tenure. For example, in Fig. 3.11a, the most recently added tabu to the list has a tenure of 5. This tabu is associated to the move that generated this solution, namely a *resize* move, and the involved partition slices are p^1_{21} and p^1_{23}.

Fig. 3.11b presents a neighboring solution obtained from Fig. 3.11a by swapping on N_2 the partition slices p^1_{21} and p^1_{23} assigned to $\mathcal{A}_1$ and $\mathcal{A}_3$, respectively. This move does not improve the solution, i.e., the value of the cost function is 7 in both cases, and thus is ignored. Since this solution is ignored, the tabu list is not updated. Fig. 3.11c shows the solution obtained from Fig. 3.11a obtained by resizing the partition slice p^1_{13} on N_1. In this solution, p^1_{13} is increased, while the size of p^1_{11} is decreased. This solution was generated by a move that is tabu. Because this solution is better than the best-so-far solution shown in Fig. 3.11a, in terms of the cost function (the value of the cost function is 4 in the new solution, compared to 7 in Fig. 3.11a) the tabu status of the move is ignored. The updated tabu list, in case the search will continue with this solution as the *Current* solution, is presented next to the solution.

The solution in Fig. 3.11d is obtained by re-assigning task τ_{11} from partition $\mathcal{P}_{1,1}$ on N_1 (composed of partition slice p^1_{11}) in Fig. 3.11a to $\mathcal{P}_{2,1}$ on N_2 (of slice p^1_{21}). After re-assigning τ_{11}, partition $\mathcal{P}_{1,1}$ on N_1, composed of p^1_{11}, has no tasks assigned to it, therefore it is deleted and the algorithm "transfers" the CPU time of $\mathcal{P}_{1,1}$ to $\mathcal{P}_{1,3}$ (composed of p^1_{13}). Thus, on N_1, there is only one partition slice executing, i.e., p^1_{13}. Although this move does improve the solution presented in Fig. 3.11a in terms of the cost function, it is not better than the solution from Fig. 3.11c, and hence, it is ignored. Fig. 3.11e presents a solution obtained by re-assigning τ_{11} from partition $\mathcal{P}_{1,1}$ on N_1 in Fig. 3.11a to $\mathcal{P}_{1,3}$. Similar to the solution from Fig. 3.11d, since partition $\mathcal{P}_{1,1}$ has no tasks assigned to it, it is deleted and its CPU time given to $\mathcal{P}_{1,3}$. Furthermore, τ_{11} shares the partition with τ_{30} and τ_{31}. Since τ_{11} is a task with SIL 3, and tasks τ_{30} and τ_{31} are SIL 2 tasks, the two tasks from $\mathcal{A}_3$ have to be elevated to SIL 3, thus increasing the development costs of the system. This move does not result in a schedulable solution, but it improves the solution in terms of cost function. The value of the cost function

in this case is 3, and is better than the other neighboring solutions. The search will continue with this solution as the *Current* solution, and the tabu list will be accordingly updated.

Candidate List. The neighborhood of the *Current* solution is composed of all the solutions which are "one move away", that is, obtained by applying a move to the *Current* solution. To decide which move to select as the *Next* solution, we need to determine which of the neighbors minimizes the cost function (line 7 in Fig. 3.10). Calculating the cost function (Eq. 3.2) means determining the schedule tables for all the applications (term c_1 in Eq. 3.2) and, if they are schedulable, a sumation of the development costs for all tasks (term c_2). Since the size of a neighborhood is large, calculating the cost function for every neighbor would take a very long time, rendering the search process infeasible. Instead, only a part of the neighborhood is considered, and neighbors are placed on a so called *candidate list* $\mathcal{C}$. One option is to select randomly the neighbors to be placed on the candidate list. However, we use a heuristic approach that selects those neighbors which have a higher chance to lead quickly to good quality solutions.

The candidate list generation algorithm is presented in Fig. 3.12. The algorithm takes as input the *Current* solution, the set of applications Γ and the set of processing elements $\mathcal{N}$, and returns a list $\mathcal{C}$ of candidate solutions. The algorithm is called on line 6 in our Tabu Search from Fig. 3.10.

The algorithm starts with an empty candidate list $\mathcal{C}$ (line 1 in Fig. 3.12). The neighbors placed in $\mathcal{C}$ are obtained by performing moves on the *Current* solution. The following moves are considered. On each PE, the algorithm performs partition related moves (resize, swap, join and split moves) on random partition slices (lines 3–5). On each PE, the algorithm chooses as a candidate the most oversized partition, that is, the partition with the lowest ratio of used CPU time compared to actual allocated time (line 6), and resizes (shrinks) this partition. This is done to transfer "unused" CPU time from an oversized partition to another partition, in the hope of improving the overall schedulability of the system. Similarly, there might exist situations where we have undersized partitions, that is, partitions that have more tasks assigned than allocated CPU time, or partitions where the allocated time is not enough for all the assigned tasks to execute before their deadline. For such situations, on each PE, the algorithm selects the most undersized partition, i.e., the partition with the highest ratio of required CPU time compared to the actual allocated time, and resizes this partition, increasing its size (line 7).

Such an example is given in Fig. 3.11c, obtained from Fig. 3.11a. The most undersized partition in Fig 3.11a, on PE N_1 is partition $\mathcal{P}_{1,3}$ containing partition slice p^1_{13}. This partition has an allocated time of 8 time units during the MF, and has assigned to it tasks τ_{30} and τ_{31}, requiring 10 time units for execution. Thus, it has a ratio of required to allocated CPU time of 125%. The other partition on N_1, $\mathcal{P}_{1,1}$, has only τ_{11} assigned to

GenerateCandidateList($Current, \Gamma, \mathcal{N}$)

1 $\mathcal{C} \leftarrow \{\}$
2 **for** $N_i \in \mathcal{N}$ **do**
3 **for all** move $\in$ {resize, swap, join, split} **do**
4 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ apply *move* to a random partition slice on N_j in *Current*$\}$
5 **end for**
6 $\mathcal{C} \leftarrow \mathcal{C} \cup \{ New | New \leftarrow$ *resize* most oversized partition on N_j in *Current*$\}$
7 $\mathcal{C} \leftarrow \mathcal{C} \cup \{ New | New \leftarrow$ *resize* most undersized partition on N_j in *Current*$\}$
8 **end for**
9 **if** perform moves on tasks **then**
10 **for all** applications $\mathcal{A}_i$ that missed their deadlines **do**
11 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task $\tau_j \in \mathcal{A}_i$ to random partition in *Current*$\}$
12 **end for**
13 **for all** $PE_i \in \mathcal{N}$ **do**
14 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from N_i to a random partition on $N_k \neq PE_i$ in *Current*$\}$
15 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from N_i to another partition on N_i in *Current*$\}$
16 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from N_i to another application's partition in *Current*$\}$
17 **end for**
18 **end if**
19 **return** $\mathcal{C}$

Figure 3.12: Algorithm to generate the candidate list $\mathcal{C}$

it, which requires only 3 time units for execution, out of the 8 allocated to the partition. The ratio of required to allocated CPU time for this partition is only 37.5%. Hence, the most undersized partition, i.e., $\mathcal{P}_{1,3}$, is increased, by transferring CPU time from partition slice p^1_{11} to p^1_{13}. The candidate solution generated by this move is presented in Fig. 3.11c, and is better than the solution shown in Fig. 3.11a, in terms of the cost function.

The diversification stage presented in Fig. 3.10, line 18, randomly re-assigns a task from each application, while keeping the same partition tables. Furthermore, during this phase, randomly selected tasks that have decomposition options specified in the decomposition library $\mathcal{L}$, are decomposed. The introduction of the decomposed tasks may decrease the schedulability of the diversified solution. In order to allow TS to improve on the schedulability by adapting the partition table to the new mapping scheme and to the decomposed tasks, we do not allow any re-assignment moves for a certain number of iterations. This condition is shown in line 9, in Fig. 3.12. Thus, we force the

TS to explore this new design space area, while keeping the assignment of tasks to partitions fixed. When this restriction is lifted, the algorithm focuses on the applications that miss their deadlines, in order to make them schedulable (lines 10–12 in Fig. 3.12). For this, the algorithm selects a random task from each application missing its deadline, and re-assigns it to another partition. Furthermore, on each PE we perform three types of re-assign moves (lines 13–17), in hope to thoroughly explore the design space. The algorithm re-assigns a random task τ_i to another PE, but to the same application's partition (line 14); a random task to the same PE, but to another partition (line 15); and another task to another PE, to another application's partition (line 16).

3.3 Degree of Schedulability

The degree of schedulability, defined in [131], is calculated as:

$$\delta = \begin{cases} c_1 = \sum_i \max(0, R_i - D_i) & \text{if } c_1 > 0 \\ c_2 = \sum_i (R_i - D_i) & \text{if } c_1 = 0 \end{cases} \tag{3.3}$$

For applications scheduled using SCS, δ is computed at application level, and thus R_i is the response time of the application (i.e., the finishing time of the sink node) as resulted from List Scheduling (see Section 3.4), while D_i is the deadline of the application. For tasks scheduled using FPS, δ is computed at task level. Thus, R_i is the worst-case response time and D_i is the deadline of each task. The response time for each task is computed according to the response time analysis presented in Section 3.5.

If at least one FPS task or SCS application is not schedulable, there exists one R_i greater than the deadline D_i, and therefore the term c_1 will be positive. However if all the tasks (applications) are schedulable, this means that each R_i is smaller than D_i, and the term $c_1 = 0$. In this case, we use c_2 as the degree of schedulability, since it can distinguish between two schedulable solutions [131].

3.4 List Scheduling

Given a partition set $\mathcal{P}$, we use a List Scheduling (LS)-based heuristic to determine the schedule tables $\mathcal{S}$ for each application scheduled using SCS. LS heuristics use a sorted priority list, L_{ready}, containing the tasks ready to be scheduled. A task τ_i is *ready* if all the predecessor tasks have finished executing and all the incoming messages are received. We use the Modified Partial Critical Path priority function [132] to sort L_{ready}.

We define the response time R_i of an application $\mathcal{A}_i$ as the time difference between the finishing time of the sink node and the start time of the application. Thus, LS is applied to each SCS application. We have modified the classical LS algorithm to take into account the partitions. Thus, when scheduling a task, we are allowed to use only the corresponding partitions slices from $\mathcal{P}$. If a partition slice finishes before a task has completed its execution (as is the case with $\tau_{31} \in \mathcal{A}_3$ in Fig. 3.2b), we assume that the task is suspended and will continue its execution in the next partition where is assigned. Our LS implementation takes into account the partition switching overhead t_O. The suspension of the task will take place online, based on the partition scheme $\mathcal{P}$ loaded into the kernel and t_O captures the time needed to do a context switch to another partition. LS also derives the schedules tables for the messages on the bus.

3.5 Response Time Analysis

To determine the schedulability of FPS applications we use a response-time analysis [56] to calculate the worst-case response time R_i of every FPS task τ_i, which is compared to its deadline D_i. The basic analysis presented in [56] has been extended over the years. For example, the state-of-the-art analysis from [123] considers arbitrary arrival times and deadlines, offsets and synchronous inter-task communication (where a receiving task has to wait for the input of the sender task). Audsley and Wellings [30] have proposed a schedulability analysis for FPS tasks using temporal partitioning (IMA), which, when analyzing a FPS task in a certain partition, considers the other time-partitions as higher priority tasks. This analysis assumes that the deadlines are smaller or equal to the periods, that the tasks are independent, and that the start times of partition slices within a major frame are periodic. Pop et al. [136] have proposed a schedulability analysis for ET tasks, which extends the schedulability analysis with static and dynamic offsets in [123] to consider the influence of the TT tasks on the worst-case response times of the ET tasks.

In this thesis, we have extended the analysis from [136] to consider the influence of time-partitions on the schedulability of the FPS tasks. In [136], the authors introduce the notion of *ET demand* and *ET availability*, used to compute the length of the busy window w_i, which is needed in order to compute the worst-case response time R_i of a task τ_i. The busy window w_i is the longest time interval during which tasks of priority equal or greater than τ_i are continuously executing [78]. The worst-case response time is determined using Eq. 3.4, considering a certain length of the busy window w_i, and all the higher priority tasks:

$$R_i = w_i - \varphi_i - (p-1) \times T_i + \phi_i \tag{3.4}$$

where p is the number of activations of task τ_i in the busy window w_i, T_i is the period of τ_i, the offset ϕ_i is the earliest activation of τ_i relative to the occurrence of the triggering

event and the phase φ_i is the time interval between the critical instant and earliest time of the first activation of τ_i. The worst-case response time R_i for the task τ_i is the maximum value of the result in Eq. 3.4, considering all the critical instants initiated by higher priority tasks and by τ_i and also all the job instances. During the calculation, if the *available* time does not satisfy the *demand* of τ_i then the algorithm increases iteratively the length of the busy window w_i which is analyzed [136].

Similar to the notion of *ET demand* from [136], we introduce *FPS demand*, associated with a FPS task τ_i on a time interval t, as the maximum amount of CPU time which can be *demanded* by higher or equal priority FPS tasks and by τ_i. Fig. 3.13 shows the analysis for a task τ_i, considering the busy window that starts at the critical instant $qT_i + t_c$, initiated by task τ_a and ends at the moment $qT_i + t_c + w_i$, when all the higher priority tasks (τ_a and τ_b) and τ_i itself have finished execution, and when all the partition slices interrupting τ_i have finished.

During the busy window w_i, the *demand* H_i associated with task τ_i scheduled in a partition P_k is equal with the length of the busy window that would result when considering that P_k would be the only partition on the processor. Thus, similar to [136] and [123], the *FPS demand* is:

$$H_i(w_i) = B_i + (p - p_{0,i} + 1) \times C_i + W_i(\tau_i, w_i) + \sum_{\forall(a \in \mathcal{A}_a \neq \mathcal{A}_i)} W_a^*(\tau_i, w_i) \tag{3.5}$$

where B_i is the maximum blocking time for τ_i. The job activations of task τ_i during w_i are denoted with p and positive values are assigned to instances arriving after t_c, while zero and negative values indicate the instance arrived before t_c. Thus, $p_{0,i}$ is the index of the first pending instance of τ_i and is computed as follows:

$$p_{0,i} = 1 - n_{ia} = 1 - \left\lfloor \frac{J_i + \varphi_i}{T_i} \right\rfloor, \tag{3.6}$$

where n_{ia} is the number of pending τ_i jobs at t_c, during the busy window w_i initiated by τ_a.

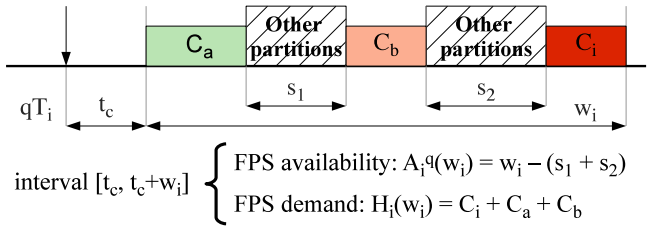


Figure 3.13: Availability and demand

$W_i(\tau_i, w_i)$ is the interference from higher priority tasks hp(τ_i) in the same application $\mathcal{A}_i$ as τ_i:

$$W_i(\tau_i, w_i) = \sum_{j \in hp(\tau_i)} \left(\left\lfloor \frac{J_j + \varphi_i}{T_i} \right\rfloor + \left\lceil \frac{w_i - \varphi_i}{T_i} \right\rceil \right) \times C_j \tag{3.7}$$

and $W_a^*(\tau_i, w_i)$ is the worst-case interference from higher priority $hp_a(\tau_i)$ tasks from other applications than the application $\mathcal{A}_i$ that τ_i belongs to:

$$W_a^*(\tau_i, w_i) = max(W_k(\tau_i, w_i)), \forall k \in hp_a(\tau_i) \tag{3.8}$$

Fig. 3.13 shows that the *FPS demand* of task τ_i during the busy window w_i is the sum of the worst case execution times of the higher priority FPS tasks, C_a and C_b, and the worst-case execution time C_i of τ_i.

We extend the concept of *availability* from [136] as the processing time available during w_i for P_k. Considering we are using time partitions and that task τ_i can execute only during its own partition P_k, the *availability* is computed by subtracting from w_i the time reserved for the "other partitions". In Fig. 3.13 the other partitions are illustrated with hashed rectangles and their duration denoted with s_1 and s_2.

3.6 Experimental Results

We have implemented the algorithms presented in Section 3.2 in Java (JDK 1.6), running on SunFire v440 computers with UltraSPARC IIIi CPUs at 1.062 GHz and 8 GB of RAM. We have evaluated our algorithms using synthetic and real-life benchmarks. We will first present the evaluation of the TPO algorithm, described in Section 3.2.1, and then the evaluation of MCDO, described in Section 3.2.2.

3.6.1 Optimization of Time-Partition

The problem of the optimization of time partitions addresses was presented in Section 3.1.1, and the TPO algorithm was described in Section 3.2.1. As a reminder, TPO assumes that the mapping of tasks is given. Moreover, we do not consider the issue of cost. Therefore, TPO does not address partition sharing or task decomposition. TPO considers that the safety-critical applications are scheduled using SCS, while the non-critical tasks are scheduled using FPS.

For the evaluation of our proposed algorithm we used 10 synthetic benchmarks and 2 real life case studies. In the first set of experiments we were interested to evaluate

the proposed TPO strategy in terms of its ability to find schedulable implementations. Thus, we have used 5 synthetic benchmarks with 3 to 5 SC applications (with a total of 15 to 53 SC tasks). All the NC tasks have been merged into a single NC application, with 5 to 9 tasks. The resulted mixed-criticality system has been mapped on architectures ranging from 2 to 6 processing elements. The mapping has been done such that the utilization on the PEs is balanced and the communication over the bus is minimized. The execution times and message lengths were assigned randomly within the 1 to 19 ms and 1 to 5 bytes ranges, respectively. The weights used for computing the cost function were $w_{SC} = 400$ for SC applications and $w_{NC} = 100$ for NC tasks (see Section 3.2.1).

We have used two time limits for the experiments: 10 minutes and 120 minutes. The results obtained with TPO using a time limit of 120 minutes are presented in Table 3.1, under the heading "TPO, 120 min. time limit".

We were interested to determine the quality of our SA-based TPO strategy. Hence, we have used an exhaustive search to determine the optimal solutions. Since the runtime of the exhaustive search is prohibitively large, we were only able to run it for smaller examples, benchmarks labelled "1.1", "2.1" and "2.2" in Table 3.1. In these cases, our SA-based approach is capable of obtaining (in 120 minutes) solutions which are very close to the optimum. For the benchmarks labelled "1.1", "2.1" and "2.2" the difference in term of the cost function is only 4.51%, 0.16% and 1.9%, respectively.

Together with TPO, Table 3.1 also presents the results obtained using a Straightforward Solution (SS), which implements the approach from the InitialSolution function presented in Section 3.2.1. SS is an approach that a good engineer would use if TPO would not be available. Columns 3 and 5 in Table 3.1 present the number of SC applications, and the NC tasks, respectively. The number of schedulable applications and tasks (out of the total) obtained by our proposed TPO strategy are presented in columns 9 and 10, respectively, while columns 7 and 8 present the results obtained using SS. Columns 11 and 12 represent the percentage increase in the degree of schedulability for SC applications, Δ_{SC}, and NC tasks, Δ_{NC}, (see Section 3.3) as obtained by the TPO strategy compared to the SS, considering a time limit of 120 minutes. A negative value for Δ_{NC} means that our optimization has decreased the degree of schedulability for the NC tasks in order to guarantee that all SC applications are schedulable. Note that the NC tasks are still schedulable in this case, but their response times have increased, compared to SS, which over-dimensioned the NC partitions. Column 13 represents the average of the percentage increase in the degree of schedulability for the whole system.

We have also run TPO with a time limit of 10 min. TPO is able to obtain schedulable solutions in all cases, except for the case study in line 4 in Table I. The average deviation of the percentage increase of the cost function (as captured by Eq. 3.1) for the schedulable results, compared to the results obtained with TPO using a 120 min. time limit, is of 10.48%.

Table 3.1: TPO experimental results

Set	Benchmark	SC		NC	PEs	SS		TPO, 120 min. time limit				
		Apps	Tasks	Tasks		Sched. SC Apps	Sched. NC Tasks	Sched. SC Apps	Sched. NC Tasks	Δ_{SC}	Δ_{NC}	avg. % increase in δ
1	1.1	3	15	5	2	1 of 3	All	All	All	1709.76	-44.00	832.88
	1.2	3	20	6	3	1 of 3	All	All	All	107.94	-53.23	27.36
	1.3	4	34	6	4	None	All	All	All	169.68	7.14	88.41
	1.4	4	40	10	5	None	All	All	All	147.54	-0.40	73.57
	1.5	5	53	9	6	3 of 5	All	All	All	542.78	14.66	278.72
2	2.1	1	6	6	4	All	All	All	All	78.38	0.00	39.19
	2.2	2	12	6	4	All	All	All	All	59.20	-2.87	28.17
	2.3	3	20	6	4	None	5 of 6	All	All	518.06	1453.85	985.96
	2.4	4	30	6	4	1 of 4	All	All	All	211.66	0.00	105.83
	2.5	5	34	6	4	2 of 5	5 of 6	All	All	466.36	673.33	569.85
3	auto	3	19	5	3	None	All	All	All	227.33	0.57	113.95
	telecom	4	19	6	3	All	All	All	All	135.29	-11.56	61.87

As we can see from "Set 1", SS which does not perform optimization, is not able to find schedulable implementations. For example, for the largest benchmark, with 5 SC application and 9 NC tasks mapped on 6 PEs only 3 out of 5 SC applications are schedulable. All the NC tasks are schedulable. Note that SS leads to schedulable NC implementations. This is because it distributes the partition slices to match the smallest period of the tasks. However, since the slices have equal lengths, there is a lot of wasted space in the schedules of SC applications, which leads to missed SC deadlines. However, by applying our proposed TPO approach, we are able to optimize the time partitions such that all applications are schedulable. We have measured the ability of TPO to improve over SS by using a percentage average increase in the degree of schedulability over all applications, presented in the last column. As we can see there is a dramatic increase in the degree of schedulability over all applications, when using TPO. This means that we can potentially implement the applications on a slower (cheaper) architecture.

In the second set of applications, labeled "Set 2", we were interested to see how TPO performs as the utilization of the system increases. We have mapped 2 to 6 applications on the same architecture of 4 PEs. As we can see, TPO is able to find schedulable implementations even as the utilization increases.

Finally, we have also used 2 real life benchmarks derived from the Embedded Systems Synthesis Benchmarks Suite (E3S) version 0.9 [70]. We have used the *auto-indust-cowls* and *telecom-mocsyn* benchmarks, labelled as "auto" and "telecom" in Table 3.1. In the case of the "auto", the first 3 applications are considered SC, while the last one is NC. In the case of "telecom" test case, the applications numbered as 0, 1, 2 and 4 were used as SC, and applications numbered as 3, 5, 6 and 7 were merged into one NC application. In both cases the applications are mapped on an architecture of 3 PEs. The results obtained from these real-life benchmarks confirm the results of the synthetic benchmarks.

3.6.2 Mixed-Criticality Design Optimization

Next we evaluate our proposed "Mixed-Criticality Design Optimization" (MCDO) strategy described in Section 3.2.2, which addresses the complete set of optimization problems presented in Section 3.1. To remind the assumptions of MCDO, the mapping of tasks to processors and the partitioning are not given, but have to be decided by MCDO. Furthermore, MCDO performs partition sharing and task decomposition. For simplicity, we consider all applications scheduled using SCS, but MCDO can be easily extended to consider FPS tasks.

For the evaluation of our MCDO algorithm we used 7 synthetic benchmarks and 2 real-life case studies. In the first set of experiments we were interested to evaluate the

proposed MCDO in terms of its ability to find schedulable implementations. Thus, we have used 3 synthetic benchmarks with 3 to 5 mixed-criticality applications (with a total of 15 to 41 tasks). We have used MCDO to implement these applications on architectures with 2 to 5 processing elements. The execution times and message lengths were assigned randomly within the 1 to 19 ms and 1 to 5 bytes ranges, respectively. The details of each benchmark, namely the benchmark number, the number of applications, the number of tasks and the number of processing elements are presented in Table 3.2, columns 2–5. For all the experiments, we used the decompositions in Table 2.1.

We were interested to compare the number of schedulable implementations found by MCDO with two other setups. In one of the setup, SIL decomposition is not used and the sharing of partitions by tasks of different criticality levels is not allowed, but mapping and partitioning optimization is performed simultaneously. Let us call this simultaneous "mapping and partitioning optimization", MPO. In the other setup, sharing and decomposition are not allowed, and in addition, mapping optimization (MO) is performed separately from partitioning optimization (PO). We call such an approach MO+PO.

MO+PO and MPO are based on the MCDO strategy presented in Fig. 3.9, and use the same Tabu Search for the optimization. The difference is in the types of moves performed by TS: there are only mapping moves for MO (without considering partitions), we use only partition-related moves in PO, considering mapping fixed, as determined by MO. In addition, MPO does not allow decomposition moves and re-assignment moves that would lead to partition sharing by mixed-criticality tasks. Further, MO, PO and MPO use as cost function the "degree of schedulability" (see Section 3.3).

The termination condition of our Tabu Search is a time limit given by the engineer. Very long runs were performed with MCDO and the best solution obtained was considered the near-optimal solution. Then, we have set the time limit for the experiments such that this near-optimal solution is found and the time is as small as possible. The time imposed for each individual experiment is 480 minutes.

The results for the first set of experiments are presented in Table 3.2 in the rows corresponding to "Set 1". The number of schedulable applications, resulted after implementing the system using MO+PO, MPO and MCDO are reported in columns 6, 7 and 9, respectively, labelled with the respective acronym and "Sched. apps.".

As we can see from the comparison between MO+PO and MPO, there is a significant improvement in the number of schedulable applications if the optimization of mapping is considered at the same time with the optimization of partitioning. For example, for the second benchmark (benchmark "1.2" in Set 1) with 4 applications mapped to 4 PEs, MO+PO is unable to successfully schedule any of the applications. MPO, which performs mapping and time optimization simultaneously, is able to schedule 3 out 4 applications. We have also compared MPO and MO+PO in terms of the degree of

Table 3.2: Comparison of MO+PO, MPO and MCDO experimental results

Set	Benchmark	Apps	Tasks	PE	MO+PO	MPO		MCDO	
					Sched. Apps	Sched. Apps	δ_{Sched} (%)	Sched. Apps	δ_{DC} (kEuro)
1	1.1	3	15	2	2	2	450	All	59
	1.2	4	34	4	0	3	3600	All	19
	1.3	5	41	5	3	All	235	–	–
2	2.1	3	20	4	All	All	1.10	–	–
	2.2	4	30	4	All	All	23.96	–	–
	2.3	5	34	4	4	All	13.27	–	–
	2.4	6	39	4	3	5	208.11	All	470
3	consumer	2	12	3	0	1	343.45	All	123
	networking	4	13	3	2	2	31.78	All	40

schedulability (see Section 3.3). The percentage improvement δ_{sched} of MPO over MO+PO is presented in column 8. An improvement in the degree of schedulability means that there is more slack available in the schedule, which can be used for future upgrades, for example.

If MPO produces a schedulable solution, i.e., the applications are schedulable without SIL decomposition or partition sharing, we do not have to run MCDO. This is indicated in the table using a dash "–" in the MCDO columns. However, MPO is not able to find schedulable implementations in the first two cases. In such situations, MCDO, which optimizes the SIL decompositions and the partition sharing at the same time with mapping and partitioning, can find schedulable implementations in all cases.

Once a schedulable implementation is found by using decomposition and elevation, the cost function from Eq. 3.2 will drive MCDO to solutions that minimize the development cost. Elevation for partition sharing will increase the development costs, whereas SIL decomposition has the potential to reduce these costs. The increase δ_{DC} in development cost that we have to pay in order to find schedulable implementations, compared to MPO which does not perform SIL elevation or decomposition, is reported in the last column of Table 3.2.

In the second set of experiments, labeled "Set 2" in Table 3.2, we were interested to see how MCDO performs compared to MO+PO and MPO as the utilization of the system increases. Thus, we have an increasing number of mixed-criticality applications, from 3 to 6 and we have used the same architecture of 4 PEs. As we can see, for the smaller benchmarks of 3 and 4 applications (benchmarks "2.1" and "2.2", respectively), MO+PO is able to find schedulable implementations. Optimizing the mapping and time partitions using MPO leads to more schedulable implementations, i.e., "All" applications are schedulable in benchmark "2.3". However, as the system utilization

increases, as is the case for the largest benchmark in this set ("4.4"), where we used 6 applications on 4 PEs, only MCDO, which considers decomposition and elevation to allow partition sharing by tasks of mixed-criticality, is able to find schedulable solutions. Therefore, MCDO is able to integrate successfully more mixed-criticality applications on the same integrated architecture, thus saving product unit costs by avoiding costly architecture upgrades across the product line.

Finally, we have also used 2 real life benchmarks derived from the Embedded Systems Synthesis Benchmarks Suite [70] version 0.9. We have used the *consumer-cords* and *networking-cords* benchmarks. In both cases we were interested to implement the applications to an architecture of 3 PEs. The results obtained from these real-life benchmarks are reported in the last 2 lines in Table 3.2 and confirm the results of the synthetic benchmarks.

CHAPTER 4

Design Optimizations at the Network-Level

The previous chapter has presented design optimizations for mixed criticality applications implemented on distributed heterogeneous architectures. We have assumed a simple statically scheduled bus for the communication. In this chapter we address mixed-criticality applications in the context of a realistic communication protocol, namely TTEthernet. The TTEthernet protocol and the models used at the communication level have been presented in Sections 2.2.3 and 2.2.2, respectively.

4.1 Problem Formulation

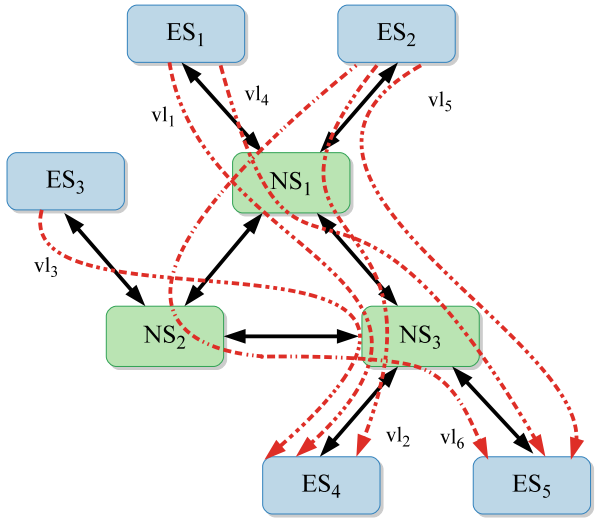
The problem we are addressing in this chapter can be formulated as follows: given (1) the topology $\mathcal{G}_C$ of the TTEthernet cluster, (2) the set of TT and RC messages $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$ and (3) for each message m_i the size, deadline, period / rate, SIL and the source and destination End Systems, we are interested to determine an optimized implementation Υ such that the deadlines for the TT and RC frames are satisfied. Determining an implementation means deciding on the (i) fragmenting Φ_m of messages and packing $\mathcal{K}$ of messages and messages fragments into frames, (ii) the assignment $\mathcal{M}_F$ of frames to virtual links, (iii) the routing $\mathcal{R}_{VL}$ of virtual links, (iv) the bandwidth for each RC virtual link and (v) the set of TT schedule tables $\mathcal{S}$.

Once both TT and RC frames are schedulable several optimization objectives can be tackled. In this chapter we are interested to optimize the network configuration such that all frames are schedulable and the end-to-end delay of RC frames is minimized. Section 4.2.1 presents the cost function used for the optimization. We ignore the BE traffic in this chapter, but a quality-of-service measure for the BE traffic could easily be added to the objective function, as we will show in Chapter 5. In this chapter we are not concerned with scheduling redundant message delivery for fault-tolerance, since TTEthernet networks can be physically replicated. The schedules we derive for TT messages are used for all the replicated channels. The design optimization problems addressed in this chapter are illustrated in the next subsections using several motivational examples.

4.1.1 Straightforward Solution

Let us illustrate the design optimization problem using the setup from Fig. 4.1, where we have a cluster composed of five end systems, ES_1 to ES_5 and three network switches NS_1 to NS_3 (see Fig. 4.1a) and an application with five TT messages, m_1 to m_5, and two RC messages, m_6 and m_7, see the table in Fig. 4.1b. The periods $m_i.period$ and deadlines $m_i.deadline$ of each message m_i are given in the table. For simplicity, in this example we assume all messages to have the same SIL. Although the standard TTEthernet speed is 100 Mbps or higher, for the sake of this example we consider a link speed of only 2 Mbps, and that all the dataflow links have the same speed. In Fig. 4.1b we also specify the source and destination for each message. For simplicity, we considered one destination for each message. The table also contains the transmission times C_i for each message m_i in our setup, considering for the moment that each message is packed into its own frame. We take into account the total overhead of the protocol for one frame (67 B for each frame).

Our problem is to determine the (i) message fragmenting and packing, (ii) the assignment of frames to virtual links, (iii) the routing of virtual links, (iv) the bandwidth for each RC virtual link and (v) the TT schedules $\mathcal{S}$ such that all the TT and RC frames are schedulable. The schedulability of a TT frame f_i is easy to determine: we just have to check the schedules $\mathcal{S}$ to see if the times are such that the TT frame f_i is received before its deadline $f_i.deadline$. To determine the schedulability of an RC frame f_j we have to compute its Worst-Case end-to-end Delay (WCD), from the moment it is sent to the moment it is received. We denote this worst-case delay with R_{f_j}. In [169], we have presented a schedulability analysis technique to determine the WCD of an RC frame. By comparing R_{f_j} with the deadline $f_j.deadline$, we can determine if an RC frame f_j is schedulable. For this example we consider that the RC and TT traffic are integrated using a "timely block" policy (see Section 2.2.3), i.e., an RC frame will be delayed if it could block a scheduled TT frame.



(a) Example architecture model

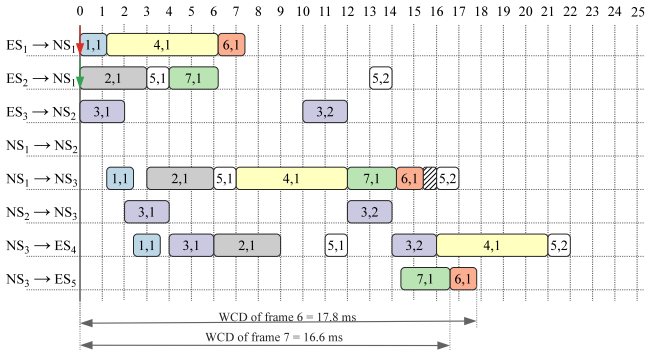
	period (ms)	deadline (ms)	size (B)	C_i (ms)	Source	Dest
$m_1 \in \mathcal{M}^{TT}$	40	40	233	1.2	ES_1	ES_4
$m_2 \in \mathcal{M}^{TT}$	40	40	683	3	ES_2	ES_4
$m_3 \in \mathcal{M}^{TT}$	10	10	433	2	ES_3	ES_4
$m_4 \in \mathcal{M}^{TT}$	40	40	1183	5	ES_1	ES_4
$m_5 \in \mathcal{M}^{TT}$	10	10	183	1	ES_2	ES_4
$m_6 \in \mathcal{M}^{RC}$	40	32	233	1.2	ES_1	ES_5
$m_7 \in \mathcal{M}^{RC}$	20	16	483	2.2	ES_2	ES_5

(b) Example application model

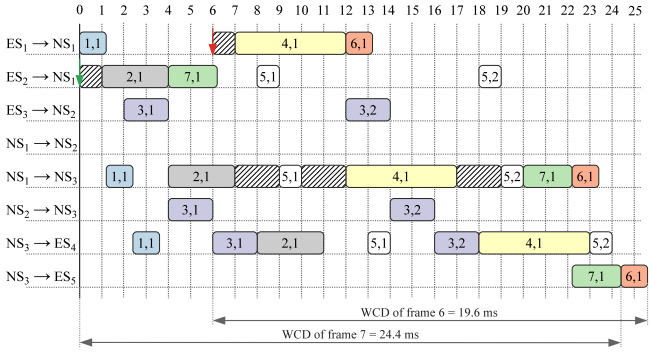
Figure 4.1: Example system model

A Straightforward Solution (SS) to our optimization problem is to (i) pack each message into its own frame and (ii) assign this frame to a virtual link, (iii) route each virtual link on the shortest paths from the frame source to its destinations, (iv) set the bandwidth for each RC virtual link to the minimum required for the respective RC frame rate, and (v) schedule the TT frames using As-Soon-As-Possible (ASAP) scheduling. Such a straightforward solution would be chosen by a good engineer without the help of our optimization tool. For the example in Fig. 4.1, this solution is depicted in Fig. 4.2a. Let us discuss it in more detail.

(i) *Fragmenting*, *packing*: SS does not fragment messages, and packs each message m_i into a frame f_i, with f_i inheriting the size, period and deadline of m_i. (ii) Frame *assignment* and (iii) VL *routing*. We assign each frame f_i to a virtual link vl_i and route the VL along the shortest path in the physical topology. The resulted VLs are vl_1 to vl_5 and they are depicted with dot dash red arrows in Fig. 4.1a. (iv) Each RC VL carrying



(a) Straighforward Solution, each message assigned to one frame, routed along the shortest path, and scheduled ASAP, results in f_7 missing its deadline in the worst-case scenario



(b) Alternative baseline solution, using SS approach for message packing and frame routing, but a different TT schedule. In this case f_7 misses its deadline in the worst-case scenario

Figure 4.2: Baseline solutions

an RC frame has an associated *bandwidth* parameter called BAG, from Bandwidth Allocation Gap (see Section 2.2.3). BAG(vl_i) is the minimum time interval between two consecutive instances of an RC frame f_i on VL vl_i. SS will set the BAG in such a way to guarantee the rate of the frame f_i, while respecting the protocol constraints on BAG sizes (see Section 2.2.3.2). Thus, the BAG(vl_i) for each VL vl_i of an RC frame is chosen as the largest value 2^i, i=0..7, not greater than the minimum inter-arrival time. If this minimum inter-arrival time is greater than 128 ms, the BAG is set to 128 ms. Thus, the BAG for f_6 is 32 ms, while for f_7 is 16 ms.

(v) *Scheduling* of TT frames. As mentioned, SS uses ASAP scheduling to derive the TT frame schedules. Fig. 4.2a presents these schedules for our example. Instead of presenting the actual schedule tables, we show a Gantt chart, which shows on a timeline from 0 to 25 ms what happens on the eight dataflow links of interest, $[ES_1, NS_1]$, $[ES_2, NS_1]$, $[ES_3, NS_2]$, $[NS_1, NS_2]$, $[NS_1, NS_3]$, $[NS_2, NS_3]$, $[NS_3, ES_4]$ and $[NS_3, ES_5]$. For the TT frames f_1 through f_5, the Gantt chart captures their sending times (the left edge of the rectangle) and transmission duration (the length of the rectangle). A periodic frame f_i has several frame instances. We denote with $f_{i,x}$ the x^{th} instance of f_i. In the Gantt chart, for readability, the rectangles associated to each frame $f_{i,j}$ are labelled only with *i,j*. We can see in Fig. 4.2a that all the TT frames are schedulable (they are received before their deadlines).

Since the transmission of RC frames is not synchronized with the TT frames, there are many scenarios that can be depicted for the RC frames f_6 and f_7, depending on when the frames are sent in relation to the schedule tables. Because we are interested in the schedulability of the RC frames f_6 and f_7, we show in the Gantt charts their worst-case scenario, i.e., the situation which has generated the largest (worst-case) end-to-end delay for these frames. Thus, in Fig. 4.2a, the worst-case end-to-end delay (WCD) of the RC frame f_6, namely $f_{6,1}$, is 18.6 ms, smaller than its deadline of 32 ms, and hence, it is schedulable. For f_7 though, the WCD is 17.4 ms, larger than its deadline of 16 ms, thus frame f_7 is not schedulable. This worst-case for f_7 happens for the frame instance $f_{7,1}$, see Fig. 4.2a, when $f_{7,1}$ is ready for transmission by ES_2 at 0 ms, depicted with a downward pointing green arrow. The worst-case arrival time for f_6, which leads to the largest WCD R_{f_6}, is depicted with a downward pointing red arrow. In this case, as the network implements the *timely block* integration algorithm, the frame f_7 cannot be sent if its transmission interferes with the TT schedule. Thus, $f_{7,1}$ cannot be sent by ES_2 until the TT frame $f_{2,1}$ finishes transmitting and it cannot be forwarded by NS_1 to NS_3 until $f_{4,1}$ is completely relayed by NS_1.

Let us illustrate the optimizations that can be performed to reduce the WCD of RC frames, and thus make frame f_7 schedulable. In order to show all the optimizations that can be performed, we propose to use Fig. 4.2b as the alternative initial solution. The solution presented in Fig. 4.2b is built using the SS approach of packing messages into frames and routing the frames, but has an alternative schedule table. In this case, the TT frames are schedulable, and the WCD for the RC frames are 19.6 ms for f_6, and

24.4 ms for f_7. Thus f_7 misses its deadline, leading to an unschedulable solution. As the network implements the *timely block* integration algorithm, the frame $f_{7,1}$ cannot be sent until there is a big enough time interval to transmit the frame without disturbing the scheduled TT frames. We denote these "blocked" time intervals with hatched boxes. The first big enough interval on dataflow link [NS_1, NS_3] starts only at time 20 ms, right after $f_{5,2}$ is received by NS_3, which is too late to meet f_7's deadline.

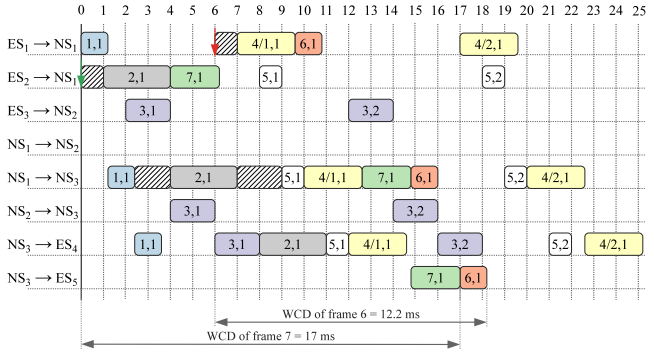
4.1.2 Message Fragmenting and Packing

Let us perform the following modification to the solution from Fig. 4.2b. We *fragment* the largest message on [NS_1, NS_3], message m_4, into two frames, namely $f_{4/1}$ and $f_{4/2}$. The two new frames will have each a payload of 592 B, and a transmission time of 2.63 ms. The schedules for the new solution are shown in Fig. 4.3a. The period of both frames is 40 ms. By fragmenting message m_4, we increase the available time interval for transmission on [NS_1, NS_3] between frames $f_{4/1,1}$ and $f_{5,1}$. Thus, in this solution, the worst-case scenario for the RC frames is an end-to-end delay of 12.2 ms for f_6 and 17 ms for f_7. Although we reduced the WCD of f_7 from 24.4 ms in Fig. 4.2b to 17 ms in this solution, frame f_7 is still missing its deadline. As it can be seen, the fragmenting of TT messages increases the porosity of the schedule.

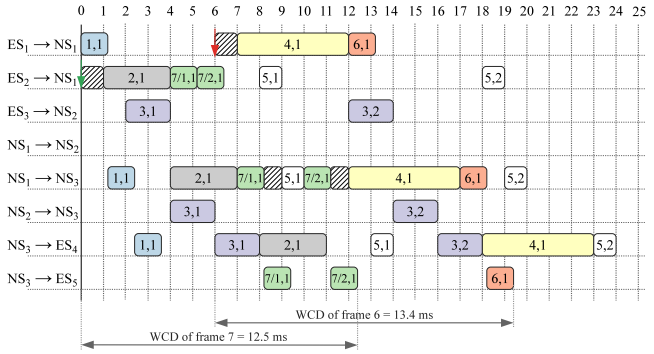
An alternative to the solution in Fig. 4.3a, depicted in Fig. 4.3b, is to *fragment* the RC message m_7 into two frames $f_{7/1}$ and $f_{7/2}$. Thus, we can use the existing empty time slots between the TT frames on dataflow link [NS_1, NS_3]. The new RC frames, with a $C_{7/1,1} = C_{7/2,1} = 1.25$ ms and a BAG of 16 ms, can be transmitted in the available time between $f_{2,1}$–$f_{5,1}$, and $f_{5,1}$–$f_{4,1}$, respectively. This solution reduces the WCD for message m_6 to 13.4 ms and for m_7 to 12.5 ms, thus making all the RC messages schedulable. Fragmenting RC messages allows the RC frames to better use the existing available time slots between the TT frames.

Another alternative is presented in Fig. 4.3c. We pack messages m_1 and m_4 into frame f_{1+4}. The new frame has thus a payload of 1416 B, and a total transmission time of 5.93 ms, with a period and deadline of 40 ms. This will reduce the WCD of f_7 by 5 ms, because it eliminates the blocked time intervals on dataflow link [NS_1, NS_3], see Fig. 4.3c. However, this WCD reduction is not enough to make f_7 schedulable.

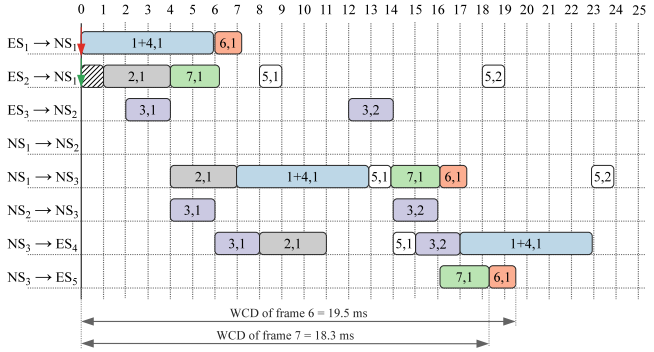
The packing move is especially advantageous to small messages, as it reduces the ratio of protocol overhead to frame payload. Consider 3 RC messages m_{RC1}, m_{RC2} and m_{RC3}, with the same SIL, transmitted from the same source to the same destinations. Messages have a size of 18 B, 10 B and 21 B, respectively, and a deadline of 20 ms, 19 ms and 50 ms. If each message is packed in its own frame, the corresponding frames would have a size of 85 B, 81 B and 88 B, respectively, with a BAG of 16, 16 and 32 ms, respectively. If we pack the three messages into one frame, the new frame would



(a) Fragmenting TT message m_4 into frames $f_{4/1,1}$ and $f_{4/2,1}$ further reduces the WCD of f_7, but it still remains unschedulable



(b) Fragmenting RC message m_7 into two frames, reduces the WCD to 12.5 ms, below its deadline.



(c) Packing two TT messages into one frame, namely m_1 and m_4 into f_{1+4}, reduces the WCD of f_7, but enough to make it schedulable

Figure 4.3: Message packing and fragmenting example

have a size of 116 B with a BAG of 16 ms. Thus, with an increase in the frame size of less than 50% compared to the smallest frame, we can deliver all three messages at once, reducing also the delivery time of the frames. Moreover, the benefits of packing several TT messages into one frame has the advantage of consolidating the available time intervals for RC transmission, between scheduled TT frames, into bigger chunks, as shown in Fig. 4.3c.

The examples in Fig. 4.3 show that by carefully deciding the fragmenting and packing of messages to frames, we can improve the schedulability of messages.

4.1.3 Virtual Link Routing

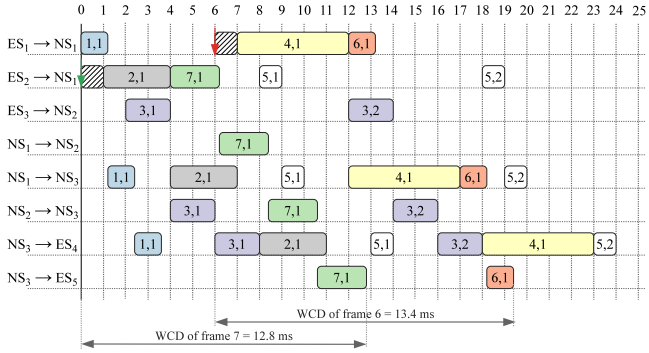
Fig. 4.1a shows for vl_1 to vl_5 the routing of VLs as performed by the Straightforward Solution, which selects the shortest route. Let us assume, however, that we route the RC frame f_7, which goes from ES_2 to ES_5, using vl_6 via NS_2. This is a longer route vl_6 = {[[ES_2, NS_1], [NS_1, NS_2], [NS_2, NS_3], [NS_3, ES_5]]}, compared to the shortest route vl_5={[[ES_2, NS_1], [NS_1, NS_3], [NS_3, ES_5]]}. We consider the same packing and scheduling as in Fig. 4.2b, and we show the WCD of f_7 on this new route in Fig. 4.4a. Thus, the WCD of f_7 has been reduced to 12.8 ms, and that of f_6 to 13.4 ms, which are both schedulable.

Another routing alternative is to keep f_7 on the shortest route, but to route the TT frame f_4 (from ES_1 to ES_4) via the longer route through NS_2 ([[ES_1, NS_1], [NS_1, NS_2], [NS_2, NS_3], [NS_3, ES_4]]), instead of the shortest route ([[ES_1, NS_1], [NS_1, NS_3], [NS_3, ES_4]]). Thus, in Fig. 4.4b we can see we have a WCD of 9.6 ms and 14.4 ms for RC frames f_6 and f_7, respectively, which are schedulable.

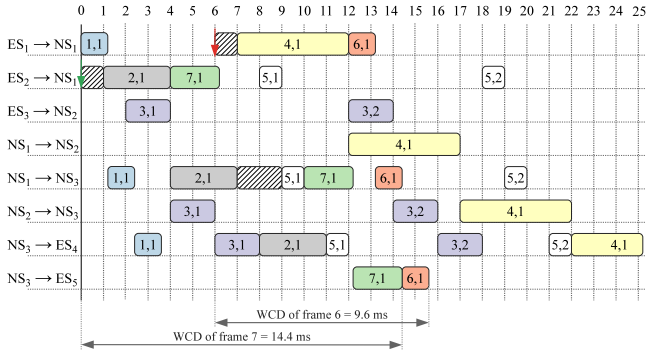
These two examples show that by selecting, counterintuitively, a longer route for a message, we can improve the schedulability.

4.1.4 Scheduling of TT Messages

In [169] we have shown how carefully deciding the schedules for the TT messages can improve schedulability. Compared to [169], which has focused only on scheduling, in this chapter we also address fragmenting, packing and routing. In addition, we also consider realistic scheduling constraints imposed by the current TTEthernet implementations. In [169] we have assumed that the offset of a TT frame instance on a dataflow link can vary across periods. Thus, frame instances of the same TT frame may have different offsets. However, this is not supported by the current TTEthernet implemen-



(a) Rerouting the RC frame f_7 is an alternative to obtain a schedulable solution



(b) Rerouting TT frame f_4 via NS_2 frees up traffic on dataflow link $[NS_1, NS_3]$, reducing the WCD of the RC messages, compared to Fig. 4.2b

Figure 4.4: Message rerouting examples

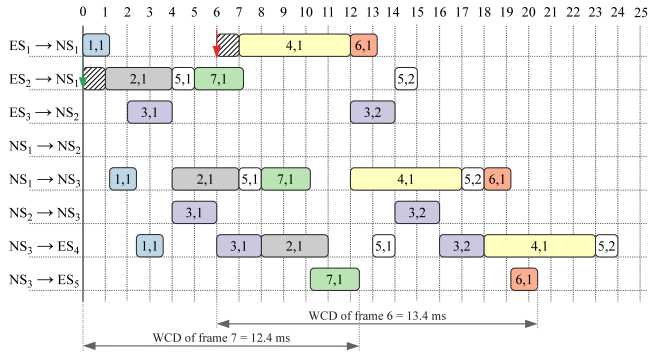


Figure 4.5: Rescheduling frame f_5 to an earlier instant on $[ES_2, NS_1]$ groups the TT frames and eliminates the timely block intervals, resulting in the WCD of the RC messages

tations, and hence in this chapter we impose the scheduling constraint that all the frame instances of a TT frame on a dataflow link should have the same offset in all periods.

Fig. 4.5 presents the impact of rescheduling a TT frame, in the context of the example in Fig. 4.2b. We reschedule the TT frame f_5 for an earlier transmission on $[ES_2, NS_1]$. Although this move increases the worst-case delay for f_7 on that dataflow link, the move groups the TT frames together on the dataflow link $[NS_1, NS_3]$. Consequently, this move eliminates the timely blocked intervals that block the transmission of RC frames, thus reducing the overall WCD for both RC frames.

4.2 Design Optimization Strategy

The scheduling problem presented in Section 4.1 is similar to the flow-shop scheduling problem and is shown to be NP-complete [81], with the packing and fragmenting of frames adding to the complexity of the problem. In order to solve this problem, we propose the "Design Optimization of TTEthernet-based Systems" (DOTTS) strategy from Fig. 4.6, which is based on a Tabu Search metaheuristic. DOTTS takes as input the topology of the network $\mathcal{G}_C$ and the set of TT and RC mesages $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$ (including the size, period/rate and deadline), and returns the best implementation Υ. Such an implementation consists of (i) the fragmenting of messages Φ_m and packing in frames $\mathcal{K}$, (ii) the assignment of frames to virtual links $\mathcal{M}_F$, (iii) the routing $\mathcal{R}_{VL}$ of virtual links, (iv) the bandwidth for each RC virtual link and (v) the schedules $\mathcal{S}$ for the TT frames.

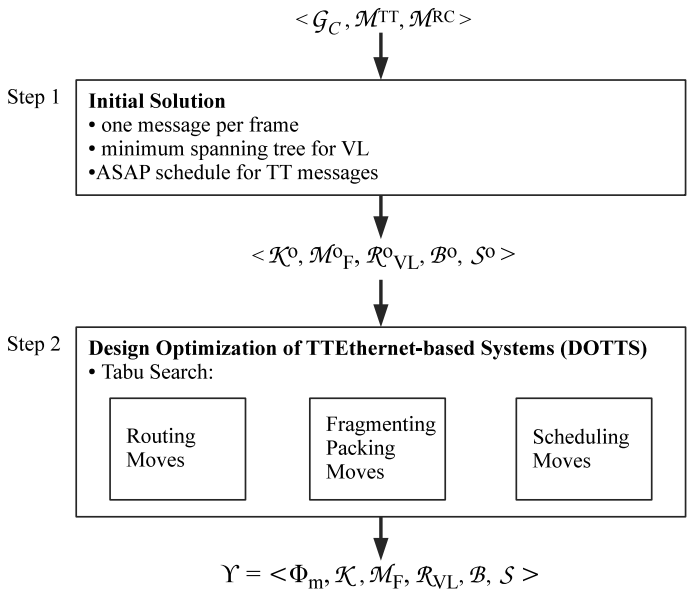


Figure 4.6: Design Optimization of TTEthernet-based Systems

Our strategy has 2 steps, see the two boxes in Fig. 4.6: (1) In the first step we determine an initial solution using the straightforward approach introduced in Section 4.1.1. The initial packing $\mathcal{K}^\circ$ is done such that there is no fragmenting (i.e., $\Phi^\circ_m = \emptyset$) and each message is packed into its own frame, with the frame inheriting the message's size, period and deadline. The initial set $\mathcal{B}^\circ$ of BAGs for each RC VL is set as explained in Section 4.1.1.

The initial routing of virtual links $\mathcal{R}^\circ_{VL}$ is done to minimize the paths. We use Prim's algorithm [61] for minimum spanning tree to determine the initial vl°_i for each frame f_i. We call Prim's algorithm for each frame f_i. Let ES^{src}_i be the source of frame f_i and $\mathcal{ES}^{dest}_i$ be the set of destinations of frame f_i. The input to Prim's algorithm is the topology graph $\mathcal{G}_C$ from which we have removed all the ESes, except $ES^{src}_i \cup \mathcal{ES}^{dest}_i$. That is, we are interested in the minimum spanning tree in the graph that connects the ESes involved in a particular frame's transmission. For frame f_1 packing m_1 in Fig. 2.3, the graph is composed of vertices $\{ES_1, NS_1, NS_2, ES_3, ES_4\}$ and the edges interconnecting these vertices. The virtual link routing for frame f_1 is the minimum spanning tree in this graph, depicted with a red dash-dotted arrow, see Fig. 2.3.

The initial schedules $\mathcal{S}^\circ$ for the TT messages are built using the ASAP scheduling, where the ESes, NSes and dataflow links are considered the resources onto which the frame instances have to execute. The initial routing for the example in Fig. 4.1 is

presented in Fig. 4.1a using VLs vl_1 to vl_5 with red dash-dotted arrow, and the initial schedule results from the one-to-one packing and ASAP scheduling depicted in Fig. 4.2a.

(2) In the second step, we use a Tabu Search meta-heuristic (see Section 4.2.1) to determine the fragmenting Φ_m and packing $\mathcal{K}$ of messages in frames, the final set of virtual links $\mathcal{VL}$, the assignment of frames to virtual links $\mathcal{M}_F$, the routing of virtual links $\mathcal{R}_{VL}$, the BAGs for the RC VLs and the TT schedules $\mathcal{S}$, such that the TT and RC frames are schedulable, and the end-to-end delay of RC frames is minimized.

4.2.1 Tabu Search

Tabu Search (TS) [83] is a meta-heuristic optimization, which searches for that solution that minimizes the *cost function*. Tabu Search takes as input the topology of the network $\mathcal{G}_C$, the set of TT and RC messages $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$ (including the size, period/rate and deadline), and returns at the output the best configuration of (i) message fragmenting Φ_m and packing $\mathcal{K}$, (ii) the assignment of frames to virtual links $\mathcal{M}_F$, (iii) the routing of virtual links $\mathcal{R}_{VL}$, the (iv) the bandwidth for each RC virtual link and (v) the TT schedules $\mathcal{S}$ found during the design space exploration, in terms of the cost function. We define the cost function of an implementation Υ as:

$$Cost_{DOTTS}(\Upsilon) = w_{TT} \times \delta_{TT} + w_{RC} \times \delta_{RC} \tag{4.1}$$

where δ_{TT} is the "degree of schedulability" for the TT frames and δ_{RC} is the degree of schedulability for the RC frames. These are summed together into a single value using the weights w_{TT} and w_{RC}, given by the engineer. In case a frame is not schedulable, its corresponding weight is a very big number, i.e., a "penalty" value. This allows us to explore unfeasible solutions (which correspond to unschedulable frames) in the hope of driving the search towards a feasible region. Once the TT frames are schedulable we set the weight w_{TT} to zero, since we are interested to minimize the end-to-end delays for the RC frames. The degree of schedulability for TTEThernet frames is calculated as:

$$\delta_{TT/RC} = \begin{cases} c_1 = \sum_i \max(0, R_{f_i} - f_i.deadline) & \text{if } c_1 > 0 \\ c_2 = \sum_i (R_{f_i} - f_i.deadline) & \text{if } c_1 = 0 \end{cases} \tag{4.2}$$

If at least one frame is not schedulable, there exists one R_{f_i} greater than the deadline $f_i.deadline$, and therefore the term c_1 will be positive. However if all the frames are schedulable, this means that each R_{f_i} is smaller than $f_i.deadline$, and the term $c_1 = 0$. In this case, we use c_2 as the degree of schedulability, since it can distinguish between two schedulable solutions.

Tabu Search explores the design space by using design transformations (or "moves") applied to the current solution in order to generate neighboring solutions. In order to

TabuSearch($\mathcal{G}_C$, $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$, $\mathcal{K}^\circ$, $\mathcal{M}_F^\circ$, $\mathcal{R}_{VL}^\circ$, $\mathcal{B}^\circ$, $\mathcal{S}^\circ$)

```
1  Best ← Current ← < K°, M_F°, R_VL°, B°, S° >
2  L ← {}
3  while termination condition not reached do
4     remove tabu with the oldest tenure from L if Size(L) = l
5     // generate a subset of neighbors of the current solution
6     C ← CLG(Current, G_C, M^TT ∪ M^RC)
7     Next ← solution from C that minimizes the cost function
8     if Cost(Next) < Cost(Best) then
9        // accept Next as Current solution if better than the best-so-far Best
10       Best ← Current ← Next
11       add tabu(Next) to L
12    else if Cost(Next) < Cost(Current) and tabu(Next) ∉ L then
13       // also accept Next as Current solution if better than Current and not tabu
14       Current ← Next
15       add tabu(Next) to L
16    end if
17    if diversification needed then
18       Current ← Diversify(Current)
19       empty L
20    end if
21 end while
22 return Υ =< Φ_m, K, M_F, R_VL, B, S >
```

Figure 4.7: The Tabu Search algorithm

increase the efficiency of the Tabu Search, and to drive the search intelligently towards the solution, these "moves" are not performed randomly, but chosen to improve the search. If the currently explored solution is better than the best known solution, it is saved as the "best-so-far" *Best* solution. To escape local minima, TS incorporates an adaptive memory (called "tabu list"), to prevent the search from revisiting previous solutions. Thus, moves that improve the search are saved as "tabu". In case there is no improvement in finding a better solution for a number of iterations, we use *diversification*, i.e., we visit previously unexplored regions of the search space.

Fig. 4.7 presents the Tabu Search algorithm. Line 1 initializes the *Current* and *Best* solutions to the initial solution formed by the tuple $< \mathcal{K}^\circ, \mathcal{M}_F^\circ, \mathcal{R}_{VL}^\circ, \mathcal{B}^\circ, \mathcal{S}^\circ >$. Line 2 initializes the tabu list L to an empty list. The size of the tabu list, i.e., its *tenure*, is set by the user. The TS algorithm runs until the termination condition is not reached (see line 3). This termination condition can be, for example, a certain number of iterations or a number of iterations without improvement, considering the cost function [82]. Our implementation stops the search after a predetermined amount of time set by the user. In case the tabu list L is filled, we remove the oldest tabu from this list (see line 4).

Evaluating all the neighboring solutions is infeasible, therefore we generate a subset of neighbors of the *Current* solution (line 6), called *Candidate List*, by running the *Candidate List Generation* (CLG) algorithm (see Section 4.2.3), and the algorithm chooses from this *Candidate List*, as the *Next* solution, the one that minimizes the cost function (line 7). In case this *Next* solution is better than the best-so-far *Best* solution (lines 8–11), TS sets the *Best* and *Current* solutions as the *Next* solution. TS accepts a solution generated by a tabu move only if it is better than the best known solution *Best*. Accepting a solution generated by a tabu move is referred to as "aspiration criteria". Then, the TS algorithm adds the move that generated this solution to the tabu list, to prevent cycling. If the move is already a tabu, it will be added to the head of the list, thus setting its tenure to the size of the list. If *Next* improves the cost function compared to the *Current* solution, but not to the *Best*, and furthermore, the move that generated *Next* is not a tabu, TS accepts it as the *Current* solution, and adds the move to the tabu list.

In case the TS algorithm does not manage to improve the current solution after a number of iterations (lines 17–20), TS proceeds to a diversification stage (line 18). During this stage, TS attempts to drive the search towards an unexplored region of the design space. As such, the algorithm randomly re-assign a task from each application, while keeping the same partition tables. After such a diversification stage, the tabu list L is emptied.

4.2.2 Design Transformations

We use three classes of moves in our Tabu Search: (1) *routing* moves applied to virtual links, (2) *packing* moves applied to messages and (3) *scheduling* moves applied to the TT frames.

(1) The *reroute* move is applied to a virtual link vl_i carrying a frame f_i. This move returns a new tree for the virtual link vl_i, which has the same source and destinations, but goes through different dataflow links and network switches. The new tree is randomly selected, but the *reroute* move can also have a parameter specifying a dataflow link dl_i to avoid in the new tree, because, for example, we have determined that dl_i is too congested.

The reroute move selects from the complete set of trees that can be used to route a virtual link vl_i. This set is determined only once, before TS is run, for every message m_i. We use breadth-first search to find every path between the source of m_i and it's destinations, and we combine these paths to obtain a complete set of unique trees. When several messages are packed into a frame, we take the union of sets of trees for each message in the frame.

(2) The *fragmenting / packing* moves change the structure of the extended messages set $\mathcal{M}^+$ and the assignment of messages to frames $\mathcal{K}$. There are two types of *fragmenting moves*: *fragment message* and *un-fragment message*, and two types of *packing moves*: *pack messages* and *unpack frames*. The *fragment message* move splits a message m_i into several same-sized message fragments $m_j \in \mathcal{M}^+, m_j \in \Phi_m(m_i)$. Each message fragment inherits the period and deadline of the message m_i. In case of the RC messages, each vl_j carrying the RC frame f_j that is packing one message fragment m_j, will inherit the BAG of vl_i carrying m_i. The *un-fragment message* undoes the *fragment message* move, and regroups all the fragments $m_j \in \Phi_m(m_i)$ back into the original message m_i.

The *pack messages* move packs into the same frame several messages and/or message fragments that (i) have the same source and destinations, (ii) belong to the same traffic class, (iii) have the same SIL and (iv) that the sum of their size does not exceed the maximum allowed payload size of 1471 B. In case we pack messages with different periods and deadlines, the new frame f_i will inherit the tightest deadline and the smallest period of the composing messages and fragments. For RC messages, the new frame f_i will inherit the smallest BAG of the composing messages.

Packing of message fragments from different frames can further reduce the WCD of the messages involved, similarly to the example of packing RC frames given in Section 4.1.2. Although packing message fragments of different messages is possible, we do not consider this to be realistic, hence we do not employ this in our optimization. Also note that the ARINC 664p7 protocol has a restriction of 4096 VLs per cluster. The *pack messages* move can be used to circumvent this restriction, in case there are more than 4096 messages to be sent.

The *unpack* move applied to frame f_i assigns each $m_j \in \mathcal{M}^+, \mathcal{K}(m_j) = f_i$, to a new frame f_j, on a one-to-one basis.

For example, let us consider $\mathcal{M} = \{m_1, m_2, m_3\}$. By fragmenting m_1 into 3 fragments, we obtain $\Phi_m(m_1) = \{m_{1/1}, m_{1/2}, m_{1/3}\}$, with the periods and deadlines equal to m_1, and their size equal to $\lceil m_1.size/3 \rceil$. Similarly, fragmenting m_3 into 2 fragments, we get $\Phi_m(m_3) = \{m_{3/1}, m_{3/2}\}$. Thus, $\mathcal{M}^+ = \{m_{1/1}, m_{1/2}, m_{1/3}, m_2, m_{3/1}, m_{3/2}\}$. Performing the *un-fragment* move on m_3 will result in $\mathcal{M}^+ = \{m_{1/1}, m_{1/2}, m_{1/3}, m_2, m_3\}$. If we pack $m_{1/1}$ and m_2 into frame f_x, such that $\mathcal{K}(m_{1/1}) = f_x$ and $\mathcal{K}(m_2) = f_x$, $f_x.deadline$ is determined as $min(m_{1/1}.deadline, m_2.deadline)$ and for TT messages, $f_x.period = min(m_{1/1}.period, m_2.period)$, while for RC messages, $f_x.rate = min(m_{1/1}.rate, m_2.rate)$. An example of packing TT messages into the same frame is presented in Fig. 4.3c.

(3) Let us now discuss the *scheduling* moves. A periodic frame f_i has several frame instances. For the *scheduling* moves we introduce the following notations: we denote with $f_{i,x}$ the x^{th} instance of frame f_i, and with $f_{i,x}^{[\nu_j, \nu_k]}$ the instance sent on the dataflow

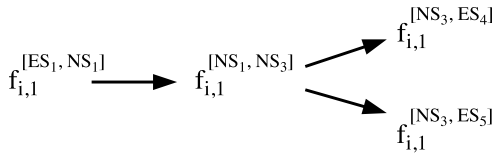


Figure 4.8: Representation of a frame as a tree

link $[\nu_j, \nu_k]$. All the frame instances $f_{i,x}^{[\nu_j,\nu_k]}$ of frame f_i have the same offset across all periods. Let us consider the topology presented in Fig. 4.1a, and frame f_i transmitted from ES_1 to ES_4 and ES_5 along the shortest route, that is vl_1={[[ES_1, NS_1], [NS_1, NS_3], [NS_3, ES_4], [NS_3, ES_5]]}. The tree model that represents the frame f_i is shown in Fig. 4.8. Each frame f_i is assigned a virtual link vl_i. A virtual link is a tree structure, where the sender is the root and the receivers are the leafs. In the case of a virtual link, the ESes and NSes are the nodes, and the dataflow links are the edges of the tree. However, in our tree model of a frame, the dataflow links are the nodes and the edges are the precedence constraints. Naturally, frame instance $f_{i,1}$ on dataflow link $[NS_3, ES_5]$ cannot be sent before it is transmitted on $[NS_1, NS_3]$ and received in NS_3. Such a precedence constraint is captured in the model using an edge, e.g., $f_{i,1}^{[NS_1,NS_3]} \rightarrow f_{i,1}^{[NS_3,ES_5]}$. We denote with $pred(f_{i,x}^{[\nu_j,\nu_k]})$ the set of predecessor frame instances of the frame instance $f_{i,x}$ on dataflow link $[\nu_j, \nu_k]$ and with $succ(f_{i,x}^{[\nu_j,\nu_k]})$ the set of successor frame instances of the frame instance $f_{i,x}^{[\nu_j,\nu_k]}$.

We propose 4 *scheduling* moves: *advance*, *advance predecessors*, *postpone* and *postpone predecessors*. The *advance* move will advance the scheduled send time offset of a TT frame f_i from node ν_j on a dataflow link $[\nu_j, \nu_k]$ to an earlier moment in time. The *advance predecessors* applied to a frame f_i will advance the scheduled send time offset for all its predecessors. Similarly, the *postpone* move will postpone the schedule send time offset of a TT frame from a node, while *postpone predecessors* will postpone the send time offset for one random predecessor of that frame.

The maximum amount of time a frame instance is advanced or postponed at a node $\nu_j \in \mathcal{V}_C$ is computed such that the frame instance will not be sent before it is scheduled to be received, or sent too late to meet its deadline. For each node ν_j, we compute the latest absolute send time for frame f_i so that it may still meet its deadline, ignoring other traffic. Also, after each move we may need to adjust the schedules (move other frame offsets later or earlier) to keep the solution valid, i.e., the schedules respect the precedence and resource constraints.

Tabu Search relies on a memory structure called “tabu list” to prevent the search from cycling through previously visited solutions, back to a local optima. Our algorithm

relies on a *tabu list* with tabu-active attributes, that is, it does not remember whole solutions, but rather attributes of the moves that generated the tabu solutions. For each tabu, we record the move that generated it, and the affected frames or messages.

4.2.3 Candidate List

As previously mentioned, Tabu Search drives the search towards schedulable solutions by applying "moves" to the current solution in order to generate neighboring solutions. The number of neighbors for each solution is very large, therefore evaluating all the neighboring solutions is infeasible. Instead, our algorithm evaluates only a subset of neighbors of the *Current* solution, called *Candidate List*. One option is to randomly select the neighbors placed on the candidate list. However, our algorithm uses a heuristic approach that selects those neighbors which have a higher chance to quickly lead to a good result. The Candidate List Generation (CLG) algorithm is described in the following. Each candidate solution is obtained by performing moves on the *Current* solution.

We consider the following classes of candidates: (1) candidates for TT frames, (2) candidates for RC frames and (3) randomly generated candidates.

4.2.3.1 Candidates for TT Frames

CLG generates a set of candidates for the unschedulable TT frames, and another set for schedulable TT frames. First we describe the candidates for unschedulable frames. For each unschedulable TT frame f_{TT}, CLG identifies the first dataflow link $dl_x \in \mathcal{R}_{VL}(\mathcal{M}_F(f_{TT}))$ where f_{TT} is unschedulable, i.e., where $f_{TT}^{dl_x}$ is sent too late for f_{TT} to reach its deadline. CLG creates candidate solutions by performing *reschedule*, *reroute*, *packing* and *fragmenting* moves to $f_{TT}^{dl_x}$ separately on the *Current* solution. In case f_{TT} is packed, CLG performs an *unpack* move instead. Similarly, if f_{TT} is fragmented, CLG performs an *unfragment* move. Next, CLG targets TT frames that might delay f_{TT} excessively. The high rate frames and the very "large" frames on dl_x are such frames. CLG *reroutes* the TT frame with the highest rate on dl_x to another link, thus decongesting dl_x and increasing f_{TT}'s chances to be schedulable. Similarly, CLG *reroutes* the largest TT frame to another randomly selected route. Furthermore, CLG *reroutes* a random frame on dl_x to another randomly selected route.

Our optimization is driven by the cost function specified in Eq. 4.1 (see Section 4.2.1). Thus, TS searches for a solution that makes TT and RC frames schedulable, and minimizes the end-to-end delay of the RC frames. Therefore, once the TT frames are schedulable, TS does not look for solutions that reduce the end-to-end delay of the

TT frames. Instead, it applies moves to the schedulable TT frames to minimize the end-to-end delay of the RC frames. Thus, the next moves focus on schedulable TT frames.

In this context, first, the CLG algorithm selects the TT frames with the highest degree of schedulablity and generates a candidate solution by *rerouting* each such frame to another route. Although this move may reduce the degree of schedulability of the rerouted frames, as a side effect, it may decongest some dataflow links. Furthermore, CLG also generates other candidates by *rescheduling* these frames.

Second, CLG selects schedulable TT frames with lowest degree of schedulability, and *reroutes* each such frame on a randomly chosen alternative route. Third, CLG generates candidates by *packing* the smallest schedulable TT frames, to consolidate the schedule. Fourth, similarly with the previous candidates, CLG *fragments* in equally sized frame fragments the largest TT frames. For the *pack* and *fragment* moves, CLG randomly choses the number of the frames and the number of the fragments, respectively, so the size of the resulting frames respect the size constraints (see Section 2.2.2.1).

4.2.3.2 Candidates for RC Frames

Similarly with the candidates for TT frames (previously described), CLG generates two sets of candidates: one set for the unschedulable, and another set for schedulable RC frames. For each f_{RC} unschedulable RC frame, the CLG algorithm identifies the first dataflow link dl_x where f_{RC} is unschedulable. Then, CLG creates candidate solutions by applying the following moves separately on the current solution: (i) CLG *reroutes* f_{RC} to another, randomly selected route, (ii) *fragments* and (iii) *packs* f_{RC}. In case f_{RC} is already fragmented, CLG *unfragments* the frame instead. Similarly, if the frame is already packed, CLG *unpacks* it.

There are cases where a high rate TT frame might greatly delay RC frames. Let $hr_{TT}^{dl_x}$ be the TT frame on dl_x with the highest rate. *Rerouting* $hr_{TT}^{dl_x}$ to another, randomly selected, route decongests dl_x, possibly reducing the delay for f_{RC} on this dataflow link. *Rescheduling* $hr_{TT}^{dl_x}$ might create sufficient time to reduce f_{RC}'s delay. CLG also creates candidates by *packing* and *fragmenting* $hr_{TT}^{dl_x}$. Similarly to the high rate TT frame $hr_{TT}^{dl_x}$ on dl_x, there are cases where large TT frame will delay RC frames. Let $lg_{TT}^{dl_x}$ be the largest TT frame on dl_x. CLG applies moves that *reroute*, *pack* and *fragment* $lg_{TT}^{dl_x}$ and moves that *advance* and *postpone* $lg_{TT}^{dl_x}$ on dl_x, just like in the case of $hr_{TT}^{dl_x}$.

Next, CLG focuses on schedulable RC frames to improve their schedulability. For these candidates, first, CLG targets the f_{RC} RC frames with highest degree of schedulability, *rerouting* each such frame to another route. Although this move may reduce the degree

of schedulability of f_{RC}, as a side effect, it may decongest some dataflow links, reducing the worst-case end-to-end delay (WCD) of other RC frames. Second, CLG focuses on schedulable RC frames with the lowest degree of schedulability, *rerouting* them in order to increase their schedulability. Third, CLG focuses on the smallest and largest RC frames. Thus, CLG creates candidates by *packing* the smallest RC frames, and by *fragmenting* the largest RC frames, respectively. The packing and fragmenting moves are done such that they respect the constraints presented in Section 4.2.2.

4.2.3.3 Randomly Generated Candidates

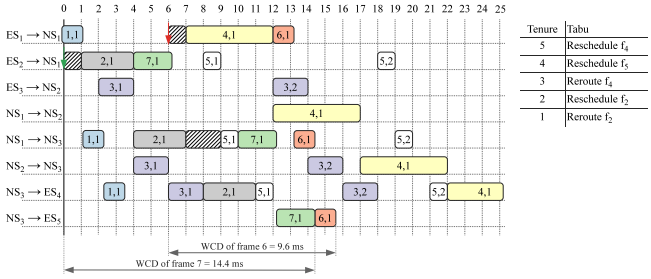
As the previous moves are targeting specific frames, in order to increase the degree of schedulability, CLG introduces a third set of candidates. On a randomly selected set of frames, CLG randomly applies packing, fragmenting or routing moves.

4.2.4 Tabu Search Example

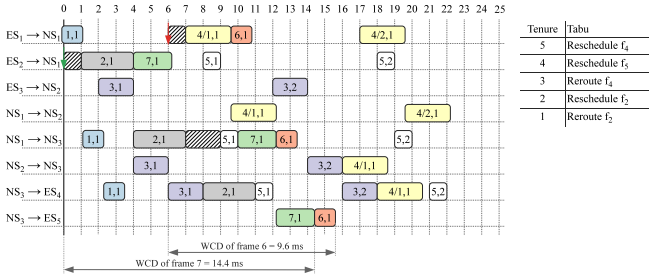
We illustrate next how Tabu Search works. We consider the applications from Fig. 4.1. The current solution, which is also the best-so-far solution, is presented in Fig. 4.9a. This solution is also presented in Fig. 4.4b, and is obtained from Fig. 4.2b by *rerouting* the TT frame f_4 via NS_2. The following 5 solutions, Fig. 4.9b to Fig. 4.9f show possible candidate solutions obtained from Fig. 4.9a. Next to each solution, we present the associated tabu list. We consider a tabu tenure of 5. The current state of the tabu list is shown next to Fig. 4.9a.

To reduce the delays, the CLG algorithm proposes candidates which fragment the largest TT frames (see Section 4.2.3). Fig. 4.9b shows a candidate solution obtained in this way. Frame f_4 is the largest frame in the system. Thus, fragmenting m_4 into two frames $f_{4/1}$ and $f_{4/2}$, reduces the delay on $[ES_1, NS_1]$ for f_6 from 7.2 to 4.83 ms. Unfortunately, reducing the delay for f_6 on $[ES_1, NS_1]$ does not improve the overall WCD for f_6. This solution does not improve the current solution, and hence, is ignored. CLG generates candidates also by rescheduling the largest TT frame on the dataflow link where it delays RC frames. Fig. 4.9c presents such a candidate solution, advancing f_4 in the schedule of $[ES_1, NS_1]$. This solution is tabu (tenure 4), and because this candidate is not better than the *Best* solution, it is ignored.

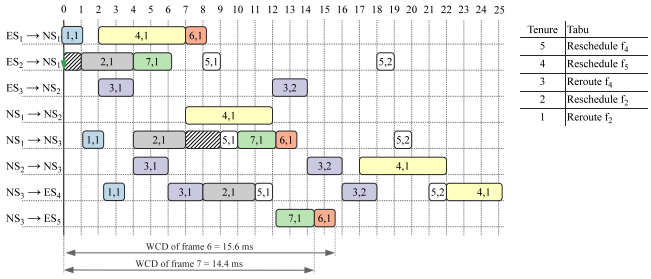
Another set of candidate solutions is obtained by fragmenting a given percentage of the largest RC frames in the system. Fig. 4.9d presents such a solution. Message m_7 is fragmented into $f_{7/1}$ and $f_{7/2}$. The newly created frames have the same BAG as f_7, of 16 ms, and a transmission duration of $C_{7/1} = C_{7/2} = 1.25$ ms. Thus, $f_{7/1,1}$ can be transmitted on $[NS_1, NS_3]$, in the interval between $f_{2,1}$ and $f_{5,1}$, which previously was



(a) Current solution

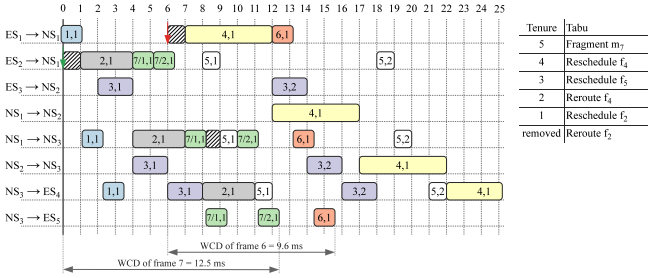


(b) Fragmenting message m_4, does not improve the current solution

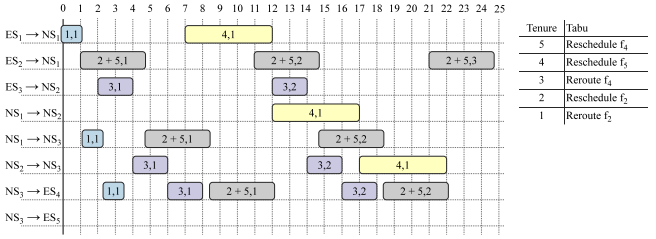


(c) Reschedule f_4 does not improve the best-so-far solution and is tabu, thus ignored

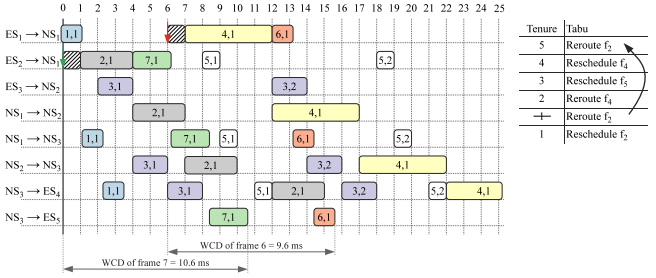
Figure 4.9: Candidate solutions and their tabu list



(d) Fragment message m_7. Better than the current solution



(e) Pack m_2 and m_5 in f_{2+5}. f_4 does not fit into the schedule, thus results in the worst solution so far and is ignored



(f) Reroute f_2 via NS_2. Although tabu, the move results in a solution better than the current solution, and thus accepted as the best so far and set as current

Figure 4.9: Candidate solutions and their tabu list

timely blocked for f_7. This move reduces the WCD of m_7 to 12.5 ms, thus improving the solution. Overall, this solution is better than the *Current* solution. If this candidate solution is chosen as the next solution, the tabu for the fragmenting m_7 is added to the head of the tabu list, with a tenure of 5. The tenures of the other tabus in the list are decremented, and the "Reroute f_2" tabu, previously with a tenure of 1, is removed from the list. The update tabu list is in Fig. 4.9d.

Fig. 4.9e presents a solution obtained by packing m_2 and m_5 into a single frame, f_{2+5}. CLG generates candidate solutions by packing the smallest frames, to reduce the reserved bandwidth and the delay on the network. Frame f_5 is the smallest frame in our example. As previously mentioned in Section 4.2.2, only messages with the same source and destination ESes may be packed into one frame. For f_{2+5}, the size of the frame equals the sum of the messages involved, that is 933 B, with $C_{2+5} = 3.73$ ms. The period of f_{2+5} is the smallest of the messages involved, i.e., 10 ms. In this new frame configuration, f_4 does not fit in the schedule of dataflow link $[NS_3, ES_4]$. The candidate solution in Fig. 4.9e is the worst solution so far, and consequently, it is ignored.

Rerouting f_2 via NS_2, see Fig. 4.9f, reduces the WCD for f_7 from 14.4 to 10.6 ms. Although the move is tabu, the solution is better than the *Current* and *Best* solutions, and thus, is accepted as the *Current* and best-so-far *Best* solution. The next TS iteration will continue with this solution as *Current*. The updated tabu list is also presented in Fig. 4.9f.

4.3 Experimental Evaluation

For the evaluation of our proposed optimization approach, "Design Optimization of TTEthernet-based Systems" (DOTTS), we used 7 synthetic benchmarks and two real-life case studies. The DOTTS algorithm was implemented in Java (JDK 1.6), running on SunFire v440 computers with UltraSPARC IIIi CPUs at 1.062 GHz and 8 GB of RAM.

The details of the benchmarks are presented in Table 4.1. For the synthetic benchmarks, we have used 6 network topologies, and we have randomly generated the parameters for the frames, taking into account the details of the TTEthernet protocol. All the dataflow links have a transmission speed of 100 Mbps. In columns 3–6, we have the details of each benchmark: the number of ESes, NSes, the load of the system and the number of messages, respectively. The load within an application cycle T_{cycle} is calculated as the ratio of the sum of the sizes of all frame instances divided by the network speed. The number of frame instances in the network, considering a one-to-one mapping of messages to frames, can be found in column 7. This number is much larger than the number of messages: there is a frame instance for each dataflow link on

which a message is transmitted to reach its destination. The number of frame instances will be influenced by packing and fragmenting.

With the first set of experiments, "Set 1", we were interested to evaluate DOTTS in terms of its ability to find schedulable implementations. Thus, we used synthetic benchmarks where we gradually increased the size of the system, both in number of messages and number of network nodes. The results obtained by DOTTS were compared with four other optimization approaches. The first approach is the Straightforward Solution (SS) presented in Section 4.1.1 and implemented by the box "Initial Solution" in Fig. 4.6. This is what a good engineer would do without the help of our optimization tool. The other three approaches are based on the same Tabu Search optimization as DOTTS, but they restrict the type of optimization performed. Thus, Routing Optimization (RO) optimizes only routing, using SS for packing and scheduling. Packing and Fragmenting Optimization (PFO) optimizes only fragmenting and packing, and not routing and scheduling. Scheduling Optimization (SO) optimizes the schedules but keeps the packing and routing from SS. These TS implementations correspond to the boxes RO, PFO and SO in Fig. 4.6, where only the respective type of moves are performed in the TS.

The results for Set 1 are presented in Table. 4.1, lines 2–6. We are interested in finding schedulable implementations. Thus, for each optimization algorithm, we report the percentage of schedulable messages in the system, after applying the respective optimization. We used a time limit of 45 minutes for all algorithms. In these experiments, we were interested to determine how DOTTS performs as the complexity of the system increases from 25 to 45 ESes and NSes. The load of the system is 40–90 %, and the number of frame instances grows from 2305 to 5509. As we can see from the results, DOTTS is able to find schedulable implementations (all the TT and RC messages are schedulable) for the benchmarks in the Set 1, see column 12, in Table 4.1. SS performs poorly, with only 48% schedulable messages for example for benchmark 13 (column 8). This shows that performing the design optimization of TTEthernet-based systems is very important.

Table 4.1: DOTTS experimental results

Set	Benchmark	ES	NS	Load	Messages	Frame Instances	SS Sched.%	RO Sched.%	PFO Sched.%	SO Sched.%	DOTTS Sched.%
1	1.1	37	8	40	43	2305	53.48	86.04	81.39	95.34	100.00
	1.2	20	5	60	84	2388	52.38	67.85	68.86	97.19	100.00
	1.3	37	8	40	77	2441	48.05	70.12	64.93	99.52	100.00
	1.4	35	8	40	132	4064	63.63	66.66	68.18	98.48	100.00
	1.5	20	5	90	145	5509	58.62	62.96	63.88	81.37	100.00
2	2.1	30	8	40	180	5809	48.88	53.33	55.00	100.00	–
	2.2			50	220	6871	50.00	53.18	54.09	78.18	100.00
	2.3			60	220	7811	50.00	51.81	54.54	74.09	100.00
3	auto	15	7	50	79	5180	53.16	58.22	72.34	89.87	100.00
	orion	31	14	40	187	6130	46.52	58.82	57.75	100.00	–

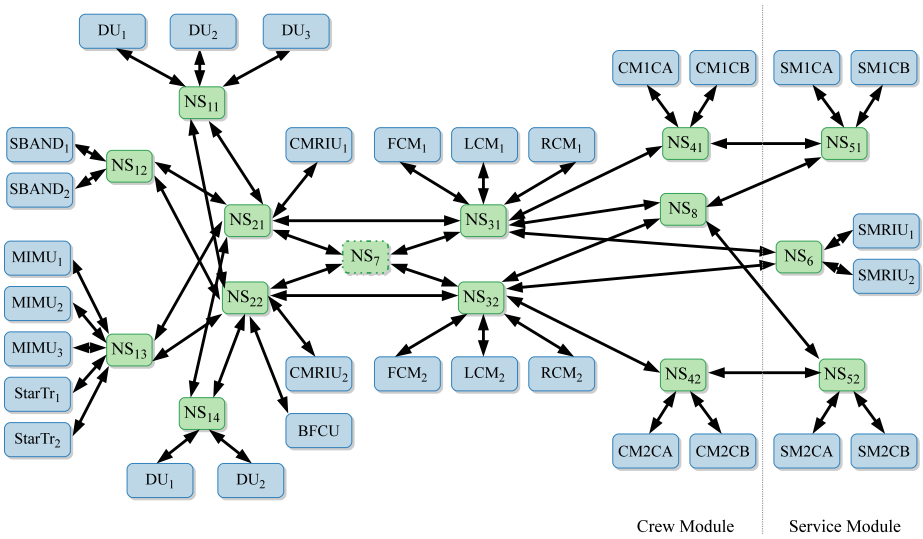


Figure 4.10: Network topology of the Orion CEV, derived from [126]

The next comparison of DOTTS is with RO, PFO and SO. The question is, where is the improvement of DOTTS coming from, compared to SS, from which kind of optimization: routing, packing/fragmenting or scheduling? As expected, SO, which performs schedule optimization, obtains the best result among RO, PFO and SO, but very rarely does it obtain schedulable solutions. Furthermore, PFO and RO are not consistently better one than the other. The conclusion is that they should all be used together, as we do in DOTTS.

For the second set of experiments, labeled "Set 2" in Table 4.1, we were interested in determining how our optimization approach DOTTS handles increased loads (while the architecture does not change). We have used an architecture of 38 ESes and NSes and we have increased the number of messages leading to loads of 40 to 60%. In the case of benchmark 21, Table 4.1, SO is able to find a schedulable implementation. In this case, we do not run DOTTS for this benchmark anymore. As the load of the system increase to 50 and 60% (benchmarks 22 and 23), SO does not perform well, but DOTTS is able to find schedulable implementations.

In the third set of experiments, labelled with "Set 3", we used two real-life benchmarks. The first benchmark is derived from [117], based on the SAE automotive communication benchmark [2]. In this benchmark we have 22 network nodes (ESes and NSes), and 79 messages (with the parameters generated based on the messages presented in [117]). The results for this benchmark are shown in Table 4.1, in the row labelled "auto". The other benchmark is derived from [126], based on the Orion Crew Exploration Vehicle

(CEV), 606E baseline [126] and labeled in Table 4.1 with “orion”. In this benchmark we have 45 network nodes (ESes and NSes) and 187 messages (with the parameters generated based on the messages presented in [126]). This benchmark is described in detail in Section 5.3. The topology for this benchmark is shown in Fig. 4.10. The results obtained for the real-life benchmarks confirm the results of the synthetic benchmarks.

CHAPTER 5

Design Optimizations for Mixed-Criticality Space Applications

We have presented the evaluation of our proposed optimization strategies on several benchmarks, including real-life case studies. The assumptions so far were that all the applications, including the non-critical ones, are hard real-time. The purpose of this chapter is to discuss the issues related to implementing mixed-criticality applications (both in safety and time domains) on partitioned architectures in the context of a given application area (space) and realistic applications. In this context, handling also soft real-time and best-effort requirements is important. For optimizations at the processing level we consider two applications: the Mars Environment Survey (MESUR) Pathfinder Rover (described in Section 5.2.1) and the Compositional Infrared Imaging Spectrometer (CIRIS; see Section 5.2.2). At the communication level, we consider the Orion Crew Exploration Vehicle (see Section 5.3). Part of this work (i.e., the controller for CIRIS) was done at the Jet Propulsion Laboratory (JPL), National Aeronautics and Space Administration (NASA), during a five-month research visit.

In this chapter we will discuss how the methods and tools we presented in Chapter 3 and in Chapter 4 can be extended to consider soft real-time and best effort requirements. The chapter is organized as follows. First, we introduce the space application area. Second, we present the applications used at the PE-level. Third, we present

the application used at the communication level. Fourth, we present and evaluate the extensions we made to our optimizations to handle soft real-time constraints.

5.1 Background

Researchers from the European Space Agency (ESA) have advocated the use of partitioned architectures (PAs) in spacecraft avionics, as a way to "manage the growth of mission functions implemented in the on-board software" [185]. A similar case was made by researchers from NASA [91]. The number of missions carrying payloads from different stakeholders increases, resulting in more integration into the same platform to reduce the size, weight and power consumption (SWaP) of the system. In most cases, these components have different criticality levels. Some are safety-critical (e.g., life support systems in a space craft), mission-critical (e.g., propulsion system) or non-critical (e.g., scientific instruments that are not part of the primary mission). In such cases, safety and security constraints require that the platform has protection mechanisms to ensure that applications do not interfere with each other. PAs implement the protection mechanisms to handle the safety constraints. Currently, ESA is working on adding security components to PAs [184]. Furthermore, ESA views PAs as an intermediate step to introducing multi-core processors in spacecraft computers [184].

Partitioned architectures rely on partitioning mechanisms at the platform level to ensure temporal and spatial separation between applications of different criticality levels, and thus to allow the safe integration on the same platform (see Section 2.2.1 for more details). Spatial partitioning protects the private data or devices of an application in a partition from being tampered with, by another application. It usually relies on hardware mechanisms such as Memory Management Units (MMUs). Alternatives were proposed for spatial partitioning in spacecraft processors that do not have MMUs [172]. An example of a spacecraft that could have benefited from a partitioned architecture implementation is the Phobos I spacecraft, lost due to the failure of a non-critical application (i.e., keyboard buffer overflow) interfering with the flight critical software [141].

A detailed discussion on the benefits of PAs for spacecraft can be found in [183, 91]. To name just a few of the advantages for spacecraft platforms of using partitioned architectures: they allow the safe and secure integration of applications of different criticality levels and from different stakeholders, reduce the the SWaP of the system and the development, verification and integration costs.

5.2 Processor-Level Partitioning

For the processor-level evaluation of partitioning, we chose two applications of different criticality levels. The proposed scenario is two have two applications of different safety and time criticality integrated onto the same processor. One application is the Mars Pathfinder Mission [63], mixed-criticality application, described in Section 5.2.1. The other application is non-critical; the controller for the Compositional Infrared Imaging Spectrometer (CIRIS), which is a Fourier Transform Spectrometer is described in Section 5.2.2. The controller for CIRIS was developed during a research visit at JPL, NASA. First we describe the CIRIS controller, and then we present another version that is considered for integration with the Mars Pathfinder Mission.

We assume the two applications are running on a single processor, under a partitioned operating system. We present the results of our evaluation in Section 5.4.1.

5.2.1 Mars Pathfinder Mission

The Mars Pathfinder, also known as the Mars Environment Survey (MESUR) Pathfinder, was a spacecraft designed, built and operated by JPL, NASA. The mission was second in NASA's Discovery program, which aimed at making cheaper spacecrafts to explore the Solar System. The spacecraft was launched on 4 December 1996, landing on Mars on 4 July 1997, using airbags as a new landing method. The spacecraft contained a lander, later named as the "Carl Sagan Memorial Station", and a robotic rover, named Sojourner, controlled by an Earth-based operator. Both the lander and rover exceeded their planned lifetimes. During the mission, the lander returned 2.3 gigabits of information, including images from both the lander and rover, chemical analyses of rocks and soil, and atmospheric measurement conditions. The scientific analyses results suggest that in a distant past Mars was a warm planet, with liquid water and a thicker atmosphere [1].

The hardware architecture of the Mars Pathfinder is presented in Fig. 5.1. The main processor of the spacecraft is the RS 6000 microprocessor on the lander, running the VxWorks real-time operating system [182]. The rover contains an Intel 8085 processor that performs automatic controls. The main processor is connected to the memory, camera and the radio providing connection to Earth via a VME bus, and to the scientific instruments via a 1553 Bus. The communication between the rover and the lander is performed through the 1553 Bus via a wireless link inherited from the Cassini spacecraft [63].

Although the software consists of over 25 tasks, the mission had several operating modes, with tasks being active only in specific operating modes. We focus on the tasks

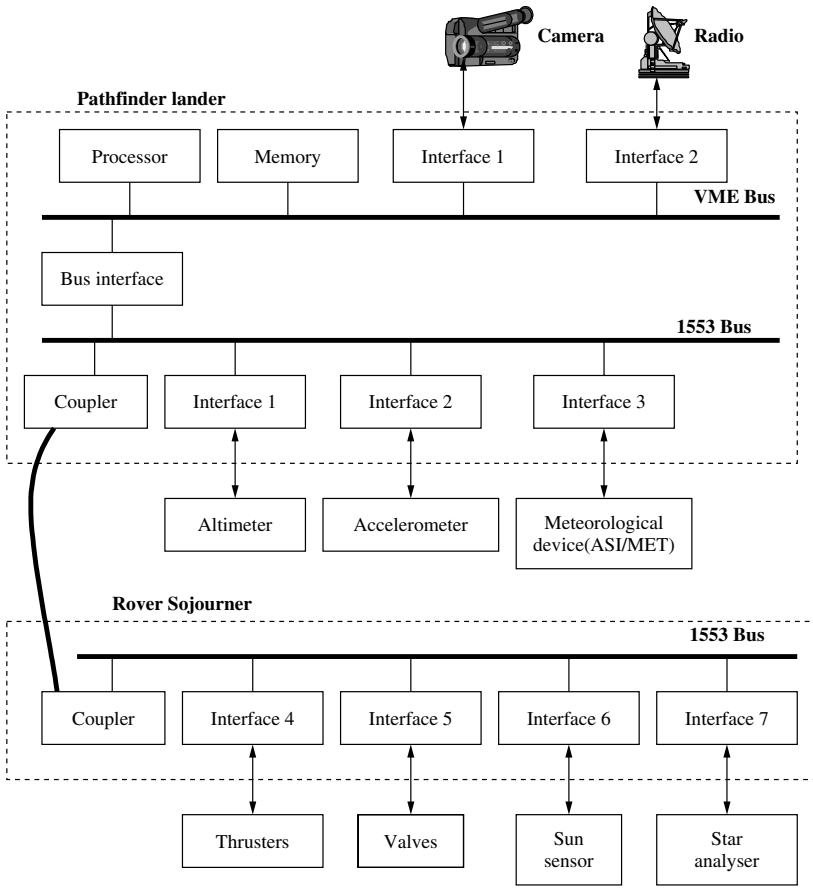


Figure 5.1: Hardware architecture of the Pathfinder spacecraft (from [63])

active in exploration mode. This set of tasks is listed in Table 5.1. The "Bus scheduling" task has top priority, as it is sets up and verifies the transactions on the 1553 Bus. The "Data distribution" task has the second highest priority in the exploration mode. This task collects data from the instruments into the shared data module [63]. The "Control task" is responsible with controlling the Sojourner rover. The rover has two types of hardware devices: control devices and measurement devices [63]. The "Radio task" manages the radio communication between the lander and Earth. We consider these four tasks as highly critical, while the three tasks with the lowest priority have a lower criticality level. The "Camera task" controls the camera on the lander. During its mission, the lander sent back 16500 images back to Earth. The "Measure task" collects measurements, while the "Meteo task" is responsible with the meteorological data [63].

Table 5.1: Pathfinder mission, exploration mode task set parameters

Set	Task name	Priority	Parameters	
			C_i	T_i
1	Bus scheduling	7	25	125
	Data distribution	6	25	125
	Control task	5	25	250
	Radio task	4	25	250
2	Camera task	3	25	250
	Measure task	2	50	5000
	Meteo task	1	75	5000

In Section 5.4.1 we will discuss how our optimization strategy can be extended to consider soft-real time applications. For this, we will integrate the MESUR tasks with the CIRIS application (see Section 5.2.2.5) onto the same processor. In this case, partitioning is necessary due to safety and timing separation requirements. The MESUR tasks are mixed-critical, with 4 high-criticality and 3 low-criticality tasks, while the CIRIS application is non-critical. Furthermore, the MESUR tasks are hard real-time, while the CIRIS tasks are soft real-time. Moreover, applications are scheduled using different policies. The MESUR tasks use FPS, while the CIRIS tasks use SCS.

The MESUR tasks are hard real-time applications, and they are scheduled using a fixed priority preemptive policy, i.e., "Rate Monotonic" (RM). In RM scheduling, the task with the smallest period has the highest priority. Liu and Layland [113] proved that a sufficient condition for schedulability for RM scheduled tasks set is if the utilization is below a bound, which depends on the number of tasks. In the case of an unpartitioned OS the processor utilization of this task set is equal to 0.725, lower than 0.729, i.e., the sufficient condition for RM scheduling for 7 tasks. But in partitioned systems, this schedulability condition is not useful anymore, as the RM tasks execute inside a partition and their execution is interrupted not only by higher priority tasks, but also by the other partitions. In such cases, we require a response time analysis, similar to the one presented in Section 3.5, to determine the schedulability of a task set.

5.2.2 Fourier Transform Spectrometer Controller for Partitioned Architectures

Spectroscopic techniques allow scientists to determine the composition of remote substances. Although there are numerous such techniques, most space-based spectrometers are dispersive spectrometers that measure the absorption of light in the near-infrared spectrum (wavelengths between 1 to 5 μm). Fourier Transform Infrared (FTIR) spectrometers are better suited for remote sensing, as they offer a considerable higher

throughput (called the Jacquinot or throughput advantage) compared to the dispersive spectrometers [145], and also due to the mid-infrared range they operate in, which contains the fundamental vibrations for most of the relevant compounds.

In the following, we will describe the implementation of the controller for a FTIR spectrometer developed at JPL, NASA, for space exploration and field measurements in rugged conditions. The instrument is the Compositional InfraRed Imaging Spectrometer (CIRIS), based on the TurboFT [177] spectrometer design. Anderson et al. [27] propose a similar FTIR instrument based on the TurboFT spectrometer that can be used for Mars missions, as well as in Antarctic field studies. A similar concept to the TurboFT instrument was developed for a spectrometer onboard the European Mars Express mission [80]. CIRIS is operational in the spectral range of 2.8 to 18 μm, or 3571 to 555 cm^{-1}, detecting various organic and inorganic compounds that are relevant for scientist.

The CIRIS controller is a soft real-time application. The real-time requirements of CIRIS stem from the fact that it has to acquire a 8192 points interferogram over a period of 33 ms every 100 ms. We have presented in Section 1.1 the difference between soft and hard real-time systems, and we talk in Section 5.2.2.6 about the Quality-of-Service function we defined for the CRIS application.

The structure of this section is as follows. Section 5.2.2.1 presents the traditional Fourier Transform Spectrometer (FTS). Section 5.2.2.2 talks in greater detail about CIRIS. Section 5.2.2.3 describes the CIRIS controller. Section 5.2.2.4 evaluates the implementation and compares it with a commercial FTS. Throughout Section 5.2.2 we present the implementation developed at JPL. In Section 5.2.2.5 we present the task model we propose for integration with MESUR on one processor. This task model has several changes compared to the FPGA implementation, the main difference being that the FFT is performed on the processor, and not on the FPGA.

5.2.2.1 Fourier Transform Spectrometry

The schematic of a traditional Michelson FTIR spectrometer is shown in Fig. 5.2. The basic configuration is comprised of a beam splitter and two mirrors, with the plane of the fixed mirror M_f perpendicular on the plane of the moving mirror M_m. As the beam of light passes through the beam splitter, it is divided into two separate beams, b_1 and b_2. The reflected beam b_1 travels a fixed distance. The transmitted beam b_2 is reflected by the moving mirror M_m, and thus travels a variable distance, depending on the position of M_m. The beams recombine at the beam splitter and the intensity of the recombined beam b_r is detected by the Detector.

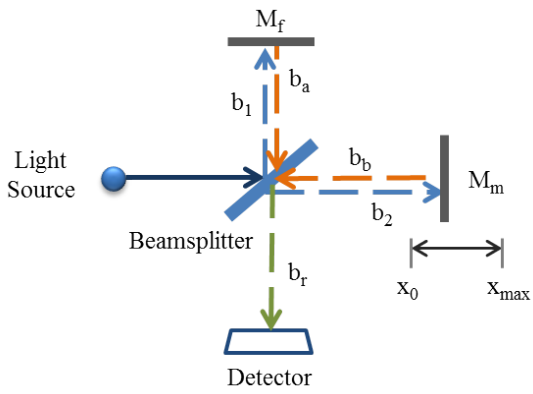


Figure 5.2: Basic Michelson interferometer

M_m moves in a linear trajectory between positions x_0 and x_{max}. When M_m is in the x_0 position, the distance travelled by beams b_1 and b_2 is equal. In this case, the optical path difference (OPD) between the two beams, i.e., the difference in the distance travelled by the beams, is zero. Consequently, constructive interference occurs as the two beams recombine at the beam splitter, and the intensity of the recombined beam b_r at the detector is maximized.

As the mirror M_m linearly moves away from the x_0 position, the OPD increases and a *phase shift* is introduced between the two beams. Thus, at each position of M_m, the recombined beam contains a different combination of wavelengths, and its intensity varies. The recorded intensity of the recombined beam, as a function of the OPD is called an *interferogram*. By processing the interferogram using a Fast Fourier Transform (FFT), we obtain the spectrum of the input beam, as the intensity of each wavelength.

According to the Reyligh criterion [87], the resolution $\Delta(\nu)$ of the spectrometer is determined by the maximum motion of the moving mirror (which in turn determines the OPD):

$$\Delta(\nu) = 1/max(OPD) \tag{5.1}$$

The reader is directed to [145, 87] for more details on the subject of Fourier Transform spectroscopy instrumentation and engineering.

5.2.2.2 Compositional InfraRed Imaging Spectrometer (CIRIS)

In the case of the CIRIS instrument (see Fig. 5.3), the OPD between the two beams is modified using a rotating refractor, instead of a linear moving mirror like in the traditional Michelson FTS. The refractor spins at a constant velocity, thus the OPD can be easily determined for each refractor angle. The OPD is zero when the refractor's plane is parallel or perpendicular on the beam splitter, that is, both beams travel at an angle of 45° through the refractor, and thus both beams travel the same length. The OPD has a maximum value when the refractor is perpendicular on one of the beams, and thus, one beam's path is maximized, while the other one's is minimized. Since during each revolution the refractor has four positions with zero OPD (ZPD), each complete refractor revolution yields 4 interferograms.

The linear motion system of the Michelson interferometer presented in Section 5.2.2.1 is very sensitive to vibrations and non-linear errors. An incorrect angle of the moving mirror causes optical path length errors [145], affecting the quality of the output spectra. Moreover, a monochromatic reference laser is usually used as a sampling clock signal, adding to the complexity of the spectrometer and can lead to data sampling errors.

The rotating refractor design of the CIRIS instrument increases the robustness of the FTS, reducing the alignment errors. Furthermore, this design eliminates the need for a reference laser, as the position of the refractor can be accurately reported using an optical encoder mounted on the DC servomotor controlling the refractor. The TurboFT spectrometer, on which CIRIS bases its design, was tested aboard helicopters for remote sensing applications in Australia, confirming the ruggedness of this design [177].

To obtain a non-distorted interferogram, the light is acquired in the optical region where the OPD is a linear relation with the angle of the refractor. The linear region corresponds to a scan angle limited to ±15 degrees around the ZPD position [177] and to a duty cycle of 33%. The rotation speed of the refractor is limited by frequency

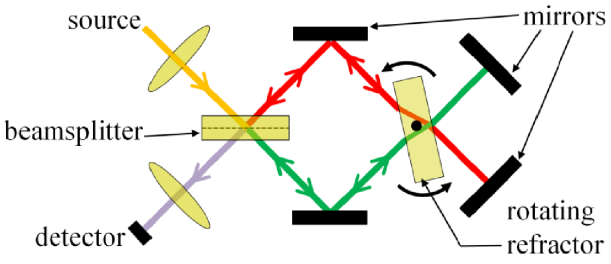


Figure 5.3: CIRIS interferometer (from [45])

bandwidth (50 kHz) of a very low noise ($1fA/\sqrt{Hz}$) high gain (10^8) transimpedance preamplifier needed to observe icy moons. In summary, the TurboFT spectrometer has an angular speed of 2.5 revolutions per second, and a single interferogram is captured during 33 ms every 100 ms.

The CIRIS instrument is operational in the spectral range of 2.8 to 18 μm, or 3571 to 555 cm^{-1}. The 4 cm^{-1} resolution is limited by the aperture of the instrument and the refractor thickness, while the optical bandwidth is limited by the scan angle. The FTS spectra has a resolution of 754 points between 3571 and 555 cm^{-1} (2.8 and 18 μm). The spectra can be computed from a single sided interferogram with 1508 points or from a double sided interferogram (insensitive to phase change), with at least 3016 points. While 4096 interferograms points acquired over 33 ms will be sufficient, our CIRIS implementation records 8192 interferograms data points for each interferogram. As a consequence, we add more data points at the short wavelength of the spectrum, while the interesting part of the spectrum is further away from the Nyquist frequency, compared to the spectrum obtained from 4096 points. This improves the anti-aliasing of the signal. Considering that the scan period is of 33 ms, the sampling frequency is set to 4 μs per interferogram data point.

More details on the spectrometer can be found in [177]. Researchers presented in [45] the testing results of the prototype, together with several detectors.

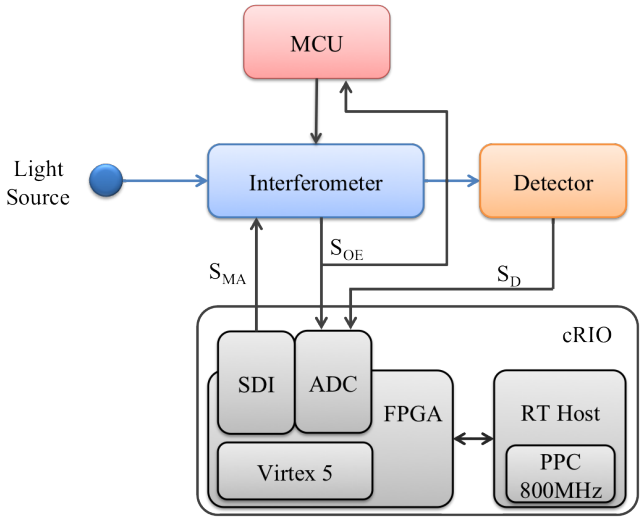
5.2.2.3 CIRIS Controller Implementation

Considering the real-time requirements of the CIRIS instruments presented in the previous Section, as well as the environment it will operate in, we implemented the controller on a CompactRIO (cRIO) platform from National Instruments (NI). cRIO is a "small rugged control and acquisition system" [20] for industrial use. A cRIO platform contains a cRIO 9025 controller module processor running a real-time operating system (RTOS), a back-plane cRIO 9118 with a reconfigurable, user-programmable FPGA and hot-swappable I/O modules such as NI 9223 with 4 ADC channels 16-Bit 1 MSamples/s[1] and NI 9263 4-Channel DAC 16-Bit 100 kSamples/s[2].

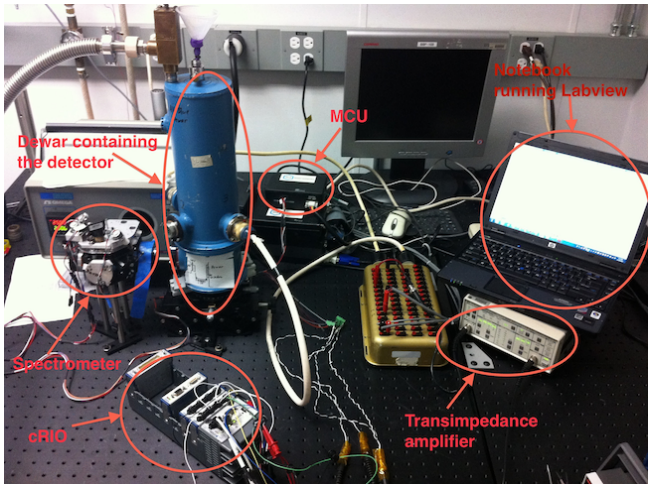
We implemented the CIRIS controller on a cRIO 9025 rugged controller, which contains an 800 MHz PowerPC processor running the VxWorks RTOS from WindRiver. The chassis is a NI 9118 Reconfigurable Embedded Chassis, containing a Xilinx Virtex-5 LX110 reconfigurable FPGA core, which executes at a default rate of 40 MHz.

Fig. 5.4a presents the schematic of our CIRIS setup, while Fig. 5.4b shows a photo of the physical setup in the lab. The cRIO 9025 controller is depicted in the figure

[1] A sampling frequency of 1 MHz.
[2] A sampling frequency of 100 KHz.



(a) CIRIS setup schematic



(b) Physical setup of CIRIS

Figure 5.4: CIRIS setup

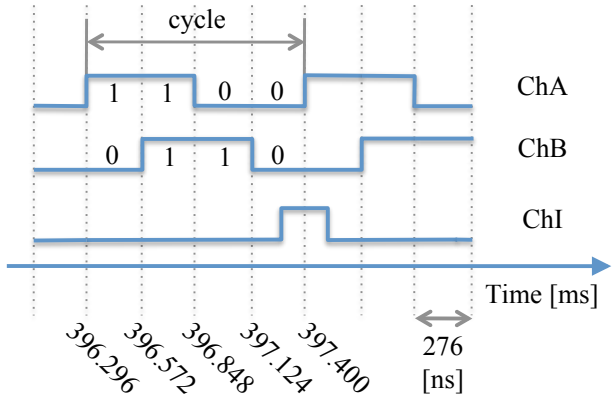


Figure 5.5: Optical encoder output channels logic states

with "RT Host" box, while the NI 9118 chassis is represented by the "FPGA" box. The Motor Control Unit (MCU) controls the velocity of the rotating refractor using the output signals of the optical encoder and is currently running in a full independent analog loop.

In order to meet the real-time requirement of reading an interferogram data point every 4 μs, we use a NI 9223 Simultaneous Analog Input module (depicted with ADC in Fig. 5.4a) to convert the detector signal SD reading. The NI 9223 is capable of simultaneously reading from its 4 channels, at a rate of 1 MSamples/s, or 1 Sample/μs, making it highly suitable for our application.

The correctness of the resulted spectrum depends not only on the noise of each point of the interferograms, but also on the proper data sampling of the interferogram. We use the angle of the rotating refractor to determine the sampling. In the CIRIS instrument, the angle of the refractor is signaled by a FAULHABER E2-360I [76] optical incremental encoder mounted on the DC servomotor controlling the rotating refractor. The E2-360I optical encoder completes 360 cycles during a revolution. It employs three outputs: Channel A and Channel B, with 90° phase shift, encode the logic state of the cycle, while Channel Index signals the completion of a revolution. Fig. 5.5 presents the logic states of the output channels. These optical encoder signals S_{OE} are converted by the ADC and used as inputs in our application.

Due to external forces, mirror misalignment may occur during the lifetime of operation, affecting the quality of the resulting interferograms. Alignment correction is performed using the 4 mirrors of the interferometer. A NI 9512 Stepper Drive Interface (SDI) Module, depicted in Figure 3, controls the stepper motor linear actuators connected to each mirror, through the mirror alignment signal S_{MA}. The optical alignment is

achieved by maximizing the power at the detector for a zero path delay configuration of the refractor.

Fig. 5.6 presents the high-level description of our acquisition and processing controller algorithm. This algorithm is partly implemented on the FPGA, and partly on the Real-Time Host (RT Host). We mark the number of the algorithm steps in green circles. In the first step, the controller identifies the rotating refractor position (1), by using the optical encoder signals from channels A, B and I, respectively. In case the refractor is at -15° from ZPD of one of the 4 rotational positions, the controller starts sampling for 8192 data points. The read data is filtered (3) using a bandpass filter between 3 and 100 kHz. After the ZPD position is identified (4), the interferogram can be zero-centered (5). Steps 1 to 5 compose the Interferogram Acquisition Process (IAP). The centered interferogram is Fast Fourier Transformed (FFT) (6), resulting in a raw spectrum. This spectrum is handed over to the RT Host, where it is further processed separately for each rotational position, with rotational position dependant coefficients. For each of the rotational positions (7), the spectra are averaged per position (8), dispersion corrected in the wave number domain (9) and the amplitude of the spectra is calibrated to spectral radiance in $W/(m^2 \times \mu m)$ (11). Finally, the average of the resulting spectra is computed (13). These steps are described in greater detail below.

Interferogram Acquisition. The interferogram acquisition process uses the optical encoder outputs Channel A and Channel B (ChA ChB) to identify the angle of rotating refractor. ChA ChB output signals each have 360 cycles per revolution, and they generate 1440 states per revolution. Fig. 5.5 presents the logic states of the output channels. Each state corresponds to a turn of 0.25° angle of the rotating refractor. An interferogram covering approximately 30° spans over 120 ChA ChB states. Once the IAP identifies that the current ChA ChB logic state corresponds to the first logic state of the 120 covered by an interferogram (step 1 in Fig. 5.6), it starts sampling (step 2) for 8192 points. IAP samples using an FPGA hardware clock with a frequency of 250 kHz. The Channel I encoder output signals the complete turn of the refractor, and triggers the reset of the sampling counters.

The read data is filtered using a band pass filter between 3 kHz and 100 kHz (step 3 in Fig. 5.6). As a state of the encoder covers around 69 samples (we are sampling the interferogram for 8192 points over 120 logic state numbers) and to get the double interferogram as symmetric as possible, the IAP centers the ZPD of the interferogram before performing the FFT. This is done first by identifying the ZPD (step 4 in Fig. 5.6) in the filtered data and then choosing 8192 points around the ZPD to obtain a centered interferogram (step 5), ready for the FFT (step 6 in Fig. 5.6). Fig. 5.7a presents the ZPD centered interferogram reading at rotational position 1.

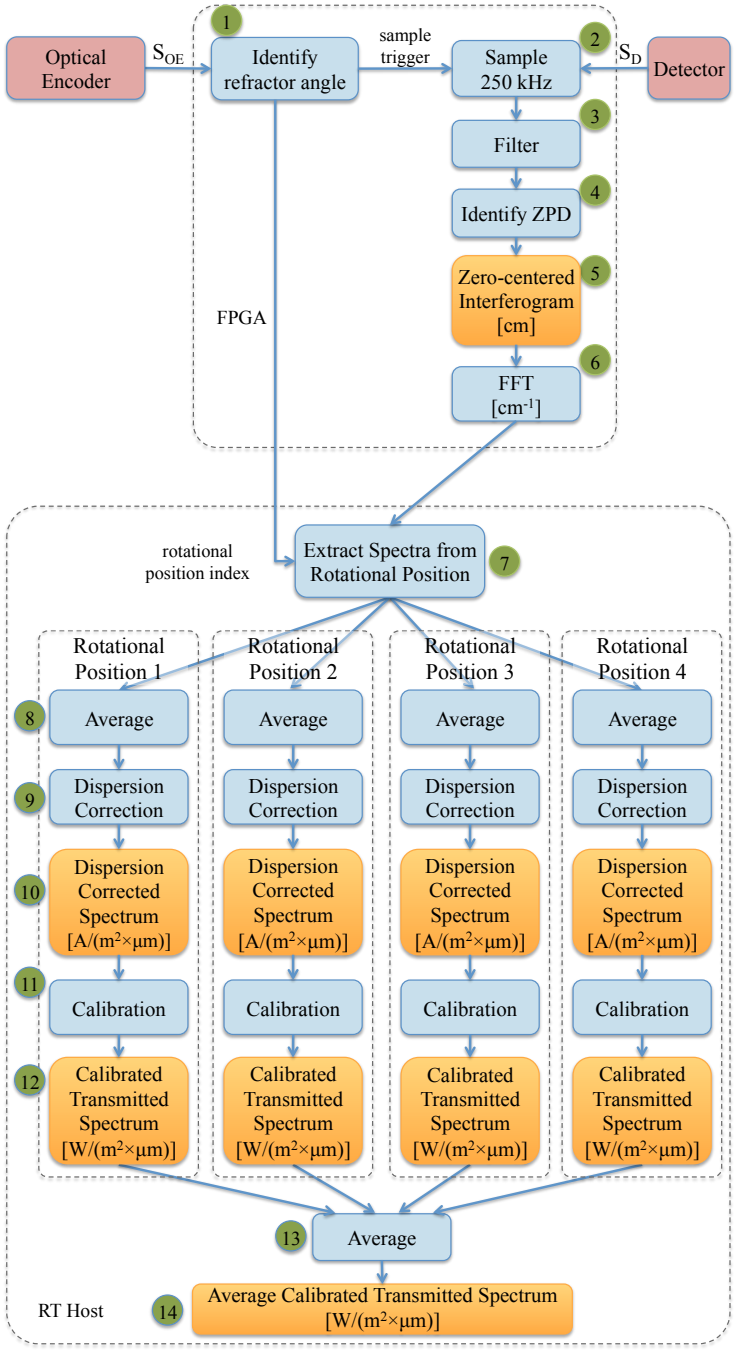
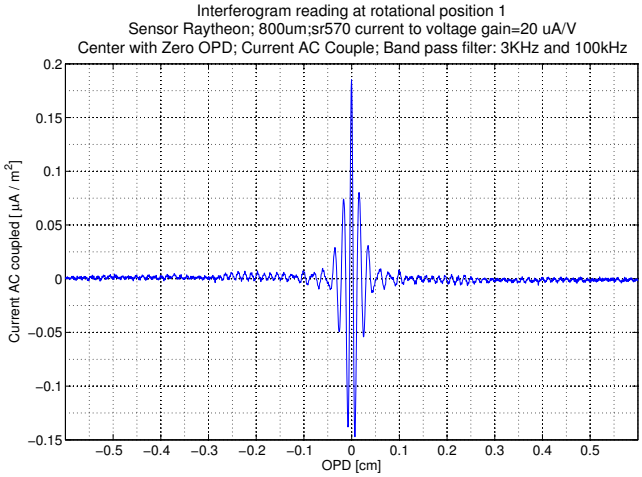
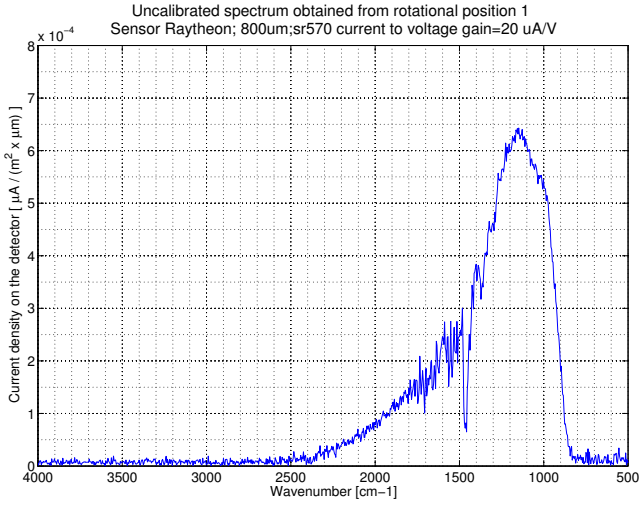


Figure 5.6: CIRIS high-level acquisition and processing algorithm description



(a) Interferogram reading at rotational position 1



(b) Uncalibrated spectrum obtained from rotational position 1

Figure 5.7: Interferogram and resulting spectrum

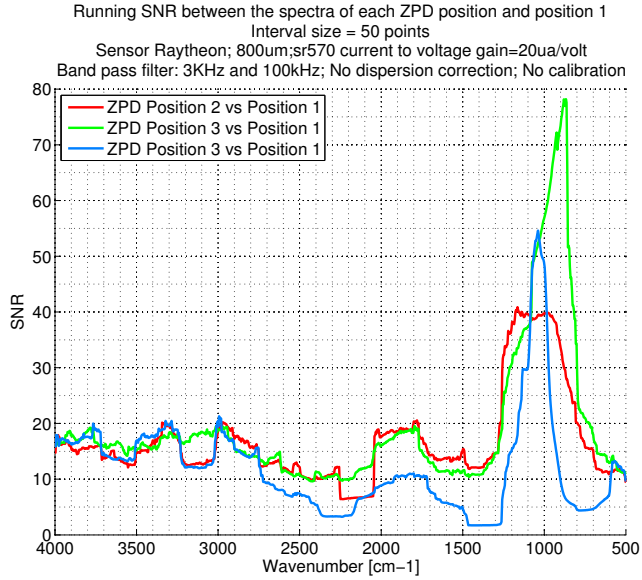


Figure 5.8: Running SNR comparison between the spectra at different ZPD positions

Position-Dependent Processing. Fig. 5.7b presents the uncalibrated spectrum obtained after the FFT of the interferogram in Fig. 5.7a. The spectra obtained after the FFT on the FPGA (step 6 in Fig. 5.6) are further processed on the RT Host, which has a dedicated floating point unit. This processing is refractor position specific, as the path delay of the beam is not exactly the same for each refractor position (see Fig. 5.3). Fig. 5.8 shows that there are differences between the spectra obtained at different rotational positions, by using the running signal to noise ratio (SNR), with an interval of 50 points (see Section 5.2.2.8 for more details on the computation of the running SNR). As such, the controller averages the resulted spectra, per each rotational position (step 8 in Fig. 5.6) and processes the averaged spectra with position specific coefficients.

The dispersion correction process (step 9 in Fig. 5.6) improves the wavenumber scale generated by the FFT, taking into consideration the rotational position of the spectra. The correction is an offset in wavenumbers (cm^{-1}), to the linear scale generated from the FFT. The form of the offset is computed using the equation below:

$$X_{corrected} = X_{fft} + c + 10^{m \times X_{fft} + b} \tag{5.2}$$

For ZnSe optics, m = 0.000172 and b = 0.993, while c is rotational position specific, and can be easily obtained by comparing the CIRIS spectra for each rotational position with a reference spectrum.

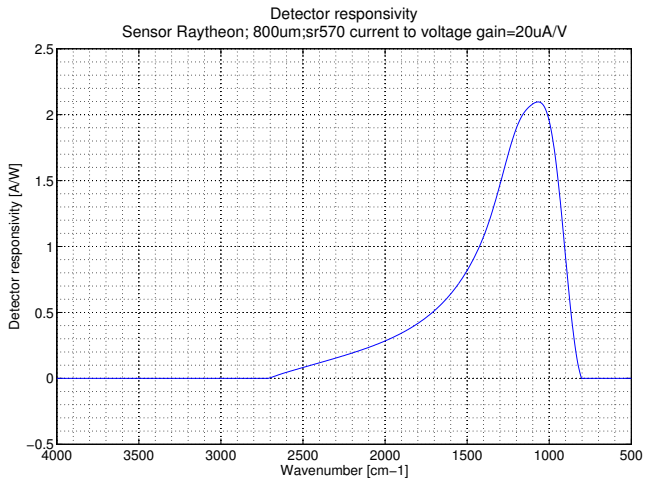


Figure 5.9: Detector responsivity

Amplitude Calibration. In order to provide consistent results over different measurements, the spectral amplitude has to be calibrated. This is done by reading the spectra corresponding to a blackbody at low and high temperature T_L and T_H, respectively, for each rotational position. We denote with S(T, ν) the measured spectrum of the blackbody at temperature T in function of the wavenumber ν. Next, we compute the theoretical spectral radiance corresponding to the two temperatures, using Planck's law. We denote with B(T, ν) the spectral radiance at the surface of the blackbody at the temperature T for wavenumber ν. For the measured spectrum S(T, ν), we compute the calibrated reading $T_{Calibrated}$ using the equation below:

$$T_{Calibrated}(T,\nu) = \frac{S(T,\nu) - S(T_H,\nu)}{S(T_L,\nu) - S(T_H,\nu)} \times (B(T_L,\nu) - B(T_H,\nu)) + B(T_H,\nu) \tag{5.3}$$

This equation can be rewritten as:

$$T_{Calibrated}(T,\nu) = S(T,\nu) \times \frac{1}{Responsivity(\nu)} + Offset(\nu) \tag{5.4}$$

The responsivity of the detector Responsivity(ν) and the Offset(ν):

$$Responsivity(\nu) = \frac{S(T_L,\nu) - S(T_H,\nu)}{B(T_L,\nu) - B(T_H,\nu)} \tag{5.5}$$

$$Offset(\nu) = S(T_H,\nu) \times \frac{B(T_L,\nu) - B(T_H,\nu)}{S(T_L,\nu) - S(T_H,\nu)} + B(T_H,\nu) \tag{5.6}$$

The values for Responsivity and Offset are computed a priori of performing the calibration presented in step 11 in Fig. 5.6. Fig. 5.9 presents the detector responsivity computed using T_L = 338.7 K and T_H = 422 K.

Transmittance and Absorbance. The calibrated transmitted spectrum (step 14 in Fig. 5.6) is obtained by averaging the calibrated spectra (step 12) over the four rotational positions (step 13). The transmittance T(ν) of a sample at wavenumber ν is measured by computing the ratio of the sample transmitted spectrum $T_{Sample}(\nu)$ over the background transmitted spectrum $T_{Background}(\nu)$. The background spectrum is obtained by measuring the spectrum without sample.

$$T(\nu) = \frac{T_{Sample}(\nu)}{T_{Background}(\nu)} \tag{5.7}$$

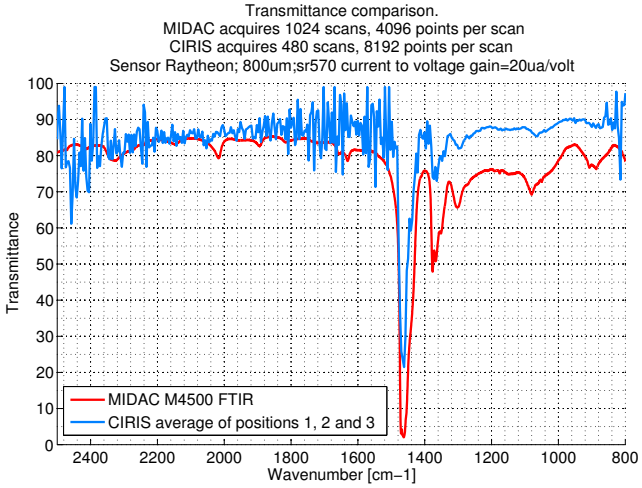
The absorbance of A(ν) of a sample at wavenumber ν is computed using the equation below.

$$A(\nu) = -log10(T(\nu)) \tag{5.8}$$

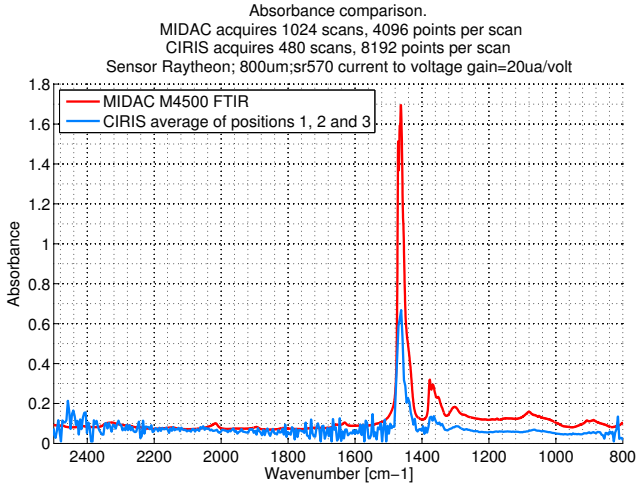
5.2.2.4 Evaluation of CIRIS

We evaluate the quality of our implementation by comparing the results from the CIRIS instrument with the results obtained from a MIDAC M4500 FTIR spectrometer. The MIDAC spectrometer uses ZnSe optics with HgCdTe detector. It has a resolution of 4 cm^{-1}, covering wavenumbers from 6000 to 600 cm^{-1}. Fig. 5.10a and Fig. 5.10b present the transmittance and absorbance, respectively, of a plastic sample obtained with the CIRIS instrument and with the Michelson-based FTIR instrument (MIDAC M4500). The spectral features are similar.

The MIDAC absorbance is smoother due to the differences in data acquisition and processing: the MIDAC instrument acquires 4096 points per interferogram and average over 1024 spectra, while the CIRIS instrument acquired 8192 points per interferogram and average over 480 spectra. In addition, the MIDAC spectrometer performs triangle apodization of the interferogram [145], and Mertz phase correction [115]. The difference in the amplitude of the absorbance peak between the two instruments is attributed to the black body calibration of the CIRIS instrument and the different experimental setup of the CIRIS instrument compared to the MIDAC.

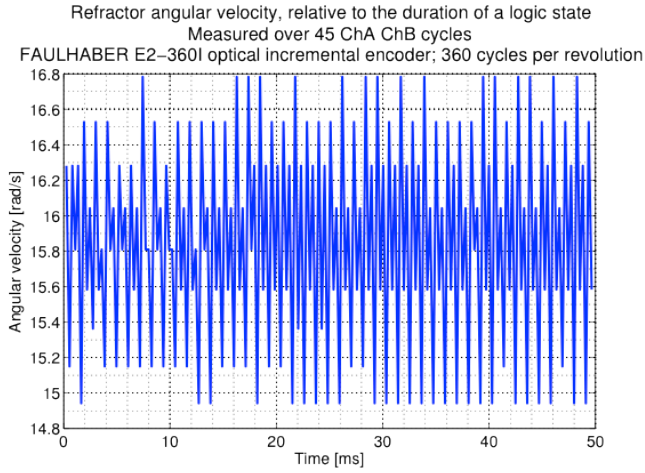


(a) Transmittance comparison

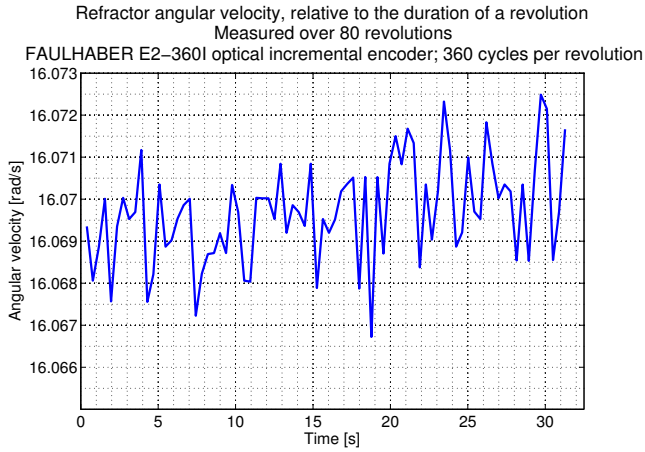


(b) Absorbance comparison

Figure 5.10: Comparison between the results obtained with CIRIS and with MIDAC M4500 FITR

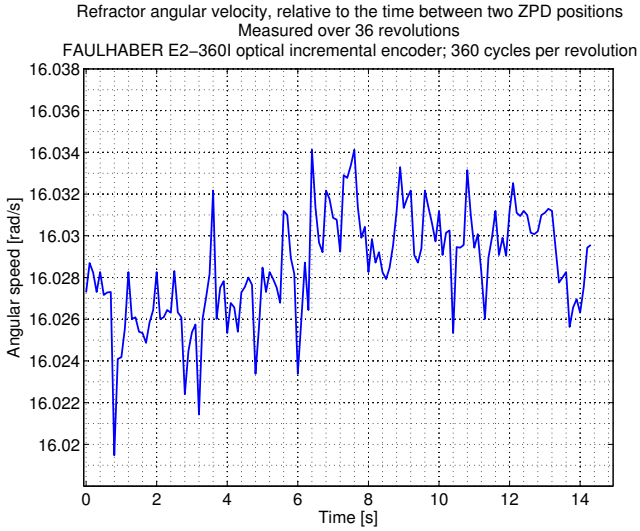


(a) Angular velocity variation, relative to the duration of a ChA ChB logic state



(b) Angular velocity variation, relative to the duration of a revolution, as reported by ChI

Figure 5.11: Angular velocity variation



(c) Angular velocity variation, relative to the duration of a revolution, as reported by the ZPD for each rotational position

Figure 5.11: Angular velocity variation

Table 5.2: Rotating refractor velocity mean and standard deviation

Frequency [Hz]	Mean velocity [rad/s]	Standard deviation [rad/s]
3600	15.834	0.606000
10	16.028	0.003611
4	16.069	0.001211

We also identified points to improve in the instruments. In the current setup, the motor control unit (MCU) is considered as a separate application, implemented on an analog board. Fig. 5.11a presents the rotating refractor angular velocity measured by the duration between two logic codes defined by a ChA ChB. Fig. 5.11b presents the angular velocity variation, relative to the duration of a complete revolution, as reported by ChI. Fig. 5.11c reports the velocity variation measured between two consecutive ZPD peaks.

Table 5.2 summarizes the results in Fig. 5.11, and shows the mean and the standard deviation of the refractor velocity measured at 3600 Hz using ChA ChB signals, at 2.5 Hz using the ChI signal and at 10 Hz using the ZPD.

As shown in Table 5.3, the encoder signals ChA and ChB measure accurately the mechanical position of the refractor. The table presents the ChA ChB logic state number for each of the four ZPD positions over 40 revolutions. Each of the four ZPD positions are located at the same logic state number. Thus, replacing the analog implementa-

Table 5.3: Logic state numbers of the ZPD positions

ZPD position	Mean value of ChA ChB logic state number	Standard deviation
1	233	0
2	595	0
3	956	0
4	1315	0

tion of MCU with a digital one will improve the acquisition process, and the resulting spectra, consequently.

5.2.2.5 Controller Application Model for Integration with MESUR

The CIRIS application, although non-critical, has soft real-time constraints: the interferometer produces every 100 ms an interferogram that has to be acquired and processed before the next 100 ms interval. We present next the task model for the CIRIS controller, for the integration scenario with the MESUR application (see Section 5.2.1), onto the same processor. In this scenario, we consider the FFT tasks executing on the processor, and not on the FPGA. Furthermore, CIRIS acquires interferograms over 40 revolutions, totaling 160 samples.

The FFT tasks are marked with fft_i, with i from 1 to 160. Tasks avg_j, dc_j and cal_j perform rotational-specific processing: they process the spectra obtained from the rotational position j, with $j = 1..4$. Task avg_j performs the averaging of all the spectra obtained from rotational position j, and is marked as step 8 in Fig 5.6. Task dc_j performs the dispersion correction for the averaged spectrum at position j (step 9 in Fig 5.6), while task cal_j performs the calibration (step 11 in Fig. 5.6). The wR task saves the resulting spectrum to the system.

The CIRIS application does not have a global deadline. Instead, the FFT tasks fft_i, $i = 1..160$, have individual soft deadlines of 100 ms after their release times. The details for the CIRIS tasks are shown in Table 5.4. Since tasks avg_1–avg_4 have the same characteristics, we grouped them in Table 5.4 as avg_j. We did the same for the FFT tasks, the dispersion correction tasks dc_j and the calibration tasks cal_j. We mark with "—" if the characteristic is not applicable, e.g., this is the case for dc_j, which does not have a release time or a deadline.

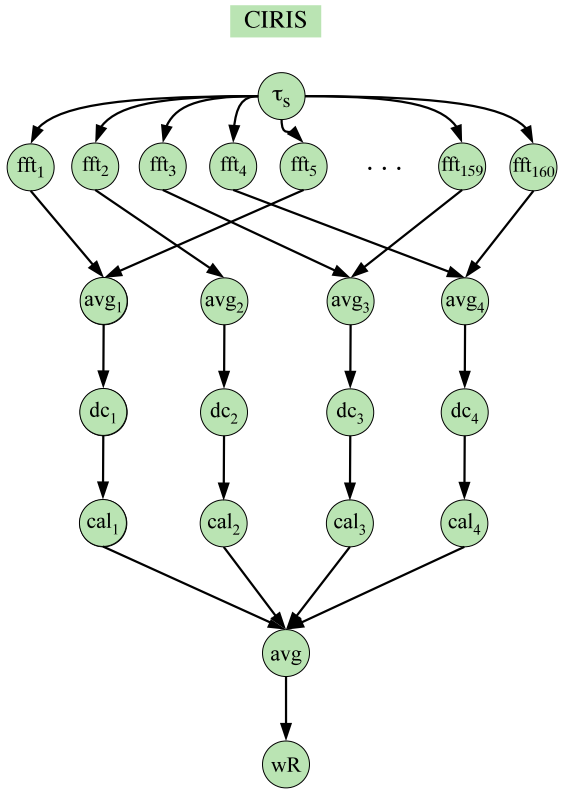


Figure 5.12: CIRIS task graph

Table 5.4: CIRIS task details

Task name	Release time	Deadline	WCET
τ_s	—	—	2
fft_i	$(i-1)\times 100$	$i\times 100$	10
avg_j	—	—	30
dc_j	—	—	5
cal_j	—	—	10
avg	—	—	30
wR	—	—	100

5.2.2.6 Quality of Service (QoS)

CIRIS is a soft real-time application (if CIRIS misses a deadline, the application will still function, but with degraded service). Therefore we do not use the degree of schedulability metric (presented in Section 3.3) to characterize the application implementation. Instead, we define a QoS function. The QoS function for CIRIS is the ratio between the number of FFT tasks executed and the total number of FFT tasks. Thus, a QoS of 1 means that all the FFT tasks executed. A QoS of 0 means that none executed. The QoS is shown in Eq. 5.9:

$$QoS(CIRIS) = \frac{executed\ FFT\ tasks}{total\ FFT\ tasks} \tag{5.9}$$

Section 5.2.2.7 presents the impact of partitioning on the QoS.

5.2.2.7 Influence of Partitioning on QoS

The proposed implementation scenario is that CIRIS is integrated on the same processor with MESUR or other applications of different criticality levels. In case this architecture is not partitioned, all applications need to be developed and certified according to the same standards and processes as the applications with the highest criticality level. This will increase the development and integration costs of the CIRIS instrument. The integration of the controller for the CIRIS instrument on an unpartitioned architecture will affect the signal to noise performance of the instrument, as the real-time requirements may not be met.

The signal to noise performance of the instrument is degraded if the real-time requirements of the controller are not achieved, as the measurements have to be interrupted to

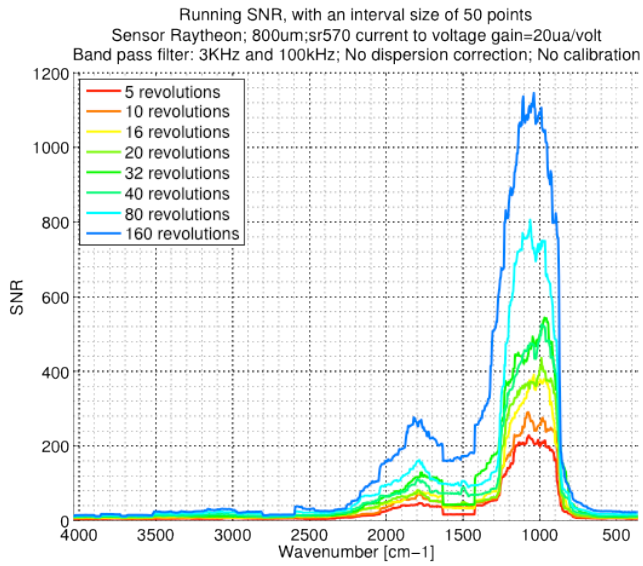


Figure 5.13: Comparison of running SNR of an interval size of 50 points, over different number of revolutions

handle other applications with higher priority. For example, let us assume the CIRIS spectrometer is aboard a satellite making spectral measurements of a particular spot on the surface of an icy moon. In this case, the instrument will need to take continuous measurements of the same spot for a number of scans to increase the SNR. Moreover, these measurements will need to be continuously processed. A usual number of required scans to average is in the range of a several hundreds, thus the interferogram acquisition and processing spans over tens of seconds. When the applications handling the acquisition and processing of the spectra share the computing resource, there are situations when mission-critical applications (navigation and power management) might unnecessarily monopolize the CPU, preventing the CIRIS controller from executing. In this case, the number of processed spectra will be reduced, severely affecting the signal to noise of the resulting spectra.

Fig. 5.13 shows the impact of the number of spectral scans used during averaging processed over the SNR of the final spectrum. We compute the running SNR with an interval size of 50 points. This figure shows that reducing the number of spectral scans by 32 reduces the SNR of the final spectrum by 6, as expected for white noise.

Considering the QoS function defined in Section 5.2.2.6, the running SNR corresponding to the spectrum obtained after 160 revolutions (640 measurements) corresponds to a QoS of 1. In comparison, the running SNR corresponding to the spectrum obtained

after 5 revolutions (20 measurements) corresponds to a QoS of 0.03125. Thus, a higher QoS value means a better SNR and a signal of higher quality.

5.2.2.8 Running Signal-to-Noise Ratio (SNR)

SNR measures "the ability to reproduce the spectrum from the same sample and the same conditions" [1]. Thus, SNR is a measure of the signal quality. We compute the SNR according to the formulas presented [1]:

$$SNR = \frac{1}{N_{rms}} \tag{5.10}$$

using the root-mean-square N_{rms} of the spectral noise $N(\nu)$:

$$SNR = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} [N(\nu_i)]^2} \tag{5.11}$$

where n is the number of wavelengths in the spectrum. The spectral noise between two spectra T_a and T_b of the same sample measured at different times is computed according to the following equation:

$$N(\nu) = 1 - \frac{T_a(\nu)}{T_b(\nu)} \tag{5.12}$$

With the running SNR, we compute the SNR over an interval of a given number of points of the two output spectra. We shift this interval one point at the time, to cover the whole spectral interval.

We present in Section 5.4.1 how to extend our proposed "Mixed-Criticality Design Optimization" algorithm (see Section 3.2.2) to take into account soft real-time applications. We evaluate this extension using the MESUR application, presented in Section 5.2.1, and a modified version of the CIRIS controller, described in Section 5.2.2. Section 5.4.1 also presents the results of our experimental evaluation.

5.3 Communication-Level Partitioning

The Orion Crew Exploration Vehicle (CEV) was planned to replace the Space Shuttle. Orion was part of the Constellation Program, implementing the Vision for Space Exploration [5] plan announced in 2004. The Orion project was awarded to Lockheed

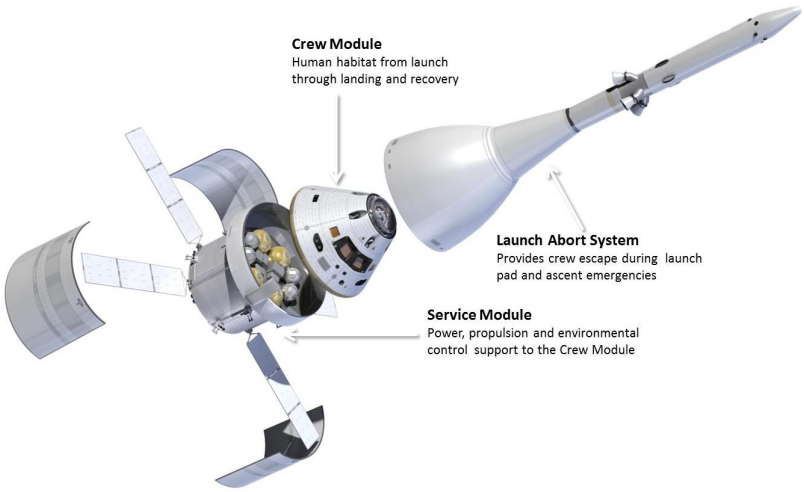


Figure 5.14: Orion major elements (from [121])

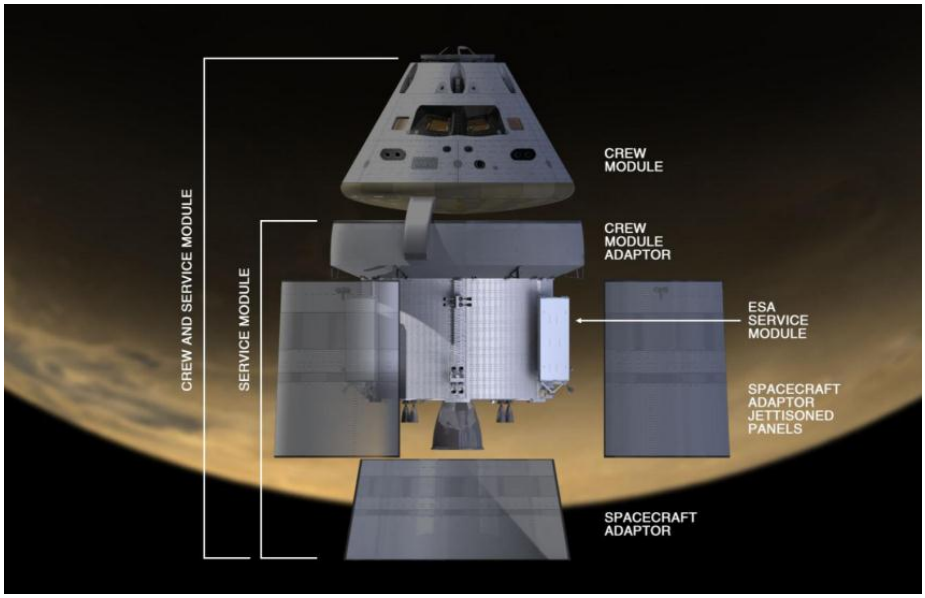


Figure 5.15: Location of ESM on Orion (from [121])

Martin in 2006. The Constellation Program was cancelled in 2010 due to budget constraints, however Orion was reformulated as the Orion Multi-Purpose Crew Vehicle (MPCV) [121].

Initially, Orion CEV was planned for lunar return missions, travel to the International Space Station (ISS) and be extensible for future Mars missions. Accordingly, the requirements for Orion CEV were: transport two to four people to the moon and zero up to six to the ISS, land anywhere on the moon, provide continuous habitat for up to 6 months and enable "any time" return [152]. Compared to Orion CEV the design requirements, the Orion MPCV does not have major changes. However, Orion MPCV will be used also for Beyond Earth Orbit (BEO) missions [121]. See [152] for a more detailed description of the Orion CEV design requirements, and [121] for the project status of the Orion MPCV in 2013.

Orion MPCV has no major design changes, compared to Orion CEV. Orion consists of four major elements, displayed in Fig. 5.14: a launch abort system (LAS), a crew module (CM), a service module (SM) and a spacecraft adapter (SCA). The LAS provides safety-critical functionality to the crew in case of malfunctioning on the pad or during ascent, by detaching the CM from the SM and the launch vehicle [152]. The CM is a reusable component that holds four people, can transport pressurized and unpressurized cargo, can dock with other vehicles and can perform re-entry and landing [152]. The SM contains life support systems, and delivers propulsion and power to the CM and is discarded before re-entry. In addition, SM can also transport scientific payload. In 2013 NASA announced that the European Space Agency (ESA) will provide a European SM (ESM) to be integrated with the crew adapter module and SCA [121]. Fig. 5.15 shows the location of the ESM on Orion.

The Orion case study used in this chapter is derived from [126], based on the Orion Crew Exploration Vehicle (CEV), 606E baseline [126]. In this case study we have 45 network nodes (ESes and NSes) and 187 messages (with the parameters generated based on the messages presented in [126]). The topology for this case study is shown in Fig. 5.18.

5.4 Evaluation

5.4.1 Processor-Level Partitioning

In Chapter 3 we proposed design optimization strategies for mixed-criticality applications, aiming at minimizing the worst-case response times and the development costs. In the following, we propose a new cost function (Eq. 5.13) for our proposed "Mixed-Criticality Design Optimization" (MCDO) algorithm presented in Section 3.2.2. This

new cost function captures, besides the schedulability of hard real-time applications, also the quality of service of the soft real-time applications. For this setup we consider two applications, the MESUR application presented in Section 5.2.1 as the hard real-time application, and the CIRIS application (Section 5.2.2) as the soft real-time application. These application also differ in their criticality levels. MESUR is mixed-criticality, containing high-criticality and low-criticality tasks, while CIRIS is a non-critical application.

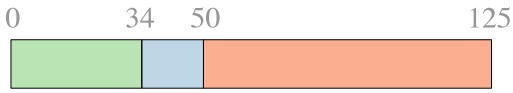
In Section 5.2.2.7 we presented how the number of the acquired and processed interferograms affects the signal to noise performance of the instrument, which is a measure of the quality of service (QoS). In Section 5.2.2, CIRIS was implemented on a dedicated processor. Here, we assume that CIRIS shares the processor with the MESUR application. Given a fixed assignment of tasks to partitions, we are interested to find the sequence and size of partition slots such that all MESUR tasks are schedulable and the QoS for CIRIS is maximized.

The MCDO algorithm proposed in Section 3.2.2 can be modified to search for the solution in which the hard real-time applications are schedulable, and the soft real-time applications have their quality of service maximized. This can be achieved by changing the cost function (Eq. 3.2 in Section 3.2.2) to the following:

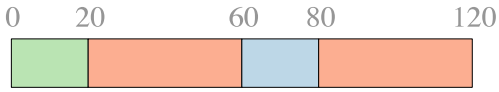
$$Cost(\psi) = \begin{cases} c_1 = \sum_{\mathcal{A}_i \in \Gamma} \max(0, R_i - D_i) & \text{if} c_1 > 0 \\ c_2 = -\sum_{\mathcal{A}_j \in \Gamma_{QoS}} QoS(A_j) & \text{if} c_1 = 0 \end{cases} \tag{5.13}$$

If at least one hard real-time application $\mathcal{A}_i$ from the set Γ is not schedulable, there exists one R_i greater than the deadline D_i, and therefore the term c_1 will be positive. The term c_1 in the equation above is the same in Eq. 3.2, and drives the search towards schedulable solutions. If all the hard real-time applications in Γ are schedulable, this means that each R_i is smaller than D_i, and the term $c_1 = 0$. In this case, we use c_2 as the cost function, since when the hard real-time applications are schedulable, we are interested to maximize the QoS for the soft real-time applications, denoted with Γ_{QoS}. We denote with $QoS(A_j)$ the quality of service function for application A_j. This function is specific for each application. In case of the CIRIS application, this function is defined in Eq. 5.9.

In the following, we will show several partition table configurations visited by MCDO, including the final solution, and their effect on the soft real-time CIRIS application. Fig 5.16 presents the partition tables as Gantt charts, with the green partitions corresponding to CIRIS. Section 5.2.1 and Table 5.1 present the MESUR tasks. We refer with MESUR-HC to the set of high-criticality tasks (Set 1 in Table 5.1). The MESUR-HC tasks are assigned to the red partition slices in Fig. 5.16. The set of low-criticality MESUR tasks (Set 2 in Table 5.1), is denoted with MESUR-LC and is assigned to the blue slices in Fig. 5.16.

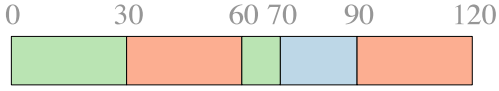


(a) Initial solution. All the hard real-time tasks are schedulable, but CIRIS processes only 4 out of 5 interferograms.

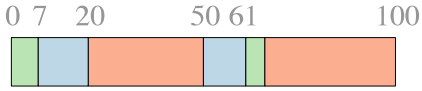


(b) The MESUR-HC tasks are schedulable. The CIRIS QoS increases to 0.837, but the MESUR-LC "Camera task" is not schedulable.

(c) All the MESUR tasks are schedulable, but the CIRIS QoS drops to 0.493.



(d) Although the CIRIS processes all interferograms, there are 2 MESUR-HC and 1 MESUR-LC tasks missing their deadlines.



(e) Final solution, all the tasks are schedulable.

Figure 5.16: Partition table configurations

Table 5.5 summarizes the impact of the partition tables on the applications considered. The first column, named "Partition table", indicates the partition table configuration used, and corresponds to the partition tables from Fig 5.16. Column 2 presents the size of the major frame for each configuration, while Columns 3, 4 and 5 show the sum of partition slice sizes assigned to the CIRIS, MESUR-HC and MESUR-LC tasks, respectively. Column 6 and 7 present the number of successfully acquired and processed interferograms and the value of the QoS function (Eq. 5.9 in Section 5.2.2.6) for the CIRIS application. Columns 8 and 9 show the number of schedulable MESUR-HC and MESUR-LC tasks.

The MCDO algorithm, presented in Section 3.2.2 starts from an initial solution which considers that all the tasks in the system are hard real-time. In this extension, we modify the initial solution algorithm to divide the processor time based on the priority of the applications. Moreover, we have added another design transformation, "resize major frame", which modifies the Major Frame size. This new move, increases or decreases the size of the major frame, proportionally adjusting the size of the partition slices. See Section 3.2.2 for the design transformations used by MCDO.

Table 5.5: Partition table configuration

Partition table	Major Frame size (ms)	Total partition slice size (ms)			CIRIS		MESUR Sched. task	
		CIRIS	MESUR-HC	MESUR-LC	No. of interferograms	QoS	MESUR-HC	MESUR-LC
a	125	34	75	16	128	0.800	4 / 4	3 /3
b	120	20	80	20	92	0.575	4 / 4	2 /3
c	200	10	160	30	79	0.493	4 / 4	3 /3
d	120	40	60	20	160	1.000	2 / 4	2 /3
e	100	12	64	24	160	1.000	4 / 4	3 /3

Fig. 5.16a presents the initial solution, computed according to the changes presented previously. The initial solution starts with a $T_{MF} = 125$ ms, given by the designer. Thus, the MESUR-HC tasks, composed of high-criticality hard real-time tasks, have their processor time assigned first. Since the MESUR-HC tasks have a processor utilization of 0.6, the tasks are assigned to a partition 75 ms long. Then, the MESUR-LC tasks, which are low-criticality hard real-time, have their processor time assigned. The MESUR-LC, with a utilization of only 0.125, are assigned to a partition 16 ms long. The rest of the processor time (34 ms) is assigned to the CIRIS tasks, which are non-critical soft real-time. This configuration is presented in Table 5.5, labelled as "a". In this case, all the hard real-time tasks are schedulable, but CIRIS has a QoS of only 0.8.

The second partition table, shown in Fig. 5.16b and described in Table 5.5 as the "b" configuration, has a $T_{MF} = 120$ ms. The MESUR-HC tasks are assigned to a partition composed of two partition slices, summing 80 ms processor time. The MESUR-LC partition is increased to 20 ms, but the CIRIS partition is reduced to 20 ms. In this configuration, all the high criticality tasks are schedulable, and the QoS for the CIRIS application increased to 0.837, but the low-criticality "Camera task" is not schedulable anymore. Considering the hard real-time tasks, this solution is worse than the initial solution.

Fig. 5.16c presents the "c" partition table from Table 5.5. This partition table has a size of 200 ms, and the processor time is distributed among the tasks as follows. MESUR-HC is assigned to two partition slices representing 160 ms of processing time. MESUR-LC is assigned to two partition slices totaling 30 ms, while CIRIS is assigned to a partition slice of 10 ms. In this configuration all the MESUR tasks are schedulable, but for the CIRIS application, only 79 interferograms (out of 160) are acquired and processed. The QoS for CIRIS is 0.493 in this case. With regards to the CIRIS tasks, this partition table is worse than the initial solution.

The "d" partition table from Table 5.5 is shown in Fig. 5.16d. CIRIS processes all the interferograms, resulting in a QoS of 1, which corresponds to a high-quality signal (see Section 5.2.2.7 for a discussion on the impact of the number of processed interferograms on the quality of the signal). In this case, not all the hard real-time tasks are schedulable. The MESUR-HC "Control task" and "Radio task", and the MESUR-LC "Camera task" miss their deadline. This is the worst solution found so far.

The final solution is presented in Fig. 5.16e, labelled with "e" in Table 5.5. In this case, the $T_{MF} = 100$ ms, and the MESUR-HC tasks are assigned to two partition slices summing 64 ms of processor time; the MESUR-LC tasks are assigned to two slices summing 24 ms, while the CIRIS tasks have 12 ms of processor time. In this case, all the hard real-time tasks are schedulable, and CIRIS processes all the interferograms, resulting in a high-quality result (QoS of 1).

This example demonstrates that our MCDO algorithm, presented in Section 3.2.2, is in fact an extensible framework that, with minor changes, can optimize systems according to different objectives. Thus, MCDO can be applied to a wide variety systems and design optimization problems.

5.4.2 Communication-Level Partitioning

The Design Optimization of TTEthernet-based Systems (DOTTS) strategy is presented in Section 4.2, Chapter 4. DOTTS is a Tabu Search-based metaheuristic that given the topology of the network and the set of messages, returns the packing of messages into frames, the assignment of frames to virtual links, the routing of virtual links, the bandwidth for each RC virtual link and the schedules for the TT frames. DOTTS is a flexible framework, that can be extended and modified to optimize different aspects of TTEthernet-based systems. In Section 5.4.2.1 we describe how we can modify DOTTS to perform topology selection in a TTEthernet-based system, and in Section 5.4.2.2 we show how to modify DOTTS to take into consideration best-effort traffic.

5.4.2.1 Topology Selection

In Chapter 4, we considered that the topology of the TTEthernet network is given. But designing the network topology for a system is not an easy task: it depends on the number of end systems, network switches and links, on the network traffic, but also on constraints like cost and SWaP (size, weight and power) of the system. In this section we show how DOTTS can be used to optimize the network topology of a TTEthernet-based system. Given an initial topology of the system, which can contain redundant links and network switches, and the set of messages in the system, we want to find an implementation that reduces the cost of the network implementation, such that all messages are schedulable. In this section, we compute the cost of the system as the number of network switches and links in the system:

$$Cost(\Upsilon) = \sum_i dl_i + \sum_j NS_j \tag{5.14}$$

We propose an iterative algorithm to solve this design problem, based on DOTTS. The algorithm is presented in Fig. 5.17. Starting with an initial topology (line 1 in Fig. 5.17), given by the system engineer, we run DOTTS to obtain a schedulable implementation (line 2). We denote with $\mathcal{NS}^\star$ the subset of NSes that, based on the embedded system engineer's decision, are fixed in the topology of the network and may not be removed. Similarly, $\mathcal{DL}^\star$ is the subset of dataflow links that may not be removed. We denote with

TopologySelection($\mathcal{G}_C^\circ$, $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$)

1 $\mathcal{G}_C^{Current} \leftarrow \mathcal{G}_C^\circ$
2 $\Upsilon^{Current} \leftarrow$ DOTTS($\mathcal{G}_C^\circ$, $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$)
3 **while** termination condition not reached **do**
4 $\quad \mathcal{G}_C^{New} \leftarrow \mathcal{G}_C^{Current} \setminus \{NS_i \vee dl_j | NS_i \in \mathcal{NS}^{Current} \setminus \mathcal{NS}^\star, dl_j \in \mathcal{DL}^{Current} \setminus \mathcal{DL}^\star\}$
5 $\quad \Upsilon^{New} \leftarrow$ DOTTS($\mathcal{G}_C^{New}$, $\mathcal{M}^{TT} \cup \mathcal{M}^{RC}$)
6 $\quad$ **if** Cost(Υ^{New}) < Cost($\Upsilon^{Current}$) **then**
7 $\quad\quad \mathcal{G}_C^{Current} \leftarrow \mathcal{G}_C^{New}$
8 $\quad\quad \Upsilon^{Current} \leftarrow \Upsilon^{New}$
9 $\quad$ **end if**
10 **end while**
11 **return** $< \mathcal{G}_C^{Current}, \Upsilon^{Current} >$

Figure 5.17: Topology Selection with DOTTS

Υ the tuple $< \Phi_m, \mathcal{K}, \mathcal{M}_F, \mathcal{R}_{VL}, \mathcal{B}, \mathcal{S} >$ obtained by running the DOTTS algorithm, where Φ_m is the fragmenting of messages, $\mathcal{K}$ is the packing of messages into frames, $\mathcal{M}_F$ is the assignment of frames to virtual links, $\mathcal{R}_{VL}$ is the routing of virtual links, $\mathcal{B}$ is the bandwidth allocation for the RC virtual links, and $\mathcal{S}$ is the set of schedules for the TT frames (see Section 4.2).

Next, we iteratively (1) modify the current network topology (line 4), by removing a network switch $NS_i \in \mathcal{NS}^{Current} \setminus \mathcal{NS}^\star$ or a dataflow link $dl_j \in \mathcal{DL}^{Current} \setminus \mathcal{DL}^\star$. We denote with $\mathcal{NS}^{Current}$ and $\mathcal{DL}^{Current}$ the set of NSes and dataflow links in $\mathcal{G}_C^{Current}$. Then (2) we rerun DOTTS with the new topology $\mathcal{G}_C^{New}$ to find a new schedulable implementation Υ^{New} (line 5). In case the cost of the new implementation Υ^{New} is smaller than of the current implementation $\Upsilon^{Current}$, in terms of the Eq. 5.14, we continue the search with the new topology (lines 6–9). The proposed topology selection algorithm can be applied iteratively, until a stopping condition is met. Our condition was a cost budget on the cost of the system, but this can be easily changed, allowing to tackle different optimization problems.

We evaluated the algorithm from Fig. 5.17 using the real-life case study Orion CEV, described in Section 5.3. The experimental results for this evaluation are presented in Table 5.6. For each topology, the number of ESes and messages is the same (columns 2 and 5). Column 3 presents the number of NSes, and Column 4 presents the cost of the system, in term of Eq. 5.14. Columns 6 and 7 show the number of resulting frame instances obtained after running DOTTS and the percentage of schedulable messages. The initial network topology, shown in Fig. 5.18, is marked in Table 5.6 with "orion 1". The Orion CEV case study has 31 ESes and 187 messages, with parameters generated based on the messages presented in [126].

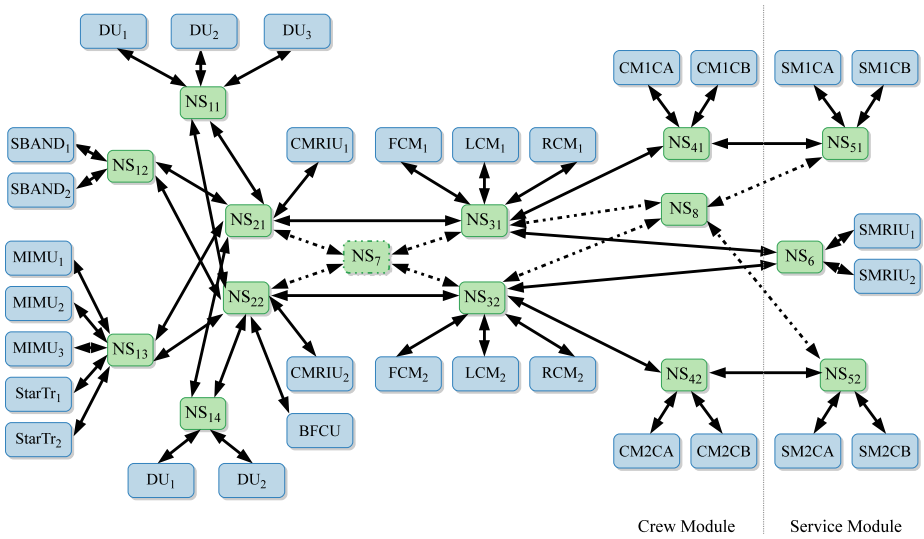


Figure 5.18: Network topology of the Orion CEV, derived from [126]

Table 5.6: Topology selection experimental results

Topology	ES	NS	Cost	Messages	Frame Instances	Sched.%
orion 1	31	14	124	187	6250	100
orion 2		13	115		7240	100
orion 3		12	106		8804	100

The initial topology "orion 1" has 14 NSes, with a cost of 124 (see Eq. 5.14). Running DOTTS, we obtain a schedulable implementation with 6,250 frame instances. In the first iteration of the algorithm from Fig. 5.17, the algorithm removes NS_8, and the connecting dataflow links, depicted with dotted lines. The results of the benchmark corresponding to the topology "orion 2" are presented in Table 5.6. DOTTS finds a schedulable implementation in this case as well. In the second iteration, the algorithm removes both NS_7 and NS_8 from the initial topology. In this case too, labelled with "orion 3" in Fig. 5.6, DOTTS is able to find a successful implementation, thus delivering a solution with a cheaper implementation (cost of 106), compared to the initial topology "orion 1", which has a cost of 124.

5.4.2.2 Optimization for Best-Effort Traffic

Although backward compatibility with Ethernet traffic is one of TTEthernet's strong points, there is no research that we are aware of that optimizes BE traffic in TTEthernet networks. Throughout Chapter 4 we focused on hard real-time traffic transported via TT and RC frames. In this section, we turn our attention to best-effort traffic, which has no real-time or quality of service guarantees.

DOTTS is a flexible framework. We have shown in Section 5.4.2.1 how DOTTS can be used to optimize the topology of a TTEthernet network. DOTTS that can be used to optimize different aspects of a TTEthernet network, by changing its cost function. In this section, we want to modify DOTTS to obtain schedulable implementations that maximize the available bandwidth for the BE traffic. We do this by changing the cost function used by DOTTS from Eq. 4.1 to the following:

$$Cost = \begin{cases} c_1 = \sum_i \max(0, R_{f_i} - f_i.deadline) & \text{if} c_1 > 0, f_i \in \mathcal{M}_{TT} \cup \mathcal{M}_{RC} \\ c_2 = \sum_i \max(0, BW_{Req}(dl_j) - BW_{Avail}(dl_j)) & \text{if} c_1 = 0 \text{ and } c_2 > 0, dl_j \in \mathcal{DL} \\ c_3 = \sum_i (BW_{Req}(dl_j) - BW_{Avail}(dl_j)) & \text{if} c_1 = 0 \text{ and } c_2 = 0, dl_j \in \mathcal{DL} \end{cases} \quad (5.15)$$

Once all the TT and RC frames are schedulable, i.e., each R_{f_i} is smaller than the deadline $f_i.deadline$, and as such, $c_1 = 0$, DOTTS will search for a solution that satisfies the bandwidth requirements of the BE frames. If at least one dataflow link $dl_j \in \mathcal{DL}$ exists, where the $BW_{Req}(dl_j)$ bandwidth required by the BE frames routed via dl_j is larger than the available bandwidth $BW_{Avail}(dl_j)$, the BE traffic has insufficient bandwidth, i.e., $c_2 > 0$, and DOTTS will use c_2 as the value of the cost function. The available bandwidth $BW_{Avail}(dl_j)$ is computed for each dataflow link, by subtracting from the dataflow link's maximum bandwidth, the bandwidth required by the TT and RC frames. In case all the dataflow links have sufficient available bandwidth to satisfy the BE traffic, c_2 is 0, and the value of the cost function will be c_3. This cost function will drive DOTTS towards solutions that maximize the available bandwidth for BE frames.

In TTEthernet networks, BE traffic can be either statically routed, or be routed via address learning in the switches. For this optimization, we assume that the BE traffic is statically routed. We evaluated this approach using the Orion CEV case study, more specifically, the topology "orion 3" obtained Table 5.6 and shown in Fig. 5.18 (without NS_7, NS_8 and the dash-dotted links) . For BE traffic, we added messages with bandwidth requirements that vary between 96 kbps (the average internet radio stream) and 10 Mbps (the read/write speed of DVD at 1x speed). The results are presented in Table 5.7. We have 4 benchmarks, denoted with "be 1" to "be 4". The number of BE

Table 5.7: Optimization of BE traffic experimental results

Benchmark	ES	NS	TT and RC Messages	BE Messages	Frame Instances	$BW_{\%}^{BE}$
be 1	31	12	187	41	8588	100
be 2				63	8500	100
be 3				83	8824	100
be 4				101	8810	100

messages (see Column 5) increase from 41 in benchmark "be 1", to 101 in benchmark "be 4". Columns 2, 3 and 4 present the number of ESes, NSes and TT and RC frames in the "orion" benchmark, using the "orion 3" topology. Column 6 presents the number of frame instances, and column 7 shows $BW_{\%}^{BW}$, the percentage of BE messages that have their bandwidth requirements met.

As the results in Table 5.7 show, DOTTS can be used also to optimize a TTEthernet network to take into account BE traffic. Even though the BE traffic is increasing, DOTTS is able to find solutions such that all the TT and RC frames meet their real-time constraints and the BE frames have their bandwidth requirements met. Furthermore, this evaluation shows that our proposed metaheuristic is flexible, and that we can tackle different optimization problems by changing the cost function.

CHAPTER 6

Conclusions and Future Work

This chapter discusses the conclusions of this thesis and presents future work ideas in the context of mixed-criticality real-time embedded systems.

6.1 Conclusions

In this thesis we have proposed methods and tools for mixed-criticality distributed real-time embedded systems. Mixed-criticality systems integrate on the same platform applications of different criticality levels. In such cases, the certification standards require that the applications of different SILs are developed and certified according to the highest level, dramatically increasing the certification costs. The other option is to provide separation mechanisms such that the applications of different criticality levels are isolated, so they cannot influence each other.

In this thesis, we consider that the platform enforces separation by implementing partitioning mechanisms similar to Integrated Modular Avionics [3]. At the communication network level we consider that the network implements the TTEthernet protocol, which provides both temporal and spatial partitioning. Chapter 2 presented our platform model, describing in detail partitioning at the processor and communication network level. Regarding the application model, we consider that each application is

composed of tasks that communicate using messages. The application model has been presented in Chapter 2.

The methods we have proposed in this thesis focus on the early life cycle phases of the system, where the impact of design decisions is greatest. We have highlighted in Section 1.3 the need for methods and tools to support the system engineers in taking design decisions. We have proposed several design optimization strategies that we have implemented as design space exploration tools.

To summarize our design optimization strategies at the processor-level:

- We have proposed an optimization approach that, for a given architecture and a fixed mapping of tasks to PEs, determines the sequence and size of the partition slices within the Major Frame on each PE. We have implemented this optimization as a Simulated Annealing metaheuristic. The experimental evaluations have shown that optimizing the partitioning is necessary to obtain schedulable implementations.

- We have discussed in Section 3.1.2 how partitioning constrains the way tasks use the PEs, introducing overheads. We have shown that by simultaneously optimizing the mapping and partitioning, the overheads are minimized, increasing thus the chance to find schedulable implementations. We have proposed a Tabu Search-based optimization strategy for this simultaneous optimization problems.

- For situations where the simultaneous optimization of mapping and partitioning does not find schedulable solutions, we have proposed an alternative to performing a costly upgrade of the architecture. Our approach elevates tasks to higher SILs to allow partition sharing, reducing the partitioning overheads at the expense of increasing the development and certification costs. We have presented this approach in Section 3.1.3. We have proposed a development cost model that captures the development costs associated to a given SIL (presented in Section 2.1.3). We have implemented this optimization approach as a Tabu Search metaheuristic.

- In Section 3.1.4 we have proposed another method to reduce the development costs. Safety-standards allow a task of higher SIL to be "decomposed", using redundancy to increase dependability, as several redundant tasks of lower SILs. We have extended our model to take into account task decomposition and we described decomposition in Section 2.1.2. We have proposed a Tabu Search-based strategy to optimize the task decomposition, mapping, partitioning and partition sharing such that the timing requirements are satisfied and the development costs are minimized.

- We have shown in Section 5.4.1 how to extend the MCDO algorithm from Section 3.2.2 to consider also soft real-time applications, in the context of realistic

space applications. We have proposed a cost function that captures, besides the schedulability of hard real-time applications, also the quality of service of the soft real-time applications. The experimental evaluation has shown that MCDO is an extensible framework, which, with minor changes, can tackle multiple design optimization problems.

- We have proposed a response time analysis to calculate the worst-case response times of tasks scheduled using fixed-priority preemptive scheduling (FPS) policy. The response time analysis extends the analysis from [136] to consider the influence of time-partitions on the schedulability of the FPS tasks. Compared to the analysis proposed by Audsley and Wellings [30] for partitioned PEs, our analysis dos not assume that the start time of partition slices within a major frame are periodic. This analysis was presented in Section 3.5.

To summarize our design optimization strategies at the communication network-level:

- In Section 4.1 we have proposed three optimization strategies to improve the schedulability of messages. In Section 4.1.2 we have shown that optimizing the fragmenting and packing of messages into frames can improve the schedulability of messages. In Section 4.1.3 we have proposed to optimize the routing of virtual links as an approach to increase the number of schedulable messages. In Section 4.1.4 we have shown that considering the RC traffic when scheduling the TT frames can greatly increase the schedulability of the RC frames. We have implemented the three strategies as Tabu Search optimizations. The experimental evaluations have proven that all the three approaches improve the schedulability of messages.

- Section 4.2 describes the DOTTS optimization strategy that includes the approaches described in Section 4.1. We have implemented the strategy using a Tabu Search metaheuristic. Given the network topology and the set of TT and RC messages, this strategy optimizes the fragmenting and packing of messages, the routing of virtual links and the schedules for the TT frames, such that the TT and RC frames are schedulable, and the end-to-end delay of the RC frames is minimized. The experimental evaluations have confirmed that the simultaneous optimization of fragmenting and packing, routing and scheduling is consistently better than any of the separate optimizations, and that this simultaneous optimization is necessary to find schedulable solutions.

- We have proposed in Section 5.4.2.1 a method to perform topology selection using DOTTS, to reduce the cost associated with the network implementation. Given the set of messages and an initial network topology we want to find a network implementation that reduces the number of network switches and links, such that all messages are schedulable. We have evaluated our proposed topology selection method using a realistic space application, i.e., the Orion Crew

Exploration Vehicle. The experimental evaluations have shown that DOTTS can be used for topology selection and that our method reduces the cost.

- We have shown in Section 5.4.2.2 how to extend the DOTTS optimization from Section 4.2 to consider also best-effort messages. We have proposed an alternative cost function that captures, besides the schedulability of the TT and RC frames, also the bandwidth requirements of the BE frames. The experimental evaluations have shown that DOTTS is in fact an extensible framework and can tackle different optimization problems with minor changes.

6.2 Future Work

In this section we present research challenges to be considered for future work.

- In this thesis we treat separately the optimizations at processor-level and at the communication network-level, and it is not trivial to integrate them. We have proposed in [164] an iterative method for a joint optimization of tasks and messages. At the system level, we will look into joint tasks and messages optimization.
- Partitioned architectures provide protection mechanisms to ensure that one application does not affect the performance of another application, in the safety domain. In some cases, systems have to be certified not only for safety, but also for security. As future work, the model provided in Chapter 2 and the optimization strategies proposed in Chapter 3 and Chapter 4 can be extended to take into account security models with different security levels. For example, Multiple Independent Levels of Security (MILS) [26, 49] is a security architecture that ensure secure integration of applications of different security levels and controlled information flow. MILS implements separation mechanisms similar to IMA (described in Section 2.2), but focusing on security. We already did a first step in this direction, by extending our model at the processor level to consider a separation requirements graph that specifies which tasks are not allowed to share partitions (see Section 2.1.4). Such a separation graph will also be needed at the communication network-level, to make sure our algorithms do not pack messages of different security levels into the same frame.

At the processor-level:

- The strategies proposed in Chapter 3 optimize the task decomposition, partitioning, partition sharing and mapping considering the architecture fixed. These strategies can be extended to also perform architecture selection. Given a library

of PEs, with different costs and capabilities, the optimization will return an architecture implementation that satisfies all the timing constraints and minimizes costs. Such an architecture implementations consists of: the set of chosen PEs, the mapping of tasks, the task decomposition, the partitioning and the partition sharing.

- We have shown in Section 5.4.1 how to extend the strategies from Chapter 3 to consider also soft real-time tasks. Our optimization strategies can be extended to take into account other constraints too, such as power consumption.

At the communication network-level:

- The optimization strategies proposed in Chapter 4 focus on design issues at the cluster level. But a TTEthernet network can consist of different clusters that have different time bases. For future work, our optimizations can be extended to consider multi-cluster networks. Since the clusters have different time bases, TT frames are transmitted between two clusters as RC frames. Thus, the extension will also have to take into account the conversion of TT frames into RC frames and vice versa.

- Due to legacy, safety or modularity constraints, or due to components having different communication requirements, the network of some systems contains subnetworks implementing different protocols interconnected using gateways. For example, a vehicle will have different networks for the body control, for the infotainment system or for the power train unit. Examples of protocols used in vehicular networks are the Controller Area Network (CAN) [4] and FlexRay [10]. A direction for future research is to extended the optimizations from Chapter 4 to consider networks composed of heterogeneous clusters, implementing TTEthernet and different communication protocols.

Bibliography

[1] Mars Pathfinder. http://www.nasa.gov/mission_pages/mars-pathfinder/.

[2] SAE Technical Report J2056/1: Class C Application Requirement considerations. Technical report, SAE International, 1993.

[3] *ARINC 651-1: Design Guidance for Integrated Modular Avionics*. ARINC (Aeronautical Radio, Inc), 1997.

[4] *ISO 11898: Road Vehicles – Controller Area Network (CAN)*. International Organization for Standardization (ISO), Geneva, Switzerland, 2003.

[5] The Vision for Space Exploration. Technical report, National Aeronautics and Space Administration, 2004.

[6] Study of Worldwide Trends and RD Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area. Technical Report MSU-CSE-00-2, F.A.S.T., Munchen, Germany, November 2005.

[7] *ARINC 664P7: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. ARINC (Aeronautical Radio, Inc), 2009.

[8] *IEEE 802.1Qav - IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queueing Enhancements for Time-Sensitive Streams*. IEEE, 2009.

[9] *IEEE 802.1Qat - IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol*. IEEE, 2010.

[10] *ISO 10681: Road vehicles – Communication on FlexRay*. International Organization for Standardization (ISO), Geneva, Switzerland, 2010.

[11] ARTEMIS Strategic Research Agenda. Technical report, ARTEMIS Industry Association, 2011.

[12] *IEEE 802.1AS - IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*. IEEE, 2011.

[13] *IEEE 802.1BA - IEEE Standard for Local and Metropolitan Area Networks - Audio Video Bridging (AVB) Systems*. IEEE, 2011.

[14] More than 50 billion connected devices. White Paper 284 23-3149 Uen, Ericsson, February 2011.

[15] *Systems Engineering Handbook V3.2*. INCOSE, 2011.

[16] Strategic Research Agenda. Technical report, Embedded Systems Institute, 2012.

[17] *ARINC 653P0: Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653*. ARINC (Aeronautical Radio, Inc), 2013.

[18] Embedded Market Study. Technical report, UBM Tech Electronics, 2013.

[19] *ETG 1000 EtherCAT Specification*. EtherCAT Technology Group, 2013.

[20] NI CompactRIO, 2013. http://www.ni.com/compactrio/.

[21] The Teardown: Apple iPhone 5s. *Engineering and Technology*, 8:84–85, November 2013.

[22] Emile Aarts, Jan Korst, and Wil Michiels. Simulated annealing. In Edmund Burke and Graham Kendall, editors, *Search Methodologies*, pages 187–210. Springer, 2005.

[23] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium*, pages 4 –13, 1998.

[24] AbsInt. aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/.

[25] Ahmad Al Sheikh, Olivier Brun, Maxime Chéramy, and Pierre-Emmanuel Hladik. Optimal design of virtual links in AFDX networks. *Real-Time Systems*, 49(3):308–336, 2013.

[26] Jim Alves-Foss, Paul W Oman, Carol Taylor, and W Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3):239–247, 2006.

[27] Mark S. Anderson, Jason M. Andringa, Robert W. Carlson, Pamela Conrad, Wayne Hartford, Michael Shafer, Alejandro Soto, Alexandre I. Tsapin, Jens Peter Dybwad, Winthrop Wadsworth, and Kevin Hand. Fourier transform infrared spectroscopy for Mars science. *Review of Scientific Instruments*, 76(3), 2005.

[28] AS 6802. Time-Triggered Ethernet. SAE International, 2011.

[29] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling. In *Proceedings of Euromicro Workshop on Real-Time Systems*, pages 36–41, 1993.

[30] N. Audsley and A. Wellings. Analysing APEX applications. In *Proceedings of the Real-Time Systems Symposium*, pages 39–44, 1996.

[31] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, Department of Computer Science, University of York, UK, November 1991.

[32] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability, 2001.

[33] H. Ayed, A. Mifdaoui, and C. Fraboul. Frame packing strategy within gateways for multi-cluster avionics embedded networks. In *Emerging Technologies Factory Automation*, pages 1–8, 2012.

[34] Luis Silva Azevedo, David Parker, Martin Walker, Yiannis Papadopoulos, and Rui Esteves Araujo. Automatic decomposition of safety integrity levels: Optimization by tabu search. In *Workshop on Critical Automotive applications: Robustness and Safety*, 2013.

[35] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russel Urzi. A research agenda for mixed-criticality systems. In *Cyber-Physical Systems Week*, 2009.

[36] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the Real-Time Systems Symposium*, pages 34–43, 2011.

[37] Sanjoy Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 536–543, 2004.

[38] Sanjoy Baruah and Nathan Fisher. Hybrid-priority scheduling of resource-sharing sporadic task systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 248–257, 2008.

[39] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proceedings of the Real-Time Systems Symposium*, pages 3–12, 2011.

[40] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium*, pages 13–22, 2010.

[41] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 147–155, 2008.

[42] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Annual European Symposium on Algorithms*, pages 555–566, 2011.

[43] S.K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[44] Von Günther Baumann. Eine elektronisch gesteuerte kraftstoffeinspritzung für ottomotoren. *Bosch Techn. Berichte*, (3):107–114, November 1967.

[45] Daniel F. Berisford, Kevin H. Hand, Paulo J. Younse, Didier Keymeulen, and Robert Carlson. Thermal testing of the compositional infrared imaging spectrometer. In *International Conference on Environmental Systems*, Jul. 2012.

[46] P. Binns. A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach. In *Conference on Digital Avionics Systems*, volume 1, pages 1B6/1–1B6/12, 2001.

[47] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches–A survey. *Annals of Software Engineering*, 10(1):177–205, 2000.

[48] Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[49] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *Proceedings of the Digital Avionics Systems Conference*, pages 1.C.2–1–1.C.2–14, 2008.

[50] Bosch automotive. A product history. *Journal of Bosch History. Supplement 2*.

[51] Robert Bosch and Michael Trick. Integer programming. In Edmund Burke and Graham Kendall, editors, *Search Methodologies*, pages 69–95. Springer US, 2005.

[52] Michelle Boucher. Sector Insight: Keeping Automotive Competitive with Embedded Systems. Technical report, Aberdeen Group, August 2013.

[53] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[54] Dennis M. Buede. Introduction to systems engineering. In *The Engineering Design of Systems: Models and Methods*, pages 3–36. John Wiley & Sons, Inc., 2000.

[55] Alan Burns and Rob Davis. Mixed criticality systems – a review. Jul 2013.

[56] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3 edition, 2011.

[57] Certification Authorities Software Team (CAST). CAST-2: Guidelines for assessing software partitioning/protection schemes. Position Paper, Federal Aviation Administration, 2001.

[58] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 190–195, 2003.

[59] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 2009. Available online at http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code.

[60] Carlos A. Coello Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary algorithms for solving multi-objective problems*. Kluwer Academic, 2007.

[61] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[62] L.A. Cortés, P. Eles, and Z. Peng. Quasi-static scheduling for real-time systems with hard and soft tasks. In *Proceedings of the Conference on Design, automation and test in Europe*, pages 21176–21181, 2004.

[63] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in Real-Time Systems*. John Wiley & Sons, LTD, 2002.

[64] Rodney Cummings, Kai Richter, Rolf Ernst, Jonas Diemer, and Arkadeb Ghosal. Exploring Use of Ethernet for In-Vehicle Control Applications: AFDX, TTEthernet, EtherCAT, and AVB. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 5(1):72–88, 2012.

[65] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the Real-Time Systems Symposium*, pages 291–300, 2009.

[66] Kalyanmoy Deb. *Search Methodologies*, chapter Multi-Objective Optimization. Springer, 2005.

[67] James A. Debardelaben, Vijay K. Madisetti, and Anthony J. Gadient. Incorporating cost modeling in embedded-system design. *IEEE Design and Test of Computers*, 14:24–35, July 1997.

[68] J. D. Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[69] T. Demmeler and P. Giusto. A universal communication model for an automotive system integration platform. In *Proceedings of Design, Automation and Test in Europe*, pages 47–54. IEEE, 2002.

[70] Robert Dick. Embedded system synthesis benchmarks suite, 2005. http://ziyang.eecs.umich.edu/dickrp/e3s/.

[71] D-jetronic history and fundamentals. Web Page. http://members.rennlist.com/pbanders/djetfund.htm.

[72] Francois Dorin, Pascal Richard, Michael Richard, and Joel Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

[73] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[74] Rolf Ernst. Certification of Trusted MPSoC Platforms. 10th International Forum on Embedded MPSoC and Multicore, 2010.

[75] EUROCAE ED-94B. Final report for clarification of ED-12B "Software Considerations in Airborne Systems and Equipment Certification". The European Organization for Civil Aviation Equipment, 2001.

[76] Faulhaber. Series E2 Optical Incremental Encoders datasheet, 2008. http://www.micromo.com/Micromo/Encoder2/E2_MME.pdf.

[77] C. Ferdinand and R. Heckmann. Verifying timing behavior by abstract interpretation of executable code. *Correct Hardware Design and Verification Methods*, 3725:336–339, 2005.

[78] C.J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems Journal*, 14(1):61–93, JAN 1998.

[79] Fletcher, Mitch. Progression of an open architecture: from Orion to Altair and LSS. Technical report, Honeywell, International, 2009.

[80] V. Formisano, F. Angrilli, G. Arnold, S. Atreya, G. Bianchini, D. Biondi, A. Blanco, M.I. Blecka, A. Coradini, L. Colangeli, A. Ekonomov, F. Esposito, S. Fonti, M. Giuranna, D. Grassi, V. Gnedykh, A. Grigoriev, G. Hansen, H. Hirsh, I. Khatuntsev, A. Kiselev, N. Ignatiev, A. Jurewicz, E. Lellouch, J. Lopez Moreno, A. Marten, A. Mattana, A. Maturilli, E. Mencarelli, M. Michalska, V. Moroz, B. Moshkin, F. Nespoli, Y. Nikolsky, R. Orfei, P. Orleanski, V. Orofino, E. Palomba, D. Patsaev, G. Piccioni, M. Rataj, R. Rodrigo, J. Rodriguez, M. Rossi, B. Saggin, D. Titov, and L. Zasova. The Planetary Fourier Spectrometer (PFS) onboard the European Mars Express mission. *Planetary and Space Science*, 53(10):963 – 974, 2005.

[81] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[82] M. Gendreau. *An Introduction to Tabu Search*. Centre for Research on Transportation, July 2002.

[83] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[84] Luis Gomes, J. Barros, and A. Costa. Modeling formalism for embedded system design. In R. Zurawski, editor, *Embedded systems handbook*, chapter 5. CRC Press, 2005.

[85] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proceedings of Design, Automation, and Test in Europe Conference and Exhibition*, pages 1227–1232, 2012.

[86] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration - The VLSI Journal*, 38(2):131–183, 2004.

[87] Joern Gronholz and Werner Herres. Understanding FT-IR data processing. Part 2: details of the spectrum calculation. *Instruments and Computers*, 3:10–16, 1985.

[88] Steve Heath. *Embedded systems design*. Newnes, 2002.

[89] Jonathan L. Herman, Christopher J. Kenna, Malcolm S. Mollison, James H. Anderson, and Daniel M. Johnson. RTOS Support for Multicore Mixed-Criticality Systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 197–208. IEEE, 2012.

[90] T. Herpel, B. Kloiber, R. German, and S. Fey. Routing of Safety-Relevant Messages in Automotive ECU Networks. In *Vehicular Technology Conference Fall*, pages 1–5, 2009.

[91] Greg Horvath, Seung H. Chung, and Ferner Cilloniz-Bicchi. Safety-critical partitioned software architecture: A partitioned software architecture for robotic spacecraft. In *Proceedings of the Aerospace Conference*, 2011.

[92] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, 1993.

[93] IBM. DO-178B compliance: turn an overhead expense into a competitive advantage. White paper, IBM Rational, 2010.

[94] IEC 61508. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission, 2010.

[95] ISO 9001. Quality management systems - Requirements. International Organization for Standardization, 2008.

[96] ISO/DIS 26262. ISO/DIS 26262 - Road vehicles — Functional safety. International Organization for Standardization / Technical Committee 22 (ISO/TC 22), 2009.

[97] ISO/IEC 15288. ISO/IEC 61508: Systems and software engineering – System life cycle processes. International Organization for Standardization and the International Electrotechnical Commission, 2008.

[98] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 915–920, 2008.

[99] M. Jakovljevic and A. Ademaj. Ethernet protocol services for critical embedded systems applications. In *Proceedings of the Digital Avionics Systems Conference*, pages 5.B.3–1–5.B.3–10, 2010.

[100] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[101] Owen R. Kelly, Hakan Aydin, Baoxian Zhao, Guojun Wang, Stephen R. Tate, Jian-Jia Chen, and Kouichi Sakurai. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Proceedings of the International Joint Conference on Trust, Security and Privacy in Computing and Communications*. IEEE Computer Society, 2011.

[102] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[103] Hermann Kopetz. An integrated architecture for dependable embedded systems. In *Proceedings of the International Symposium on Reliable Distributed Systems*, pages 160–161, 2004.

[104] Hermann Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered Ethernet (TTE) design. In *Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 22–33, 2005.

[105] Alexander Kossiakoff and William N. Sweet. The system development process. In *Systems Engineering Principles and Practice*, pages 50–89. John Wiley & Sons, Inc., 2005.

[106] Jerry Krasner. DO 178B-Redux: Looking at Developer Preferences, Issues and Vendor Cost. Technical report, EmbeddedMarket Forecasters, 2013.

[107] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan (Raj) Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 47–56. IEEE Computer Society, 2011.

[108] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[109] Yann-Hang Lee, Daeyoung Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Proceedings of Dependable Systems and Networks*, pages 14–23, 2000.

[110] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. A Comparison of Partitioning Operating Systems for Integrated Systems. *Computer Safety, Reliability, and Security*, pages 342–355, 2007.

[111] Haohan Li and Sanjoy Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of the Real-Time Systems Symposium*, pages 183–192, 2010.

[112] Haohan Li and Sanjoy Baruah. Global mixed-criticality scheduling on multiprocessors. In *Euromicro Conference on Real-Time Systems*, pages 166–175, 2012.

[113] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[114] Sorin O. Marinescu, Domiţian Tămaş-Selicean, V. Acretoaie, and P. Pop. Timing analysis of mixed-criticality hard real-time applications implemented on distributed partitioned architectures. In *Proceedings of the Conference on Emerging Technologies Factory Automation*, pages 1–4, 2012.

[115] L. Mertz. Auxiliary computation for Fourier spectrometry . *Infrared Physics*, 7(1):17 – 23, 1967.

[116] V. Mikolasek, A. Ademaj, and S. Racek. Segmentation of standard ethernet messages in the Time-Triggered Ethernet. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation*, pages 392–399, 2008.

[117] Utayba Mohammad and Nizar Al-holou. Development of an automotive communication benchmark. *Canadian Journal on Electrical and Electronics Engineering*, 1(5):99–115, 2010.

[118] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the Conference on Computer and Information Technology*, pages 1864–1871, 2010.

[119] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[120] Min-Young Nam, Jaemyun Lee, Kyung-Joon Park, Lui Sha, and Kyungtae Kang. Guaranteeing the End-to-End Latency of an IMA System with an Increasing Workload. *IEEE Transactions on Computers*, 99(PP):1, 2013.

[121] Scott D. Norris and Paul F. Marshall. Orion project status. *AIAA SPACE 2013 Conference and Exposition*, 2013.

[122] R. Obermaisser. *Time-Triggered Communication*. CRC Press, Inc., 2011.

[123] J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of Real-Time Systems Symposium*, pages 26–37, 1998.

[124] Y. Papadopoulos, M. Walker, M.-O. Reiser, M. Weber, D. Chen, M. Törngren, David Servat, A. Abele, F. Stappert, H. Lonn, L. Berntsson, Rolf Johansson, F. Tagliabo, S. Torchiaro, and Anders Sandberg. Automatic allocation of safety integrity levels. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety*, pages 7–10, 2010.

[125] David Parker, Martin Walker, LuísSilva Azevedo, Yiannis Papadopoulos, and RuiEsteves Araújo. Automatic decomposition and allocation of safety integrity levels using a penalty-based genetic algorithm. In *Recent Trends in Applied Artificial Intelligence*, volume 7906 of *Lecture Notes in Computer Science*, pages 449–459. Springer Berlin Heidelberg, 2013.

[126] Michael Paulitsch, E Schmidt, B Gstöttenbauer, C Scherrer, and H Kantz. Time-triggered communication (industrial applications). In *Time-Triggered Communication*, pages 121–152. CRC Press, 2011.

[127] P. Pedreiras and L. Almeida. Message routing in multi-segment FTT networks: the isochronous approach. In *Proceedings of Parallel and Distributed Processing Symposium*, pages 122–129, 2004.

[128] Alain Petrissans, Stephane Kraqczyk, Lorenzo Veronesi, Grabriella Cattaneo, Nathalie Feeney, and Cyril Meunier. Design of future embedded systems toward system of systems: trends and challenges. Technical report, European Commission.

[129] P. Pop, P. Eles, Z. Peng, and T. Pop. Analysis and optimization of distributed real-time embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):593–625, 2006.

[130] Paul Pop, Petru Eles, and Zebo Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the International Workshop on Hardware/Software Codesign*, pages 178–182, 1999.

[131] Paul Pop, Petru Eles, and Zebo Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 567–575, New York, NY, USA, 2000. ACM.

[132] Paul Pop, Petru Eles, and Zebo Peng. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. Kluwer Academic Publishers, 2004.

[133] Paul Pop, Petru Eles, and Zebo Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *ACM Transasctions on Embedded Computing Systems*, 4(1):112–140, Feb. 2005.

[134] Paul Pop, Petru Eles, Zebo Peng, Viacheslav Izosimov, Magnus Hellring, and Olof Bridal. Design optimization of multi-cluster embedded systems for real-time applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 21028–21033, 2004.

[135] Paul Pop, Leonidas Tsiopoulos, Sebastian Voss, Oscar Slotosch, Christoph Ficek, Ulrik Nyman, and Alejandra Ruiz. Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the RECOMP approach. In *Proceedings of the Workshop of Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2013.

[136] Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. Analysis and optimisation of hierarchically scheduled multiprocessor embedded systems. *International Journal of Parallel Programming*, 36(1):37–67, 2008.

[137] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1-3):205–235, 2008.

[138] Yves Robert. Task graph scheduling. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 2013–2025. Springer US, 2011.

[139] Rockwell-Collins. Certification cost estimates for future communication radio platforms. Technical report, Rockwell-Collins, 2009.

[140] RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), 1992.

[141] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[142] John Rushby. Just-in-time certification. In *Proceedings of Conference on the Engineering of Complex Computer Systems (ICECCS)*, pages 15–24, 2007.

[143] Rishi Saket and Nicolas Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing*, 2(1):93–102, January 2006.

[144] A. Sangiovanni-Vincentelli. Electronic-system design in the automobile industry. *IEEE Micro*, 23(3):8–18, 2003.

[145] V. Saptari. *Fourier-Transform Spectroscopy Instrumentation Engineering*. SPIE Press, 2004.

[146] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. *Real-Time and Embedded Technology and Applications Symposium*, pages 89–98, 2010.

[147] Stefan Schneele and Fabien Geyer. Comparison of IEEE AVB and AFDX. In *Proceedings of the Digital Avionics Systems Conference (DASC)*, pages 7A1–1–7A1–9, 2012.

[148] Reinhard Schneider, Dip Goswami, Alejandro Masrur, and Samarjit Chakraborty. QoC-oriented efficient schedule synthesis for mixed-criticality cyber-physical systems. In *Proceedings of the Forum on Specification and Design Languages*, pages 60–67, 2012.

[149] Reinhard Schneider, Licong Zhang, Dip Goswami, Alejandro Masrur, and Samarjit Chakraborty. Compositional analysis of switched ethernet topologies. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, pages 1099–1104, 2013.

[150] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009:2:1–2:17, January 2009.

[151] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the International Symposium on Networks on Chip*, pages 152–160, 2012.

[152] Kathleen E. Schubert, Frank Gati, James M. Free, and Harry A. Cikanek III. Orion crew exploration vehicle preliminary design. *Proceedings of the International Astronautical Congress*, 5:3620–3634, 2010.

[153] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the USENIX Conference on Security*, pages 1–1, 2010.

[154] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the Real-Time Systems Symposium*, pages 181–191. IEEE Computer Society, 1986.

[155] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

[156] Till Steinbach, Hyung-Taek Lim, Franz Korf, Thomas C. Schmidt, Daniel Herrscher, and Adam Wolisz. Tomorrow's In-Car Interconnect? A Competitive Evaluation of IEEE 802.1 AVB and Time-Triggered Ethernet (AS6802). In *Proceedings of the Vehicular Technology Conference*, pages 1–5. IEEE Press, September 2012.

[157] W. Steiner. Synthesis of Static Communication Schedules for Mixed-Criticality Systems. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 11–18, 2011.

[158] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan. TTEthernet Dataflow Concept. In *Proceedings of the International Symposium on Network Computing and Applications*, pages 319–322, 2009.

[159] Wilfried Steiner. An Evaluation of SMT-based Schedule Synthesis For Time-Triggered Multi-Hop Networks. In *Proceedings of the Real-Time Systems Symposium*, pages 375–384, 2010.

[160] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[161] J.F. Suen, R.B. Kegley, and J.D. Preston. Affordable avionic networks with Gigabit Ethernet assessing the suitability of commercial components for airborne use. In *Proceedings of SoutheastCon*, pages 1–6, 2013.

[162] E. Suethanuwong. Scheduling time-triggered traffic in TTEthernet systems. In *Emerging Technologies Factory Automation*, pages 1–4, 2012.

[163] D. Tămaş-Selicean, D. Keymeulen, D. Berisford, R. Carlson, K. Hand, P. Pop, W. Wadsworth, and R. Levy. Fourier transform spectrometer controller for partitioned architectures. In *Proceedings of the Aerospace Conference*, pages 1–11, 2013.

[164] Domiţian Tămaş-Selicean, S. O. Marinescu, and Paul Pop. Analysis and optimization of mixed-criticality applications on partitioned distributed architectures. In *Proceedings of the IET System Safety Conference*. Institution of Engineering and Technology, 2012.

[165] Domiţian Tămaş-Selicean and Paul Pop. Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures. In *Proceedings of the Real-Time Systems Symposium*, pages 24–33, 2011.

[166] Domiţian Tămaş-Selicean and Paul Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 1–10, 2011.

[167] Domiţian Tămaş-Selicean and Paul Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 282–283, 2011.

[168] Domiţian Tămaş-Selicean and Paul Pop. Design optimization of mixed-criticality real-time systems. *ACM Transactions on Embedded Computing*, 2014. Submitted to.

[169] Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of Communication Schedules for TTEthernet-based Mixed-Criticality Systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 473–482, 2012.

[170] Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Design Optimization of TTEthernet-based Distributed Real-Time Systems. *Real-Time Systems Journal*, 2014. Submitted to.

[171] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, 1975.

[172] Santiago Urueña, José A. Pulido, Jorge López, Juan Zamorano, and Juan A. Puente. A new approach to memory partitioning in on-board spacecraft software. In *Reliable Software Technologies – Ada-Europe 2008*, volume 5026 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2008.

[173] Frank Vahid and Tony Givargis. *Embedded system design - a unified hardware / software introduction.* Wiley, 2002.

[174] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

[175] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, 2007.

[176] Arun Viswanathan and BC Neuman. A survey of isolation techniques. 2009.

[177] Winthrop Wadsworth and Jens-Peter Dybwad. Rugged high-speed rotary imaging Fourier transform spectrometer for industrial use. *Proceedings of SPIE*, 4577:83–88, 2002.

[178] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *Operating Systems Review*, 27:203–216, 1993.

[179] Armin Wasicek and Thomas Mair. Secure Information Sharing in Mixed Criticality Systems. In *Proceedings of the World Conference on Engineering and Science*, 2013.

[180] Wikipedia. Apollo guidance computer, December 2013. http://en.wikipedia.org/wiki/Apollo_Guidance_Computer.

[181] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstroem. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.

[182] WindRiver. Vxworks. Web Page. www.windriver.com/products/vxworks.

[183] James Windsor, M.-H. Deredempt, and R. De-Ferluc. Integrated modular avionics for spacecraft – user requirements, architecture and role definition. In *Proceedings of the Digital Avionics Systems Conference*, pages 8A6–1–8A6–16, 2011.

[184] James Windsor, K. Eckstein, P. Mendham, and T. Pareaud. Time and space partitioning security components for spacecraft flight software. In *Proceedings of the Digital Avionics Systems Conference*, pages 8A5–1–8A5–14, 2011.

[185] James Windsor and K. Hjortnaes. Time and space partitioning in spacecraft avionics. In *Proceedings of the International Conference on Space Mission Challenges for Information Technology*, pages 13–20, 2009.

[186] J. Xu and D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *Transactions on Software Engineer*, 19(1):70–84, 1993.