SAFURE

D3.2
Final Analysis of Integrity Algorithms

Project number:	644080
Project acronym:	**SAFURE**
Project title:	SAFety and secURity by dEsign for interconnected mixed-critical cyber-physical systems
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-644080-D3.2/ 1.0
Work Package:	WP 3
Due Date:	July 2017 - M30
Actual Submission Date:	31st July, 2017
Responsible Organisation:	ESCR
Editor:	André Osterhues
Dissemination Level:	Public
Revision:	1.0
Abstract:	This document will cover final results regarding the extension of temperature, data, and timing integrity to safe and secure systems. The report describes integrity methods and protection mechanisms related to data management, timing and thermal analysis for safe and secure systems as developed in WP3 and is the follow-up deliverable of D3.1.
Keywords:	Algorithms, Mixed-Criticality, Temperature, Data integrity, Timing integrity, Resource sharing integrity

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080.

This work is supported (also) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Editor

André Osterhues (ESCR)

Contributors (ordered according to beneficiary numbers)

Martin Deutschmann (TEC)
André Osterhues, Lena Steden (ESCR)
Mikalai Krasikau (SYSG)
Jonas Diemer (SYM)
Sylvain Girbal (TRT)
Robin Hofmann, Borislav Nikolic (TUBS)
Gabriel Fernandez, Jaume Abella, Francisco J. Cazorla (BSC)
Marco Di Natale, Youcheng Sun, Alessandro Biondi (SSSA)
Rehan Ahmed, Philipp Miedl, Lothar Thiele (ETHZ)

Disclaimer

The information in this document is provided ”as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The users thereof use the information at their sole risk and liability.

Executive Summary

There has been a tremendous improvement in performance and efficiency of processing platforms in the past four decades. System designers have exploited various architecture and device level techniques to bring about this improvement.
While the average case performance has increased tremendously, there is a large gap between the requirements of real-time applications and what architectures of embedded processors offer today. On the one hand, real-time applications need predictability in order to enable safe operation based on worst-case execution time analysis. On the other hand, following the end of Dennard scaling, embedded processors increasingly feature a multicore architecture with shared resources (e.g., last-level cache, memory controller) in order to keep improving performance and efficiency.

Contents

Chapter 1 Introduction 3
1.1 Meaning of Integrity in the Context of SAFURE . . . 3
1.2 Temperature Integrity . . . 3
1.3 Data Integrity . . . 4
1.4 Timing and Resource Sharing Integrity . . . 4
Chapter 2 Temperature Integrity 5
2.1 Thermal Isolation Servers . . . 5
2.1.1 Processor Model . . . 5
2.1.1.1 Thermal Model . . . 5
2.1.1.2 Power Model . . . 6
2.1.2 Application Model . . . 6
2.1.3 Sample platform architecture . . . 6
2.1.4 Formalizing Thermal Isolation Servers . . . 7
2.1.4.1 Thermal properties of Thermal Isolation Server (TI Server)s . . . 7
2.1.5 Timing guarantees . . . 8
2.1.5.1 Composing multiple servers . . . 9
2.1.6 Designing thermal Isolation servers . . . 9
2.1.6.1 Optimal task partitioning . . . 9
2.1.6.2 Optimal server configuration search . . . 10
2.2 Thermal Covert Channel . . . 10
2.2.1 Extended Robustness Analysis . . . 10
2.2.1.1 Thermal Noise . . . 11
2.2.1.2 Fans . . . 11
2.2.1.3 Core Pinning . . . 11
2.2.1.4 Sleep States . . . 12
2.2.1.5 Frequency Governor . . . 12
2.2.1.6 Scheduling . . . 12
2.2.2 Results from a realistic attack scenario . . . 12
2.3 Frequency Covert Channel . . . 13
2.3.1 Frequency Scaling in Linux . . . 14
2.3.1.1 The CPU frequency driver . . . 15
2.3.1.2 The governor . . . 15
2.3.2 Userspace Interfaces . . . 15
2.3.3 Threat Model . . . 16
2.3.3.1 Frequency Covert Channel Model . . . 16
2.3.4 Threat Potential Indicators . . . 17
2.3.4.1 Determining the Capacity Bound . . . 17
2.3.4.2 Experimental Evaluation . . . 18
2.4 Thermal Task Inference . . . 19
2.4.1 Thermal Data Preprocessing . . . 19
2.4.2 Correlation Based Approach . . . 20
2.4.2.1 Reference Traces . . . 20

2.4.2.2 Test Traces . . . 20
2.4.2.3 Video inference algorithm . . . 20
2.4.2.4 Results . . . 20
2.4.3 Neural Network Approach . . . 21
2.4.3.1 Used Layers . . . 21
2.4.3.1.1 Convolutional Layer . . . 21
2.4.3.1.2 Dense Layer . . . 22
2.4.3.1.3 Long Short Term Memory . . . 22
2.4.3.2 Implementation . . . 22
2.4.3.2.1 Model . . . 22
2.4.3.2.1.1 LSTM Based Approach . . . 23
2.4.3.2.1.2 Dense Layer Approach . . . 23
2.4.3.3 Data Augmentation . . . 23
2.4.4 Results . . . 25
2.4.5 Conclusion . . . 25
Chapter 3 Data Integrity **26**
3.1 Message Authentication Codes . . . 26
3.1.1 Hash-based Message Authentication Codes (HMACs) . . . 26
3.1.2 Keccak Message Authentication Codes (KMACs) . . . 26
3.2 Evaluation of Data Integrity Algorithms . . . 27
3.3 Key Management . . . 29
3.3.1 Key Generation . . . 29
3.3.2 Key Distribution . . . 29
3.3.3 Key Lengths . . . 29
3.3.4 Key Life-Cycle . . . 29
3.4 Design Guidelines . . . 30
3.4.1 Guidelines . . . 30
Chapter 4 Timing Integrity and Resource Sharing Integrity **31**
4.1 Timing Integrity for Multi-Cores: Challenge & Existing Solutions . . . 31
4.2 Timing Integrity for Multi-Cores: SAFURE Solution . . . 32
4.2.1 Principles . . . 32
4.2.2 Support & Tooling . . . 33
4.2.3 Process to determine and ensure budgeting . . . 33
4.2.4 Hardware characterization & budgeting . . . 34
4.2.5 Critical application characterization & budgeting . . . 34
4.2.6 The SAFURE Budget-Based RunTime Engine (BB-RTE) . . . 35
4.2.7 Results . . . 35
4.3 Timing Integrity in Overload Conditions . . . 35
4.3.1 The System Model . . . 37
4.3.2 The Solution Model . . . 38
4.3.2.1 Variables and basic constraints . . . 39
4.3.2.2 Busy periods . . . 39
4.3.2.3 Offsets . . . 40
4.3.2.4 Finish times . . . 40
4.3.2.5 Level-i idle time inside a job window . . . 40
4.3.2.6 Schedulability of each job of τ_i . . . 40
4.3.2.7 Interference from the previous jobs of the same task . . . 41
4.3.2.8 Number of interfering jobs from higher priority tasks . . . 41
4.3.2.9 Refining the interferences from higher priority tasks . . . 42
4.3.2.10 Constraints on the idle time and workload . . . 42
4.3.2.11 Minimum level-i idle time . . . 43

4.3.2.12 Idle time inside a job window 43
4.3.2.13 Formulation of the busy period $[a_k - L_k, f_k)$ when $\beta_{k-1} = 0$ 43
4.3.2.14 Formulation of the busy period $[f_{k-1}, f_k[$ when $\beta_{k-1} = 1$ 43
4.3.2.15 Formulation of f_k by accumulating the idle time and workload 44
4.3.2.15.1 Refining the arrival time of a higher priority job before the beginning or the end of a busy period 44
4.3.2.16 Length of a busy period 44
4.3.3 Weakly hard schedulability analysis 44
4.4 Extensions of the Solution Model 45
4.4.1 Shared resources 45
4.4.2 Jitter 45
4.4.3 Experiments 46
4.5 Optimizing the placement of time-critical automotive tasks in multicores 48
4.5.1 Modeling the Functionality and the Platform 50
4.5.2 Fuel injection applications and AVR tasks 51
4.5.3 Analysis Models 52
4.5.4 Optimization Algorithms 53
4.5.5 Linear Optimization 54
4.6 Vulnerability Detection for Multi-Cores 55
4.6.1 Shared Hardware Resources in the Juno board 56
4.6.2 On-Chip Cacheless Resource Sharing 56
4.6.2.1 Formalization of RUs and RUI 57
4.6.2.1.1 Resource usage signature (*RUs*) 57
4.6.2.1.2 Resource usage template (*RUI*) 58
4.6.2.1.3 RUs and RUI through an example 58
4.6.2.2 RUs & RUI for Measurement-Based Timing Analysis 59
4.6.2.2.1 Methodology 59
4.6.2.2.2 The case of a ARM big.LITTLE (Juno) architecture 60
4.6.2.2.3 Bus 61
4.6.2.2.4 Multi-resource signatures 62
4.6.3 OnChip Cache Sharing 63
4.6.3.1 Introduction to Stack Distance 63
4.6.3.2 Stack Distance as Proxy for Cache Contention 64
4.6.3.3 sdki 64
4.6.3.4 Obtaining sdki 65
4.6.3.5 Surrogate Cache Application Generator 65
4.6.3.5.1 Initialization and Pre-Initialization phases 65
4.6.3.5.2 Execution Phase 66
4.7 Timing Integrity for Network: SAFURE Solution 67
4.7.1 Timing Integrity for Network: Challenges & Existing Solutions 67
4.7.2 Compositional Performance Analysis for Ethernet 68
4.7.3 Switched Ethernet 68
4.7.4 Ethernet TSN - Time Sensitive Networking 70
4.7.4.1 TSN Time-Aware Shaper 70
4.7.4.2 TSN Peristaltic Shaper 72
4.7.4.2.1 Analysis of Non Peristaltic Traffic Streams 72
4.7.4.3 TSN Burst Limiting Shaper 74
4.7.5 TSN Frame Preemption 76
4.7.5.1 Frame preemption in IEEE802.1Q 77
4.7.5.2 Frame preemption in TSN (IEEE802.1Qbv) 80
4.7.6 Software Defined Networking 81

4.8 Vulnerability Detection for Networks 82
Chapter 5 Conclusion **84**
5.1 Temperature Integrity 84
5.2 Data Integrity 84
5.3 Timing and Resource Sharing Integrity 84
5.4 Integration Plan 84
Bibliography **86**

List of Figures

2.1 System parameters for the sample architecture 6
2.2 Period/Utilization trade-off when core 1 thermal budget for TI Server S_i executing on core 1 is fixed. 8
2.3 Packet structure 13
2.4 The source application (src) has access to restricted data, while the sink application (snk) has access to the internet. Although source and sink are isolated from each other, they manage to establish a covert channel by observing the frequency of the cores, thus compromising the security paradigm of permission separation and application isolation. 14
2.5 A simplified structure block diagram of the CPU frequency control in the Linux Kernel. 14
2.6 Results of the experimental evaluation of both platforms. The upper diagram sows the achieved throughput in bits per second (bps) for both platform, depending on the packet size. The middle diagram shows the percentage of throughput degradation compared to a reference platform, and the bottom diagram presents the packet error rate (PER). 18
2.7 Preprocessing applied on thermal traces 20
2.8 Result of the initial version of the thermal classifier, where the videos with the same content as the reference video have a higher score than the other videos. 21
2.9 A one dimensional convolution layer showing all the parameters 22
2.10 Two possibilities for the structure of the Neural Network (NN), where the dashed square shows which part is pretrained on the features. Afterwards the weights are fixed while training the Long Short-Term Memory Neural Network (LSTM) (a) or the second dense layer (b). 23
2.11 The procedure to generate the sequences for training the second part of the NN. . . . 24
2.12 Traces to train the network. Sub-figures (a) and (b) are randomly shifted versions of the originally collected data. Sub-figures (c) and (d) are concatenated ten seconds snippets of the black and the alternating video in two different orderings, which serve as training data for the LSTM. 24

4.1 Evaluation of existing Deterministic Platform Software (DPS) solutions 31
4.2 Budget-Based RunTime Engine (BB-RTE) 32
4.3 Budget-Based RunTime Engine (BB-RTE) Principles 32
4.4 Measure Environment for TIme-Critical Systems (METrICS) 33
4.5 Timing integrity process for mixed time-critical systems 33
4.6 Determining an acceptable level of slowdown and the associated extra access budget 34
4.7 A problem window with 3 job windows 38
4.8 Notation for the definition of a problem window 39
4.9 Runtime results for $K = 5$ 48
4.10 The main characteristics of the mapping problem. 50
4.11 Stereotypes in Rhapsody for modeling hardware (IO) resources in AUTOSAR. 53
4.12 Stereotypes in Rhapsody for the representation of the execution time of runnables (extracted from traces) as a function of the activation event and the execution mode. . 53
4.13 Pseudo-code of the Simulated Annealing routine. 54

4.14 Runtime of the MILP Optimization problem. 55
4.15 Shared Hardware resources alongside the memory path on the ARM Juno board. As shown each of the two clusters includes a shared L2 cache and communication channels towards the DDR memory controllers. 56
4.16 Main steps in the *RUs* and *RUI* methodology. 58
4.17 Hypothetical impact (in cycles) from/to the different access types to the bus. $l2h$, $l2m$ and st refer to L2 load hits, L2 load misses and stores respectively. 61
4.18 Block Diagram of the pointer chaising approach followed for loads. 66
4.19 Example queuing delay 70
4.20 TSN/TAS: (a) Common notations. Frame 3 experiences same-priority blocking from Frames 1 and 2. (b) Maximum blocking by a single TAS class. 71
4.21 TSN/PS: Frames of PS class I experience interference from higher priority classes K (non-PS) and J (PS) and one lower-priority frame. It takes more than one PS interval to transmit the first three frames of PS stream I. 73
4.22 TSN/BLS: (a) Frames of class I are sent at their maximum rate until H I is reached and traffic from I is blocked for credit replenishment. (b) Frames of class I are interleaved with frames of other classes and class I's credit is decremented while these frames are being transmitted. Notice how in (b) more workload is released in the marked interval t. 74
4.23 Example of non-preemptive and preemptive frame transmission. 77
4.24 Frame preemption introduces new frames, which all have to look like valid Ethernet frames to the PHY. 77
4.25 Example of non-preemptive and preemptive frame transmission in IEEE 802.1Qbv . . 80
4.26 SDN network (re)configuration protocols 82
4.27 Example network for the SDN admission control 82
4.28 CPA model for the explicit flow configuration protocol from Figure 4.26 a) 83
4.29 CPA model for the predefined flow protocol from Figure 4.26 b) 83

List of Tables

2.1 Parameters of the conservative governor and the characteristics of the platforms *Laptop* and *Hand-Held*. . . . 17
2.2 Experiment packet payload, the corresponding number of bits per experiment trace and the throughput on the reference platforms. . . . 18
2.3 The results of the two model types with two to five different sequences to classify. The time column is normalized on the time to train the two sequence LSTM. . . . 25
2.4 With increasing number of features computed by part 1, the number of parameters to train rises faster for the dense layer approach. . . . 25

3.1 Evaluation results for Infineon AURIX TC277T TriCore Microcontroller 28

4.1 A summary of variables defined 39
4.2 The counting of higher priority jobs 41
4.3 An automotive case study 47
4.4 Percentage of sets with K consecutive deadline misses 47
4.5 Experiments that confirm the m-K property and run out of time limit (n/a) with variable n 48
4.6 Percentage of valid m-K property with variable K 48
4.7 Percentage of valid m-K property with variable U 48
4.8 Illustrative example of how stack distance helps capturing misses due to contention . 63
4.9 Parameters used by SurAppGen 65

5.1 Integration plan for technologies presented in deliverable D3.2 85

Glossary

sink	receiving application	10–12
source	sending application	9–12
AMBA	Advanced Microcontroller Bus Architecture	59, 60
BB-RTE	Budget-Based RunTime Engine	31–34
BER	bit error rate	12
bps	bits per second	12, 16, 17
CFS	Completely Fair Scheduler	12
COTS	Commercial off-the-shelf	55, 56
DBF	Demand Bound Function	7
DPS	Deterministic Platform Software	30, 31
DVFS	Dynamic Voltage and Frequency Scaling	11–13
EDF	Earliest Deadline First	6, 7
ETB	Execution Time Bound	55–59, 61
FIFO	First In, First Out	11, 60
HMAC	Keyed-Hash Message Authentication Code (MAC)	25, 26
KMAC	Keccak MAC	25, 26
LSTM	Long Short-Term Memory Neural Network	21–24
MAC	Message Authentication Code	25, 26
MD5	Message-Digest Algorithm 5	25
METrICS	Measure Environment for TIme-Critical Systems	32
NN	Neural Network	20–23
OS	Operating System	10, 11, 13, 15
PER	packet error rate	17, 18
PMC	Performance Monitor Counters	32–34
RAM	Random Access Memory	59
SB	Stressing Benchmarks	33

SBF	Supply Bound Function	7
SHA	Secure Hash Algorithm	25
SoC	System-on-Chip	16
TI Server	Thermal Isolation Server	4, 6–9, 83, 84
VM	Virtual Machine	11, 12
WCET	Worst Case Execution Time	5, 33, 55
XOR	exclusive disjunction	25

Chapter 1 Introduction

The focus of Work Package 3 is to study algorithms for ensuring integrity of Safe and Secure systems which are the focus of SAFURE project. This document overviews the integrity algorithms and presents first results on the application of these algorithms and/or development of new algorithms for preserving system integrity.
This document reflects the research done since the publication of D3.1 and therefore mainly can be seen as continuation of D3.1. While this chapter was already provided in D3.1, it is also provided here for the sake of making D3.2 a self-contained complete report.

1.1 Meaning of Integrity in the Context of SAFURE

Before delving into the different aspects of integrity, it is important to define the integrity of a computer system. In one of the seminal works titled “Integrity Consideration for computer Systems” [11], K.J. Biba states that ”We consider a subsystem to have the property of integrity if it can be trusted to adhere to a well defined code of behavior”. This “code of behaviors” is the specification of a given system.
Deliverable D1.3 states the SAFURE framework specification in detail. However, important aspects of this specification are outlined here to motivate the different aspects of system integrity. As stated in D1.3, SAFURE adopts a unified presentation of properties that make a system dependable. Specifically, in SAFURE we focus on the following system attributes:

- Safety attributes: maintainability, reliability and safety
- Security (as of IT security) attributes : availability, confidentiality and integrity

For ensuring that a mixed critical system has these attributes, we have identified three separate areas of system integrity:

- Temperature integrity
- Data integrity
- Timing and resource sharing integrity

We will now briefly overview these integrity aspects. Details on each of these aspects are given in Chapters 2, 3 and 4.

1.2 Temperature Integrity

Temperature integrity refers to maintaining system temperature below a safe operating threshold. It mainly affects reliability, safety, availability and confidentiality attributes of a safe and secure system. This aspect has become important due to rapid increase in power density of modern processing platforms. High temperature conditions adversely affect the reliability/safety of a system. Since reliability and safety are fundamental requirements of mission-critical real-time systems, adherence to thermal constraints is vital for maintaining system integrity. In SAFURE we study the thermal impact of executing tasks of multiple criticalities on a multi-core platform.

We also identify that temperature can be used to compromise the security of a system by its use as a covert communication channel. This compromises the confidentiality attribute. We study analysis/mitigation strategies for countering these threats to system integrity.

1.3 Data Integrity

Data integrity refers to assuring and maintaining the accuracy of data. Thus it can be said that data integrity techniques aim at preventing unintentional changes to information. Data integrity mainly affects reliability, safety, confidentiality and integrity attributes of a safe and secure system.
Problems in that domain comprise unintended changes to data due to storage, retrieval or processing operations. This also includes targeted changes, unexpected hardware failures, human errors and malicious attackers. Measures to preserve integrity of data are diverse, including application of checksums, error correcting codes as well as cryptographic message authentication codes (MACs) and access control techniques, the latter two of which are considered in SAFURE.

1.4 Timing and Resource Sharing Integrity

Timing integrity refers to the property of a (real-time) system to meet its timing requirements, e.g. deliver a response to an external stimulus in time. Timing integrity mainly effects maintainability, reliability, safety and availability attributes of a safe and secure system.
There are varying degrees of criticality w.r.t. timing integrity, ranging from ”best effort” (e.g. providing some service at all) all the way up to safety-critical control loops with short deadlines (e.g. advanced driver assistance, autonomous driving).
Guaranteeing timing integrity requires that all effects that impact the timing of a certain function are controlled. Typically, this means controlling interference during the sharing of resources (e.g. processor time, memory access, ...). This can be done by providing corresponding hardware/software mechanisms on the execution platform. Giving a guarantee on timing integrity typically involves performing a formal analysis of the timing properties.

Chapter 2 Temperature Integrity

This chapter focuses on temperature integrity mechanisms developed in SAFURE. It covers both safety and security implications of temperature. For the safety aspect, we overview thermal isolation servers, which can be used to provide thermal protection in a multi-core mixed-critical system. For the security aspect, we provide a progression of the thermal covert channel analysis covered in D3.1. Additionally, we present an analysis of the frequency covert channel. Towards the end of this chapter, we also provide preliminary results for a temperature side-channel attack.

2.1 Thermal Isolation Servers

To provide thermal protection, we developed a new scheduling construct called Thermal Isolation Server which significantly simplifies the real-time application design process under thermal constraints. A TI Server has thermal budget associated with it, which upper-bounds the temperature increase caused by tasks executed by it. Through detailed theoretical analysis, we prove that a given server will never exceed its thermal budget. This allows us to thermally isolate different sets of tasks. The total budget is governed by the thermal constraint of the hardware platform. Furthermore, we prove that TI Servers are time- and space-composable; which implies that we can simply *add* the thermal budgets of several servers executing in parallel to compute their net worst-case temperature increase. This composability significantly simplifies the server design problem. We also propose a heuristic to design a set of TI Servers given the computation requirements of tasks and the thermal constraint of the hardware platform.
We now formalize the models used in this work. Bold characters represent vectors and matrices while non-bold characters represent scalars. Subscripts are used to reference individual elements of matrices/vectors; e.g. $H_{k,l}$ denotes the element in k^{th} row and l^{th} column of matrix $\boldsymbol{H}$, and T_i denotes the ith element of vector $\boldsymbol{T}$. $\boldsymbol{I}$ is used to denote identity matrix.

2.1.1 Processor Model

We consider a multi-core constituting m identical cores. The set of all cores is denoted by M. Each core has two operation modes: *active* and *idle*. The mode of a given core has a direct impact on its dynamic power dissipation. A detailed power model is discussed later in this section. The platform has a thermal constraint $\boldsymbol{T^\Delta}$. The temperature of each core $i \in M$ is required to be at or below T_i^Δ for safe system operation.

2.1.1.1 Thermal Model

We develop the thermal model of the multi-core using an equivalent RC network [80, 21, 73]. In this abstraction, temperature is modeled by voltage and power dissipation is modeled by a current source. We model the layout of the chip by four vertical layers, namely heat sink, heat spreader, thermal interface, and silicon die. This is identical to the model adopted by the Hotspot thermal simulator [80]. Each layer is divided into a set of blocks according to the architectural-level units. In our case, we select a processing component abstraction, , i.e., we represent each core as an individual node with separate power source and temperature characteristics. $R_{i,j}$ represents the thermal resistance

between note i and node j, with $R_{i,i} = \infty$. The processing component abstraction has been shown to be reasonably accurate [24, 90]. In our thermal model, 12 additional nodes are introduced in the heat spreader and heat sink layers to account for the area which is not covered by the subjacent layer. Therefore, a multi-core system with m cores is modeled by $n = 4 \cdot m + 12$ thermal nodes. The n-dimensional temperature vector $\boldsymbol{T}(t)$ at time t is described by a set of first-order differential equations:

$$\boldsymbol{C} \cdot \frac{\mathrm{d}\boldsymbol{T}(t)}{\mathrm{d}t} = (\boldsymbol{G} - \boldsymbol{K}) \cdot \boldsymbol{T}(t) + \mathbf{Power}(t) + \boldsymbol{K} \cdot \boldsymbol{T}^{\mathbf{A}} \tag{2.1}$$

where $\boldsymbol{C}$ is the thermal capacitance diagonal matrix, $\boldsymbol{G}$ is a square matrix composed with thermal conductances such that:

$$G_{\mathrm{i,j}} = \begin{cases} 1/R_{i,j} & \text{If } i \neq j \\ -\sum\limits_{0 \leq k < n} 1/R_{i,k} & \text{Otherwise.} \end{cases} \tag{2.2}$$

$\boldsymbol{K}$ is the thermal ground conductance diagonal matrix, $\mathbf{Power}(t)$ the power dissipation vector at time t, and $\boldsymbol{T}^{\mathbf{A}} = T^{\mathrm{A}} \cdot [1, \ldots, 1]^{\intercal}$ is the ambient temperature vector.

2.1.1.2 Power Model

The total power dissipation has leakage and dynamic power components. Leakage power can be approximated to linearly increase with temperature [56, 21]. Dynamic power of a core depends on its mode (*active* or *idle*). Based on these assumptions, the power dissipation of the system is given by the following equation:

$$\mathbf{Power}(t) = \boldsymbol{\phi} \cdot \boldsymbol{T}(t) + \boldsymbol{\psi}(t) \tag{2.3}$$

where $\boldsymbol{\phi}$ is diagonal matrix of dimension n with constant coefficients, and $\boldsymbol{\psi}$ a vector with n elements. Furthermore:

$$\psi_i(t) = \begin{cases} \psi^a & \text{if node } i \text{ is a core } \textit{active} \text{ at time } t \\ \psi^i & \text{if node } i \text{ is a core } \textit{idle} \text{ at time } t \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

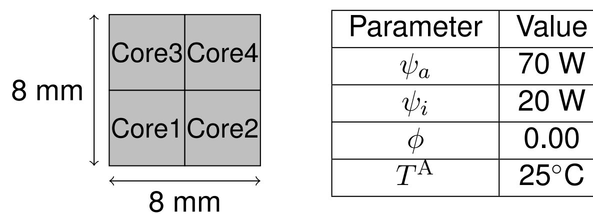
i.e. ψ^a and ψ^i are used to denote the temperature independent power dissipation of a core in *active* and *idle* mode respectively. ψ^d is used to denote $\psi^a - \psi^i$. The power dissipation of non-core nodes is zero. Therefore, $\phi_{ii} = 0$ if node i is not a core.

2.1.2 Application Model

We assume that individual tasks are sporadic. A given task τ_i is characterized Worst Case Execution Time (WCET) E_i, minimum inter-arrival time W_i, relative constrained deadline $D_i \leq W_i$. The set of all tasks is denoted by Π. In a later setting, we also consider mixed-critical applications.

2.1.3 Sample platform architecture

Unless otherwise stated, we use the following representative platform for all empirical results, evaluations and examples:



Parameter	Value
ψ_a	70 W
ψ_i	20 W
ϕ	0.00
T^{A}	25°C

Figure 2.1: System parameters for the sample architecture

The thermal parameters ($\boldsymbol{C}$, $\boldsymbol{G}$ and $\boldsymbol{K}$) are acquired by simulating the sample architecture floorplan with Hotspot thermal simulator's [80] default configuration. This configuration captures the typical thermal characteristics of silicon, thermal interface material, heat spreader material, and heat sink.

2.1.4 Formalizing Thermal Isolation Servers

In this section we introduce the TI Server. which is the scheduling construct used to provide thermal isolation/protection. TI Server S_i is characterized by period P_i, utilization U_i, phase ϕ_i, and the core on which S_i executes cr_i. TI Servers are partitioned, i.e. S_i always executes on cr_i without migrating to a different core. We will refer to cr_i as S_i's *self-core*. state(S_i, t) represents the state of TI Server S_i at time t and can be either *active* or *idle*. state(S_i, t) follows a fixed periodic schedule such that:

$$\mathrm{state}(S_i, t) = \begin{cases} \textit{active} & \text{if } P_i(k + U_i) + \phi \geq t \geq k \cdot P_i + \phi \\ \textit{idle} & \text{otherwise} \end{cases}$$

where k is any non-negative integer. Each server is assigned a set of sporadic tasks. When server state is *active*, Earliest Deadline First (EDF) is used to schedule individual tasks. Nothing is executed when server state is *idle*. Multiple servers can execute on the same core as long as their active times (time range where server state is active) do not overlap.
In order to provide thermal isolation, a TI Server is assigned a thermal budget, $\boldsymbol{\Lambda^i}$ which is the upper bound on temperature increase caused by all tasks executed by S_i. $\boldsymbol{\Lambda^i}$ is a function of P_i, U_i and cr_i. In this section, we will first compute $\boldsymbol{\Lambda^i}$ and analyze how changing P_i and U_i changes $\boldsymbol{\Lambda}^i$. We will then study how timing guarantees can be given using TI Server and how multiple TI Servers can be executed together without violating the thermal constraint. For proof of the proposed concepts, please refer to [5].

2.1.4.1 Thermal properties of TI Servers

Let $\boldsymbol{\Theta}(t, S_i)$ represent the thermal component of server S_i, i.e. temperature increase at time t due to all executions performed by S_i. Also assume that S_i never runs out of tasks to execute when its state is *active*.

Theorem 1 *The thermal component at the end of active time of TI Server S_i with $P_i > 0$ converges to:*

$$\boldsymbol{\Theta}(\lim_{k\to\infty}(P_i(k+U_i)+\phi), S_i) = (\boldsymbol{I} - \boldsymbol{e}^{\boldsymbol{A}\cdot P_i})^{-1}(\boldsymbol{I} - \boldsymbol{e}^{\boldsymbol{A}\cdot P_i \cdot U_i})\boldsymbol{T}^{\boldsymbol{\infty}}(\boldsymbol{B}^{\mathbf{cr}_i, \boldsymbol{\psi}^{\boldsymbol{d}}}) \tag{2.5}$$

Theorem 2 *The thermal component of TI Server S_i with $P_i \to 0$ converges to:*

$$\boldsymbol{\Theta}(\lim_{t\to\infty} t, S_i) = \boldsymbol{T}^{\boldsymbol{\infty}}(\boldsymbol{B}^{\mathbf{cr}_i, \boldsymbol{\psi}^{\boldsymbol{d}}}) \cdot U_i \tag{2.6}$$

Theorem 3 *The upper bound on thermal component of server S_i is given by:*

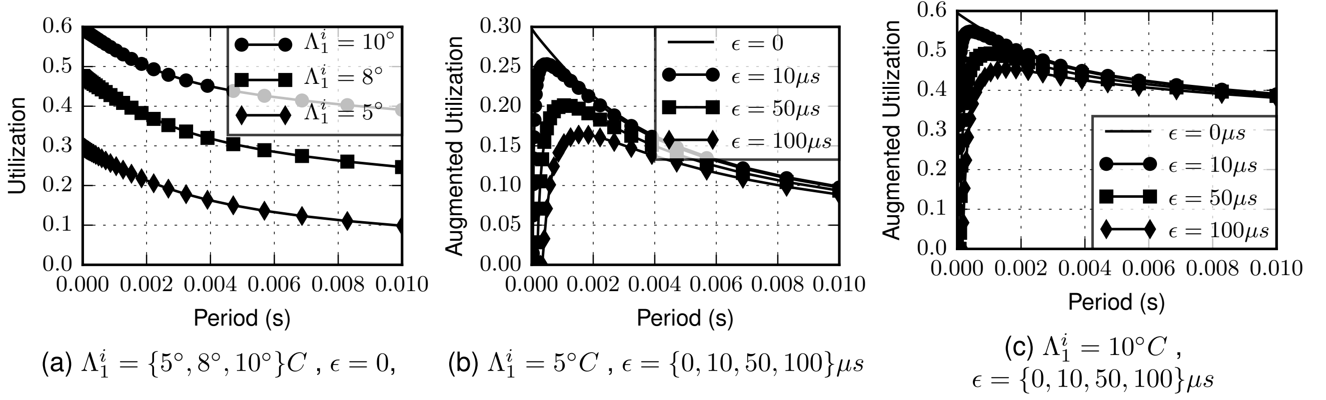
$$\Lambda^i_j = -A^{-1}_{j,\mathrm{cr}_i} \cdot \Theta(t_e, S_i)_{\mathrm{cr_i}} / (-A^{-1}_{\mathrm{cr}_i,\mathrm{cr}_i}) \tag{2.7}$$

$$\textit{where } \boldsymbol{\Theta}(t_e, S_i) = \begin{cases} \textit{R.H.S of (2.5)} & \textit{if } P_i > 0 \\ \textit{R.H.S of (2.6)} & \textit{if } P_i \to 0 \end{cases}$$

We now analyze the constant thermal budget design space. Given a fixed value of self-core thermal budget $\Lambda^i_{\mathrm{cr}_i}$ of server S_i, following inequality has to be satisfied:

$$\Lambda^i_{\mathrm{cr}_i} \geq \begin{cases} [(\boldsymbol{I} - \boldsymbol{e}^{\boldsymbol{A}\cdot P_i})^{-1}(\boldsymbol{I} - \boldsymbol{e}^{\boldsymbol{A}\cdot P_i \cdot U_i})\boldsymbol{T}^{\boldsymbol{\infty}}(\boldsymbol{B}^{\mathbf{cr}_i, \boldsymbol{\psi}^{\boldsymbol{d}}})]_{\mathrm{cr}_i} & \text{if } P_i > 0 \\ T^{\infty}(\boldsymbol{B}^{\mathbf{cr}_i, \boldsymbol{\psi}^{\boldsymbol{d}}})_{\mathrm{cr}_i} \cdot U_i & \text{if } P_i \to 0 \end{cases} \tag{2.8}$$

For the $P_i > 0$ case, we have a continuous range of $\{P_i,\ U_i\}$ tuples that satisfy (2.8).



(a) $\Lambda_1^i = \{5°, 8°, 10°\}C$, $\epsilon = 0$, (b) $\Lambda_1^i = 5°C$, $\epsilon = \{0, 10, 50, 100\}\mu s$ (c) $\Lambda_1^i = 10°C$, $\epsilon = \{0, 10, 50, 100\}\mu s$

Figure 2.2: Period/Utilization trade-off when core 1 thermal budget for TI Server S_i executing on core 1 is fixed.

Theorem 4 *Given fixed self-core thermal component budget $\Lambda^i_{\text{cr}_i}$ of S_i, the highest utilization point occurs when $P_i \to 0$.*

Now we empirically look at the constant self-core thermal component design space. Fig. 2.2a shows the tradeoff between server period and utilization. The server is run on core 1 and the thermal budget of core 1 is fixed. We present results for three different values of the fixed self-core thermal budget $[5, 8, 10]$. For each of these self-core thermal budgets, the thermal budgets of other cores are calculated using (2.7). As stated by Thm. 4 and illustrated by Fig. 2.2a, it is preferable to have the server period as low as possible. However, such a scheduling scheme is hardly practical since it incurs a high number of preemptions. To take this into account, we introduce a server overhead ϵ. ϵ time is wasted when a server switches from idle to active state. The power dissipated during ϵ is ψ_a. Therefore, the server utilization which can be used to execute tasks becomes:

$$U_i(\epsilon) = \frac{\max(P_i U_i - \epsilon, 0)}{P_i} \tag{2.9}$$

We call $U_i(\epsilon)$ the *augmented utilization* of server S_i. When $\epsilon > 0$, the optimal server period is > 0. Fig. 2.2b and Fig. 2.2c show the tradeoff between the augmented utilization and period for different values of ϵ and Λ_1^i. In general, the optimal point (maximum utilization) moves to the right (higher period values) when ϵ is increased and moves to the left when self-core budget is increased.

2.1.5 Timing guarantees

In this section, we cover how timing guarantees can be provided using TI Servers. As stated earlier, TI Servers execute tasks using EDF in their active time slots. Under EDF scheduling, a well known schedulability test is based on service bound and demand bound functions [78, 79]. These functions have the following definitions:

Supply Bound Function: Supply Bound Function (SBF) lower bounds the amount of supplied execution within any time window of fixed length l. The SBF for TI Server S_i is given by:

$$\text{sbf}(S_i, l, \epsilon) = \lfloor l/P_i \rfloor \cdot P_i \cdot U_i(\epsilon) + \max\{l - P_i(1 - U_i(\epsilon)) - \lfloor l/P_i \rfloor \cdot P_i, \ \ 0\} \tag{2.10}$$

Demand Bound Function: Demand Bound Function (DBF) upper bounds the minimum amount of compulsory execution within any time window of fixed length l. For a given sporadic task τ_j we can compute DBF using the following equation:

$$\text{dbf}(\tau_j, l) = \max\left\{\left(\left\lfloor \frac{l - D_j}{W_j} \right\rfloor + 1\right) \cdot E_j, \ \ 0\right\} \tag{2.11}$$

Theorem 5 *(condition (1) from [79]) Suppose that Π_i is the set of tasks assigned to TI Server S_i. A necessary and sufficient condition for meeting deadlines of all tasks in Π_i is:*

$$\mathrm{sbf}(S_i, l, \epsilon) \geq \sum_{\tau \in \Pi_i} \mathrm{dbf}(\tau, l) \qquad \forall l \geq 0 \tag{2.12}$$

Theorem 6 *Given two TI servers S_1 and S_2 with $U_1(\epsilon) \geq U_2(\epsilon)$ and $P_1 \leq P_2$ and $P_2/P_1 \in Z^+$, then $\mathrm{sbf}(S_1, l, \epsilon) \geq \mathrm{sbf}(S_2, l, \epsilon)\ \ \forall l$.*

The implication of Thm. 6 is that, keeping the augmented utilization the same, low harmonic server periods are *always* better in-terms of schedulability. This means that, for a given self core thermal budget and ϵ, all harmonic server periods greater than the period that yields maximum utilization (see figures 2.2b and 2.2c), are sub-optimal.

2.1.5.1 Composing multiple servers

Theorem 7 *Given n TI Servers, the maximum temperature increase due to their execution is upper bounded by:*

$$\sum_{1 \leq i \leq n} \mathbf{\Lambda}^i \tag{2.13}$$

Theorem 8 *Suppose that $\boldsymbol{T}^{\mathbf{\Delta}}$ is the thermal constraint, and the platform is only executing n TI Servers. In this setting, we have the following sufficient thermal feasibility condition:*

$$\left[\sum_{1 \leq i \leq n} \mathbf{\Lambda}^i\right]_j \leq \boldsymbol{T}_j^{\mathbf{\Delta}} - [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_j \qquad \forall j \in M \tag{2.14}$$

Theorems 7 and 8 are the direct result of thermal composability of TI Servers. This simplifies the application design process under thermal constraints. Effectively, it allows us to individually design TI Servers for subsets of tasks that constitute the entire application. Each TI Server can be designed relatively independently of other servers; where only (2.14) has to be checked to guarantee adherence to thermal constraint. The simplicity of design process is also illustrated in the following text, where we present a heuristic for designing TI Servers; and in Section 2.2 of SAFURE Deliverable D3.3 where we illustrate how TI Servers can be used to provide thermal protection to a mixed-critical application running on a hardware test-bed.

2.1.6 Designing thermal Isolation servers

This section proposes a heuristic for designing TI Servers such that the peak temperature is minimized while adhering to timing constraints. Note that due to the compositional nature of thermal component based analysis used by TI Servers, the design process is relatively low complexity compared to exhaustively evaluating all combinations of scheduling options. The proposed heuristic has two stages:1) Optimal partitioning of tasks to cores. 2) Optimal server configuration search.

2.1.6.1 Optimal task partitioning

In the partitioning phase, we assign tasks to cores *optimally* assuming TI server period of 0. This partitioning is done using a Mixed-Integer-Linear-Program (MILP).
In Formulation 2.1, we assign tasks for cores such that the minimum temperature difference between threshold temperature and the maximum temperature of a given core is maximized. The constraints state that: (2.15) each task has to be assigned to exactly one core, (2.16) utilization of each core ($\mathrm{CoreUtil}$) is the sum of utilization of tasks assigned to that core, (2.17) utilization of each core is ≤ 1 for timing feasibility, and (2.18) assignment of thermal component using (2.6) assuming a *fluid* system.

Variables:

$\alpha_{i,j} = \begin{cases} 1, & \text{if task } \tau_i \text{ is assigned to core } j \\ 0, & \text{otherwise} \end{cases}$

$\Theta_i =$ Thermal component of core i

$\text{CoreUtil}_i =$ Utilization of core i

Objective:

$$\text{maximize} \min_{1 \le i \le m} \left(T_i^{\Delta} - [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\text{idle}})]_i - \Theta_i \right)$$

Constraints:

$$\sum_{1 \le j \le m} \alpha_{i,j} = 1 \qquad \forall \tau_i \in \Pi \qquad (2.15)$$

$$\text{CoreUtil}_j = \sum_{1 \le i \le |\Pi|} \left(\frac{C_i}{W_i} \cdot \alpha_{i,j} \right) \qquad \forall j \in M \qquad (2.16)$$

$$\text{CoreUtil}_j \le 1 \qquad \forall j \in M \qquad (2.17)$$

$$\Theta_j = \frac{\psi^d}{C_{j,j}} \cdot \sum_{i \in M} -\boldsymbol{A}_{j,i}^{-1} \cdot \text{CoreUtil}_i \qquad \forall j \in M \qquad (2.18)$$

Formulation 2.1: MILP for optimal thermal partitioning of sporadic tasks

2.1.6.2 Optimal server configuration search

Next we employ a search strategy to find the server configurations that meet the timing requirements of the partitioned tasks. For the ith core, let us suppose that the utilization after partitioning is U_i^* and the server preemption overhead is ϵ. We consider server periods in the range $[\frac{\epsilon}{1-U_i^*}, P^{\max}]$, in steps of 0.01. $P^{\max}$ is specific to a given problem, and in our evaluations, was set statically to $2ms$. However, dynamic allocation of $P^{\max}$ based on self-core thermal budget is also possible. For each period, we find the minimum TI Server utilization such that the schedulability condition of Theorem 5 is satisfied. This gives us a set of timing feasible TI Servers. Out of this set, we select the configuration which has the lowest self core thermal component.

2.2 Thermal Covert Channel

The thermal covert channel transmission rates reached in previous work were achieved under laboratory conditions by mitigating interfering system characteristics [60, 8]. This leaves the open question of whether the thermal covert channel can still be considered a threat in a real scenario. In this deliverable, we extend our investigations of the thermal covert channel by demonstrating the potential of this threat by leaking sensitive information through the thermal covert channel in a real attack scenario.

2.2.1 Extended Robustness Analysis

The power consumption of a CPU is correlated to its heat generation. By controlling how much power a core is currently using, it is possible to influence the temperature of that core. By inducing variations in the power consumption and consequently variations in the temperature of the CPU, the sending application (*source*) app can transmit data through the thermal covert channel. In order to increase the power consumption, a *source* generates utilization on the cores of the system. The transmitted symbols can then be encoded in different utilization patterns and decoded from the resulting temper-

ature variations. In an ideal scenario, the thermal behaviour can be entirely controlled by the *source*, however in modern systems this is not the case. Power and thermal management, as well as other system performance tools, change the way the *source* affects the channel. Other processes may also impose CPU utilization creating thermal noise.
In a realistic scenario, all the factors that can influence the thermal behaviour of a system have to be taken into account as we cannot alter the environmental conditions. As an extension to the work presented in D3.1, we analyze these influence factors in detail and show how we can deal with them in a real environment.

2.2.1.1 Thermal Noise

Thermal noise can be caused by two main factors: changes in the device surrounding temperature and other processes generating high load on the CPU. In a laboratory setup we can keep the utilization noise caused by other processes to a minimum, allowing just the core processes of the Operating System (OS) to run. The temperature changes are then almost entirely caused by the *source*.
In a real attack scenario the same condition of low base utilization can be reached when the system is not actively used, such as a phone charging at night or a laptop left turned on in the office during the weekend. We consider the covert channel attack to happen during those low usage periods. We expect that the remaining background noise will not be able to disturb the thermal channel, meaning that no high external utilization will occur over a long period of time.
In order to exclude thermal noise generated by the receiving application (*sink*), the rate at which the temperature measures are sampled and the execution of the sampling itself cannot generate a high load on the CPU. This means that the sampling rate of the *sink* needs to be adapted accordingly.

2.2.1.2 Fans

Fans are the most commonly used active cooling devices in computing hardware, while the system can alter the efficiency of heat disposal by changing the fan speed. Modern systems usually change the fan speed depending on the current temperature, altering the behaviour of the thermal channel with correlated thermal noise.
To exclude interference from changes in fan speed, in a laboratory setup the fan speed is fixed to the maximum level during the experiments. When fixing the fan speed is not possible, the transmission scheme has to be adapted to changing thermal dynamics. Since the changing thermal dynamics caused by the fan speed is correlated to the actual temperature, the *source* could adapt the transmission scheme depending on the measurement of the thermal sensors. However, we assume that the *source* has no access to the temperature readings. This means that the *source* does not have the ability to create a back-channel, i.e. cannot use the output of the channel to improve its input.

2.2.1.3 Core Pinning

Modern multicore systems take advantage of their multiple cores to increase the system performance. This can result in processes being shifted between cores, which also causes a shift of the thermal dynamics from one core to another. To exclude these events from happening, in a laboratory setup the *source* and *sink* processes can be pinned to specific cores. In a realistic attack scenario core pinning is not possible.
Since the attack is happening on a not actively used system, only the core on which the *source* is running will have strong temperature variations. We already defined the all-cores in the work presented in deliverable D3.1, which consists of the thermal channel created by observing the summed temperature of all cores. All the thermal noise generated by other processes are also summed up in the all-cores along with the temperature variations of the transmission. Thus in a system with low thermal noise, the main variations of the all-cores represent the transmitted signal. The *sink* will collect and sum the temperature measurement of all cores to decode the signal on the all-cores, as the *sink* can access all thermal sensors.

2.2.1.4 Sleep States

When the system is idle, as in the attack scenario, the CPU can enter so called sleep states to optimize energy consumption. Waking up a physical core takes some time, lowering the immediate performance and introducing delays in the execution of a process.
The system used in a laboratory setup may not be allowed to enter deep sleep states by limiting the *cpu dma latency*. This system functionality cannot be modified by our attack because changing the *cpu dma latency* requires root access. Hence the *source* will have the task to wake up the CPU before transmitting in the case all cores entered a sleep state. This can be done by generating an initial utilization on the system and not allowing longer idle periods during the transmission.

2.2.1.5 Frequency Governor

Modern devices try to optimize power consumption through Dynamic Voltage and Frequency Scaling (DVFS), which changes the CPU frequency and the supply voltage at runtime. By lowering the CPU frequency and the supply voltage, the system can decrease the amount of power consumed, while still maintaining high performance. The OS software that regulates the operating frequency is called governor. The governor uses the CPU utilization to decide which frequency to set, depending on its policy. Since different frequency levels produce different amount of heat, in a system with varying frequency the thermal behaviour will have a more complex correlation with the utilization generated by the running processes.
In the systems used in a laboratory setup, the frequency can be fixed to exclude interference from the governor's behaviour. In the presented scenario, processes won't have the rights to fix the frequency, so the effects of governors on the system have to be taken into account. Different devices have different default governors on them, which may hinder or enhance the thermal dynamics of the thermal channel. However, frequency governors base their policy for the operating frequency on the CPU utilization, which *source* can influence. This means that we can adapt the transmission scheme to the frequency governor running on the target device.

2.2.1.6 Scheduling

Scheduling is used by the OS to assign resources to the different processes. For example, schedulers can work with a priority system, where higher priority processes will have precedence over lower priority ones, in terms of access to resources. Moreover, a scheduler may force a process to leave resources to another one, in order to prevent starvation or to support critical system processes. Thus the scheduler can cause timing jitters in the channel, which means that there are variations between the expected and the actual execution timing of processes.
To exclude interference from the scheduler, First In, First Out (FIFO) scheduling can be used in a laboratory setup with *source* and *sink* at highest priority. Using FIFO scheduling, any process that uses a resource cannot be interrupted until completion by processes with the same or lower priority. FIFO scheduling and maximum priority allows *source* and *sink* to work undisturbed.
In the real case, the attacker cannot change the scheduling algorithm, which means that the default has to be used. However, in an idle system, timing jitter caused by the scheduling algorithm should be sufficiently small to be corrected by a synchronization effort in the transmission.

2.2.2 Results from a realistic attack scenario

The thermal covert channel attack is performed on a platform representative of a modern commercial business laptop. We use a Lenovo ThinkPad T460s, with an Intel Core i7-6600U CPU that has two physical cores, each running two hyper-threads resulting in four logical cores. The OS on the device is Ubuntu 15.10, and to grant isolation between the two applications, Oracle VM VirtualBox is used. The used Virtual Machine (VM) runs the same OS (Ubuntu 15.10), has two cores and the CPU execution cap is set to 100%. Other notable specifications of the laptop that influence the thermal

covert channels are the default frequency governor, which is *intel p-state* powersave, and the default scheduler called Completely Fair Scheduler (CFS).
A message sent by the *source* through the thermal channel will be composed of one or more packets. Each packet is structured as in figure 2.3, starting with a synchronization pulse, in short *sync pulse*, followed by the payload composed of the data bits and error detection coding. The sync pulse is used

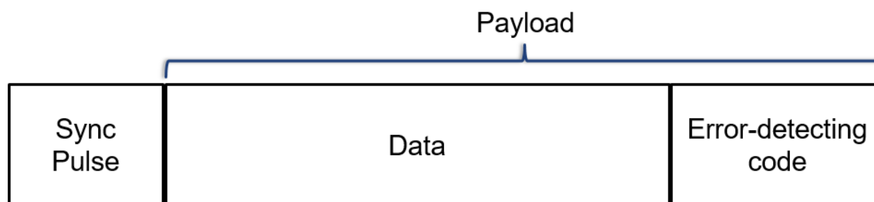


Figure 2.3: Packet structure

during the decoding of the thermal trace to localize the packets in order to extract the payload. This initial pulse needs to be distinguishable from the rest of the signal, but the longer it is, the more the effective transmission rate is reduced. Another aspect of the transmission is that the implementation of sync pulses helps us to correct timing jitters inside packets. Timing jitters occur because *source* and *sink* use different timing sources, which are not synchronized, as the *source* is running within the VM and the *sink* on the host OS. The two timing sources used are different and timing jitter on the VM process causes synchronization issues between *source* and *sink*.
Payload symbols are divided between data and error detection. We use Cyclic-Redundancy-Check (CRC) error-detecting code inside the packets sent in our experiments. The CRC coding uses a fixed polynomial to calculate the remainder of the polynomial division on the packet's data. The remainder is then attached at the end of the packet as a check value, which is recomputed and compared after the transmission. CRC is good at detecting random errors generated by noise in transmission channels and is easy to implement. Using this transmission scheme we can reach bitrates as high as 20 bps with less than 5% bit error rate (BER).
We then successfully performed an attack in a real case scenario exploiting the thermal covert channel, successfully leaking the $13'432$ bits of a *id_rsa* file containing a private SSH key. The key was correctly decoded after an average of six transmissions of all packets, which has a resulting average goodput[1] of 1.358 bps.

2.3 Frequency Covert Channel

One of the most commonly used techniques for power optimization is DVFS, due to the relatively low impact of DVFS on the user experience. As previously mentioned, DVFS allows the optimization of the energy consumption by changing the operating frequency and the supply voltage of the cores, according to performance needs. Typically, a software component in the operating system of a device, denoted as governor, is responsible for selecting the frequency based on a specific predefined policy, the associated parameters, and run-time information. In this section, we investigate the implications of frequency scaling on the security of mobile multicore systems, i.e., the possibility to establish a covert channel between two isolated applications with a sufficient bandwidth and a low error rate to break the security paradigm of permission separation and application isolation.
Similar to previous research on covert channels [8] or D3.1, we consider two colluding applications trying to establish a covert channel on a multicore system (see figure 2.4). The first application, the *source application* (`src`) running on *core 0*, has access to restricted data but no access to the communication interfaces. The restricted data could have very high security demands, for example a cryptographic key, which only needs a channel with a very low bandwidth to be leaked. The second

[1]The goodput is the application level throughput of a transmission, i. e. the number of bits that can really be used by a application after removing all transmission overhead and errors from the received data

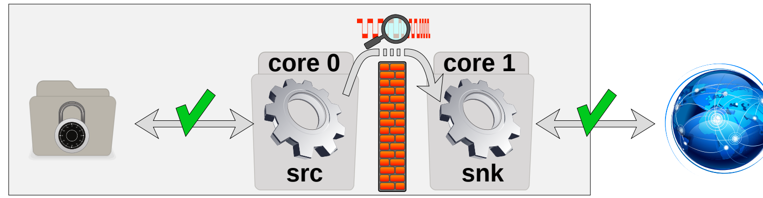


Figure 2.4: The source application (src) has access to restricted data, while the sink application (snk) has access to the internet. Although source and sink are isolated from each other, they manage to establish a covert channel by observing the frequency of the cores, thus compromising the security paradigm of permission separation and application isolation.

application, called *sink application* (snk), runs on *core 1* and has full access to a communication interface but no permitted access to the restricted data available to src. The two applications are isolated from each other and, according to this security principle, are not allowed to communicate. We consider a common setting in current multicore architectures; a processor with multiple cores in the same frequency and voltage domain, i. e., the cores share the same voltage and frequency level at any point in time. Further, we assume that the system is idle, hence only the OS and its vital processes are active at the time of the attack. As the system is idle, the behaviour of DVFS will depend on the utilization generated by the source application and scale the frequency accordingly. If the sink application is able to detect the frequency changes induced by the source application, the applications can establish a covert channel. From now on we will refer to this channel as *frequency covert channel*, as the key shared resource is the frequency of the cores.

2.3.1 Frequency Scaling in Linux

We base our investigations and the communication model of the frequency covert channel on the implementation of frequency scaling in the Linux Kernel. Linux is used among a diverse range of devices, i.e. Linux can be found in server or desktop systems, laptops, or powering Android on smartphones or tablets. In addition, the open source nature of the Linux Kernel and its components allows us to review the code which handles frequency scaling.
Examples of CPU frequency driver implementations in Linux are the *intel p_state* or the *acpi-cpufreq*[2] driver. In this work, we consider the acpi-cpufreq driver, which is used in Android systems as well as in Ubuntu and similar desktop OSs. Figure 2.5 gives a simplified overview of its main components, which we discuss in detail in this section.

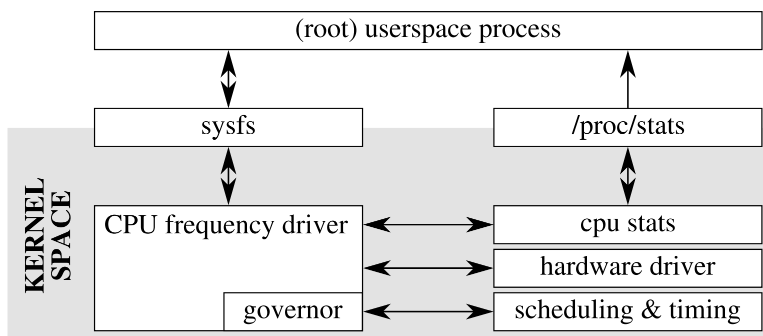


Figure 2.5: A simplified structure block diagram of the CPU frequency control in the Linux Kernel.

[2] http://www.acpi.info

2.3.1.1 The CPU frequency driver

The CPU frequency driver operates as an interface to all the other kernel components, most importantly the sysfs nodes and the hardware driver. One responsibility of the CPU frequency driver is to maintain the sysfs nodes used to control frequency scaling, which we describe in subsection 2.3.2. Furthermore, the CPU frequency driver instructs the hardware driver to set the frequency for each frequency domain, interacts with the scheduling unit and passes information like the utilization statistics from the Kernel to the governor.

2.3.1.2 The governor

The governor should be called periodically with the timer period T_s, also called *sampling period*, to determine the new frequency f_{set} for every frequency domain. T_s is determined as $T_s = \max(T_{s,min}, T_{s,U})$, where $T_{s,min}$ is the minimum possible sampling period and $T_{s,U}$ is the sampling period set via a sysfs node. While $T_{s,min}$ depends on hardware and kernel limitations and is typically in the range of tens of milliseconds, $T_{s,U}$ can be changed by any process in the userspace with sufficient permissions. The default value for $T_{s,U}$ is set by the CPU frequency driver.
Due to differences in hardware and user needs, many customized frequency governors are available for devices based on open source platforms like the Linux kernel and acpi. The multitude of governor options allows the user to set the trade-off between performance and battery lifetime. Different governors can vary in terms of static characteristics like minimum and maximum frequency, but also in their frequency dynamics. In this work, we will focus on the *conservative governor* which is one of the most commonly used governors in mobile, battery powered systems.
Roughly speaking, the conservative governor uses the average core utilization in the past time interval to determine the new frequency f_{set}. If the utilization is below or above a certain threshold, the governor reduces or increases the frequency, respectively. The new frequency target f_{new} is calculated according to Equation 2.19. In our calculations, f_{cur} is the current frequency target and Δf is the frequency scaling step. Δf is defined as $\Delta f_{rel} \cdot f_{\max}$ and relative frequency step Δf_{rel} is set in the corresponding sysfs node.
To scale the frequency, the governor uses the relationship between idle time t_i and the total measurement time t_m, which is equal to the time elapsed between the last and the current call time of the governor.

$$f_{new} = \begin{cases} f_{max} & \forall \quad \frac{t_i}{t_m} < I_{low} \wedge f_{max} < f_{cur} + \Delta f \\ f_{cur} + \Delta f & \forall \quad \frac{t_i}{t_m} < I_{low} \wedge f_{max} \geq f_{cur} + \Delta f \\ f_{cur} & \forall \quad I_{low} \leq \frac{t_i}{T_s} \leq I_{high} \\ f_{cur} - \Delta f & \forall \quad \frac{t_i}{t_m} > I_{high} \wedge f_{min} \leq f_{cur} - \Delta f \\ f_{min} & \forall \quad \frac{t_i}{t_m} > I_{high} \wedge f_{min} > f_{cur} - \Delta f \end{cases} \tag{2.19}$$

For simplicity, let us call the term t_i/t_m *idleness*. The governor adapts the frequency depending on the lower idleness threshold I_{low} and the higher threshold I_{high}, which can also be set via sysfs nodes. Equation 2.19 shows that there are three main cases: (i) If the idleness is lower than I_{low}, the frequency is increased by Δf. In case the target would be higher than f_{max}, f_{new} is set to f_{max}. (ii) If the idleness is between I_{low} and I_{high}, the frequency is not changed. (iii) If the idleness is higher than I_{high}, the frequency is decreased by Δf. Whenever the target would be lower than f_{min}, f_{new} is set to f_{min}.
After calculating f_{new}, the CPU frequency driver selects f_{new} from the discrete set of frequency levels that are available on the system. To select the frequencies, the CPU frequency driver applies one of two rules: (A) If the frequency is scaled up, $f_{new} > f_{cur}$, f_{set} is set to the highest available frequency below or at the target. (B) If the frequency is scaled down, $f_{new} < f_{cur}$, f_{set} is set lowest available frequency at or above the target. After the frequency has been changed to f_{set}, f_{cur} is set to f_{new}.

2.3.2 Userspace Interfaces

The sysfs and the /proc/stats pseudo file system nodes allow processes to access frequency scaling relevant information from userspace. On the one hand, detailed information on the CPU usage statistics can be

read via the `/proc/stats` node. On the other hand, the `sysfs` nodes offer not only information about the governor but also give processes with root permissions the opportunity to change the governor behaviour during runtime.
The `sysfs` holds a so-called *policy*[3] for every frequency domain. Among others, the policies contain following parameter nodes:

- All cores affected by the policy in `affected_cpus`.
- The current frequency f_{cur} in `scaling_cur_freq`.
- The minimum frequency f_{min} in `scaling_min_freq`.
- The maximum frequency f_{max} in `scaling_max_freq`.
- All possible frequency steps in `scaling_available_frequencies`.
- All available governors, `scaling_available_governors`.
- Currently used driver and governor in `scaling_driver` and `scaling_governor` respectively.

Furthermore, the `sysfs` contains nodes for the global governor and frequency driver settings which apply to all frequency domains[4]. The global parameters contain, e. g. (i) the current sampling period in `sampling_rate`, (ii) the minimum sampling period in `sampling_rate_min`, and (iii) the utilization thresholds in `down_threshold` as $(1 - I_{high}) \cdot 100$ and `up_threshold` as $(1 - I_{low}) \cdot 100$, (iv) the size of the frequency scaling steps in `freq_step` as $\Delta f_{rel} \cdot 100$.

2.3.3 Threat Model

We consider the scenario presented in Figure 2.4. The sink application `snk` measures and records the current frequency. The resulting frequency trace is then forwarded over a communication interface (network) to an attacker device, which handles the message decoding off-line. By doing the message decoding off-line, we can minimize the interference between the `snk` application and the communication channel and also minimize the complexity of `snk`. Further, we assume that the attacked device is idle or only lightly utilized during the attack, e.g., a hand-held device like a smartphone during the night sitting on a table or a laptop in an empty office during a weekend. Therefore, the source application `src` and the sink application `snk` can wait until the average system utilization is low and will presumably stay low for some time.
The target platforms for this work are mobile devices with shared frequency domains among multiple cores. This architectural feature is present in almost all current processor architectures that are used in computing devices such as mobile phones, tablet computers, laptops or servers. For instance, today's Intel Core mobile processors, e. g. based on the Haswell architecture, feature one frequency domain for all physical cores. As another example, big.LITTLE architectures in mobile phones typically feature one frequency domain for all LITTLE and one frequency domain for all big cores, e. g. the Samsung Exynos 5422 Octa.
Cores in the same frequency domain can inspect the shared frequency with two methods: (i) reading system files, or (ii) timing measurements. On Linux, method (i) can easily be blocked by requiring elevated privilege levels to read the appropriate system file[5]. In subsubsection 2.3.4.2 we show the performance of a frequency covert channel implementation using timing measurements, without the need for elevated privileges.

2.3.3.1 Frequency Covert Channel Model

The proposed system abstraction model is composed of three parts: the Input Stage, the Frequency Channel and the Output Stage.
Input Stage: In this stage, the input bitstream is converted to a symbol stream.
Frequency Channel: The Frequency Channel has three characteristics:

- It is time-discrete, as the governor is supposedly called periodically.
- It is value-discrete, as there is only a limited set of discrete frequencies which can be used.
- It is noise-free, as the frequencies are taken from a discrete set and cannot be changed continuously.

Output Stage: Here the measured frequency trace is converted into an output bitstream.
Our model enables us to better understand the frequency covert channel and to determine a capacity bound. We also refer to this model when we design our test environment and the experiments to evaluate the channel.

[3] `/sys/devices/system/cpu/cpufreq/policy$i/` in Ubuntu OSs, where i is the frequency domain number.
[4] `/sys/devices/system/cpu/cpufreq/[governorname]/` in Ubuntu OSs
[5] I.e. with the acpi-cpufreq drivers `/sys/devices/system/cpu/cpu$i/cpufreq/scaling_cur_freq`

Param.	Value	Param.	*Laptop*	*Hand-Held*
Δf_{rel}	5%	T_s	80 ms	100 ms
I_{low}	20%	$f_{\min}$	0.8 GHz	0.2 GHz
I_{high}	80%	$f_{\max}$	2.4 GHz	2.0 GHz
f-levels in 0.1 GHz steps			w/o $\{1.2, 2.0\}$GHz	all

Table 2.1: Parameters of the conservative governor and the characteristics of the platforms *Laptop* and *Hand-Held*.

2.3.4 Threat Potential Indicators

To quantify the threat potential of the frequency covert channel we need to first find a capacity bound. As a second step, we show an implementation of the channel to empirically evaluate the performance. Our experiments are carried out on two diverse hardware platforms that are representative for two kinds of mobile devices:

1. A Lenovo ThinkPad T440p laptop, based on a 4th generation Intel Core i7-4710MQ quad-core processor. It can be clocked at frequencies in the range from 800 MHz to 2.4 GHz in 15 frequency levels, excluding the Intel Turbo Boost;
2. An Odroid-XU3 board, featuring a Samsung Exynos 5422 System-on-Chip (SoC) with an ARM big.LITTLE processor with two quad-core clusters of Cortex-A7 and Cortex-A15 cores, respectively. The LITTLE cluster is clocked at frequencies in the range of 200 MHz to 1.4 GHz in 13 frequency levels; the big cluster in a range of 200 MHz to 2.0 GHz in 19 levels.

In the rest of the section, we refer to platform 1 as *Laptop* and to platform 2 as *Hand-Held*. While *Laptop* is representative for current business laptops, *Hand-Held* is representative for hand-held devices (i.e. tablets or smartphones). As we expect a similar behaviour across different Ubuntu versions, in this experiment *Laptop* is running Ubuntu 16.04.1 LTS and *Hand-Held* is operating on Ubuntu 14.04.4 LTS.
Source and sink application are not executed on the same core, but on two separate cores which share a frequency domain. Finally we note that during all the experiments the systems are only running the source, the sink and the default system services of Ubuntu. Furthermore, we do not alter the standard settings of the governor, which are presented in Table 2.1.

2.3.4.1 Determining the Capacity Bound

MacKay [59, Chapter 17] presents Equation 2.20, which shows how to calculate an upper bound C for the capacity per channel use of a noise-free channel.

$$C = \lim_{N\to\infty} \frac{1}{N} \log M_N [bit] \tag{2.20}$$

Here, N denotes the number of uses of the channel, and M_N denotes the number of distinct and feasible symbol series that could be sent by using the channel N times.
Inserting the parameters for the *Laptop* platform yields an upper bound on the channel capacity of $C = 0.972$ bits per channel use. If we apply this scheme to our second platform, *Hand-Held*, we get a capacity of $C = 0.982$ bits per channel use.
Knowing the capacity bound and the sampling period T_s of the governor, we can calculate the theoretical bandwidth of the channel as shown in Equation 2.21.

$$B = \frac{C}{T_s} \tag{2.21}$$

As outlined in Table 2.1, *Laptop* has a sampling period $T_s = 80$ ms by default and *Hand-Held* $T_s = 100$ ms, which yields a maximum bandwidth $B_{\max} = 12.15$ bps and 9.82 bps, respectively. As we are considering the frequency covert channel to be noiseless, at the same time, this also equals the maximum throughput.
Finally, note that these calculated upper capacity bounds cannot be achieved in a practical setting, as they are based on idealized conditions. The bounds assume a perfect transmission scheme, no errors due to interferences of other processes, a perfect synchronization between the source and sink applications and no implementation artifacts of the governor.

Packet Payload	Payload Bits per Trace	*Laptop* Ref.	*Hand-Held* Ref.
8 bit	1600 bit	1.79 bps	1.43 bps
16 bit	3200 bit	2.27 bps	1.82 bps
32 bit	6400 bit	2.63 bps	2.11 bps
64 bit	12800 bit	2.86 bps	2.29 bps

Table 2.2: Experiment packet payload, the corresponding number of bits per experiment trace and the throughput on the reference platforms.

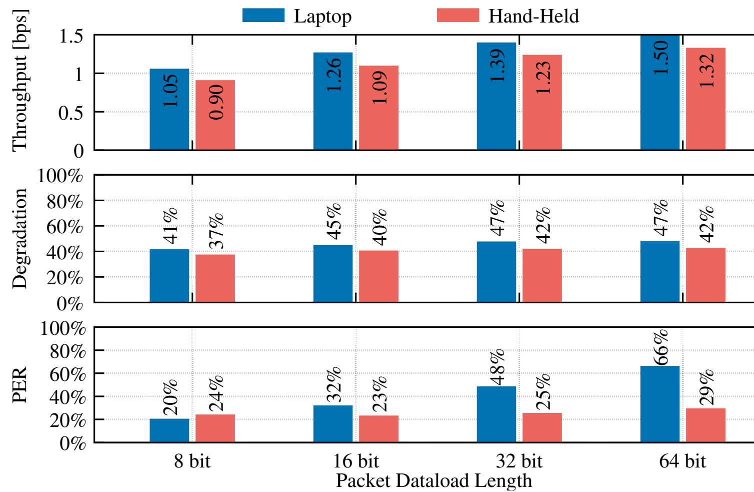


Figure 2.6: Results of the experimental evaluation of both platforms. The upper diagram sows the achieved throughput in bps for both platform, depending on the packet size. The middle diagram shows the percentage of throughput degradation compared to a reference platform, and the bottom diagram presents the packet error rate (PER).

2.3.4.2 Experimental Evaluation

Our transmission implementation is, similar to the final implementation of the thermal covert channel, based on dividing the input bitstream into packets with a preamble and a postamble. We evaluate the robustness of our implementation for different payload length of the packets, i. e. 8, 16, 32 and 64 bit, for the two platforms *Laptop* and *Hand-Held*. For every packet length we send 5 traces of 200 packets each with random payload bits, which means that every trace could contain multiple transmissions of a 512 bit elliptic-curve cryptography key [7], see Table 2.2.
We compare the results of our experiments with two theoretical reference platforms. These reference platforms have the same parameters as the real platforms, but we assume that no frequency covert channel and governor implementation artifacts occur.

$$B_{ref} = \frac{payload}{T_s \cdot (CU_{PRE} + CU_{BIT} \cdot payload + CU_{POST})} \tag{2.22}$$

Therefore, we calculate the throughput of these reference platforms using Equation 2.22, in which the $payload$ is the number of bits in the packet. CU_{PRE} is the number of channel uses for the preamble and CU_{POST} for the postamble. The number of channel uses per bit is denoted as CU_{BIT}. The respective bandwidth for each packet length, and therefore the throughput, is given in Table 2.2 for the two reference platforms.
Figure 2.6 illustrates the results of the experimental evaluation. The upper diagram shows the achieved throughput in bps, calculated as an average of each single packet throughput for all packets that have been transmitted without error. The packet throughput is calculated by dividing the number of payload bits in a packet by the time needed to send the whole packet, including preamble and postamble. The middle diagram presents the degradation of the throughput between the reference and the real platform, i. e. the percentage of throughput loss. The packet error rate (PER) in per cent is illustrated in the bottom diagram.
For *Laptop* we can observe that the throughput increases by 0.45 bps with increasing packet length, the throughput degradation only increases marginally by about 6%. The increase in throughput degradation is caused by the higher number of bits per packet, as this also increases the likelihood for problematic frequency

scaling behaviour to occur in a given packet. Non-ideal governor behaviour has to be compensated by the source application, which slows down the transmission. The higher likelihood for non-ideal governor behaviour also increases the likelihood for errors during the transmission of a given packet, which we observe in the rising PER in respect to packet length.
The results for *Hand-Held* show a similar trend. The throughput increases with the number of payload bits per packet, similarly to *Laptop*. However, there is less throughput degradation for *Hand-Held* than for *Laptop*, which means that the transmission throughput on *Hand-Held* suffers less from the compensation measures the source application needs to apply due to non-ideal governor behaviour. Last, we can observe that the PER on *Hand-Held* does not increase significantly with the length of the packets. We chalk the lower packet error rate increase up to a combination of two reasons: (i) different kernel version of *Hand-Held* and *Laptop*, and (ii) the architecture of the processor of *Hand-Held*. *Hand-Held* is operated on kernel version 3.10.96 while *Laptop* uses 4.4.0-59-generic. Different kernel versions can cause differences in the behaviour of the governor due to implementation artifacts. Furthermore, on *Laptop* all cores share one frequency domain, whereas the LITTLE and the big cores in *Hand-Held* have separate frequency domain. Due to the fact that the *Hand-Held* will try to utilize the LITTLE cores as much as possible and only migrate processes into the big cluster if necessary, we experience less interferences for our transmission.
Interference by other processes: In our scenario, we assume that the attacked device is idle, which is a valid expectation considering the usage pattern of mobile devices. Nonetheless, other processes could still increase the core utilization and interfere with the channel. While short utilization bursts caused by background processes and idle applications can be handled easily, a high utilization floor would cause more problems. Short utilization bursts would cause a single problematic frequency scaling. We show with our implementation that single problematic frequency scalings can already be handled with very simple measures like a back channel and an appropriate transmission scheme. In contrast, if the interfering utilization is constant and high enough to force the governor to scale to a frequency higher than the lowest possible frequency, the number of reachable frequency levels is reduced. The reduction of frequency levels can only be compensated in the design of the transmission scheme or by a smart source application. Furthermore, a high utilization floor can lead to additional measurement artifacts. In conclusion, we can state that burst interference by other processes can be compensated during an attack, while permanent interference might make an attack impossible. Therefore, only by launching an attack while the device is idle maximizes the chance of a success.

2.4 Thermal Task Inference

In this section, we cover the feasibility of temperature side-channel attacks that could compromise confidentiality/security. For a use case, we profile on-chip thermal sensors while a video is being played on a smart phone platform (Dragonboard 810). The resulting thermal trace is then used to *infer* the video being played. High inference success rate indicates that thermal traces include sufficient information about the video and a side-channel attack is possible.
The rationale behind how and why videos can influence temperature is the following: fast video transitions (large number of scene cuts or fast moving scenes) require more decoding power, resulting in higher processor temperature. Therefore we expect to see temperature increase at specific points in time for different videos. To increase temperature variations, the .mkv video codec is used, which is not supported by the hardware video decoding unit in Dragonboard 810.
For the analysis presented here, two video inference strategies were developed: 1) Correlation based approach 2) Neural Network based approach.

2.4.1 Thermal Data Preprocessing

In both of the schemes, we always work with thermal traces. These traces are preprocessed before analysis on them is performed. The preprocessing performs three functions:

1. Converts the raw thermal profile into a csv file format
2. Remove temperature peaks caused by context switches. These peaks are at the beginning and end of every trace (at the start and end of every video run).
3. Traces are high-pass-filtered. This is done to remove any temperature offsets and global temperature trends. The resultant trace is a zero-mean trace.

The whole preprocessing flow is shown in Figure 2.7.

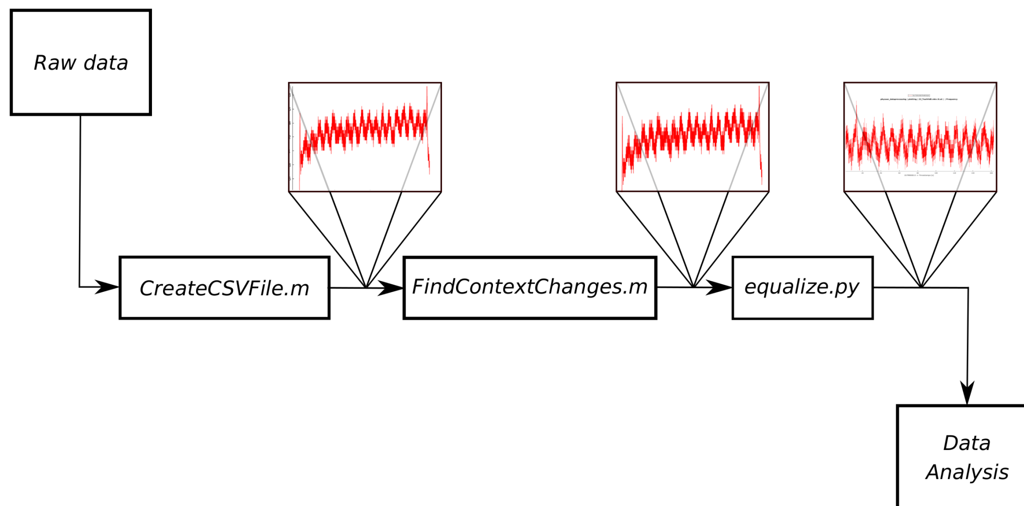


Figure 2.7: Preprocessing applied on thermal traces

2.4.2 Correlation Based Approach

There are two types of traces for the correlation based approach, the reference traces and the test traces.

2.4.2.1 Reference Traces

The reference trace of a given video (e.g. video1) is generated by running video1 multiple times and recording the corresponding thermal traces. All thermal traces are then preprocessed (formatted, context switches removed, and high-pass-filtered). After preprocessing, all the traces are averaged to compose a single reference trace for video1. By averaging, we make sure that the reference trace does not contain peaks which are caused by random fluctuations in a single trace. The reference trace is used to identify/label the test traces.

2.4.2.2 Test Traces

A test trace is the temperature profile for a single run of a given video. Like the reference traces, test traces are also preprocessed. A given test trace is compared with existing reference traces to identify the video which was being played when the test trace was recorded.

2.4.2.3 Video inference algorithm

The test traces are labeled using reference traces. The algorithm has two stages: 1) Correlation block generation, 2) cross-correlation based scoring.
Block Generation: In this phase, we identify sections of the reference trace which have high temperature variations. We call these sections *high-entropy blocks*. The rationale here is that the high-entropy blocks and the temporal distance between successive high-entropy blocks will be unique for a given video. We find these blocks by sliding a window over the reference trace and compressing the content of the window. An entropy-like measure is computed by determining the number of bits which are needed to represent the data in this block with no information loss. Using this entropy-like measurement, the algorithm selects the blocks with the highest entropy according to a configurable threshold. These blocks are used in the next phase of the algorithm.
Scoring: We use cross-correlation to identify the videos. However, the cross-correlation is only performed on the high-entropy blocks. Specifically, the cross-correlation is computed by shifting the previously determined high-entropy blocks with the right time offsets over a test trace and calculating the Pearson correlation. The final *score* of a trace is the maximum correlation value.

2.4.2.4 Results

The algorithm described in subsubsection 2.4.2.3 yields the results shown in Figure 2.8 using multiple traces of several different videos. As the maximal score of a test video, which is not video 1, is always lower than the minimum score of all video 1 trace, all video 1 traces get the right label. A drawback is that the correlation values are in a varying range for different reference videos. Therefore, choosing the appropriate threshold

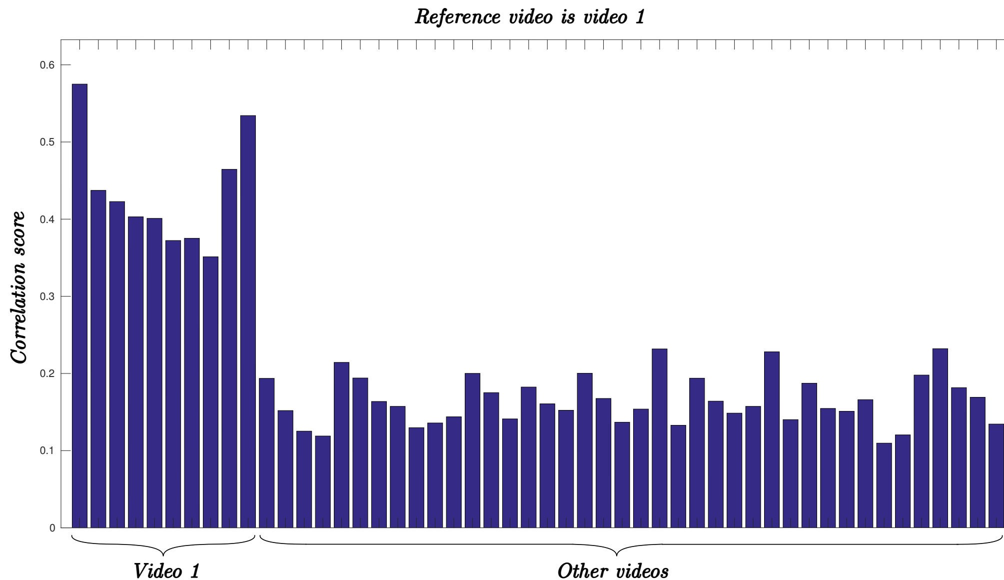


Figure 2.8: Result of the initial version of the thermal classifier, where the videos with the same content as the reference video have a higher score than the other videos.

value is crucial for this scheme. Furthermore, this approach is inflexible as the distance of the high-entropy blocks are essential, which makes it difficult to extend it on classifying non-video applications.

2.4.3 Neural Network Approach

In this subsection, the outline of a NN-based approach, which first learns the feature and then classifies full traces, is presented. For this preliminary study, we use two types of videos.

1. Black video: Three minutes of a black screen. The thermal trace has low temperature variations.
2. Alternating video: Five seconds of black screen followed by five seconds of a noise screen. This pattern is repeated for three minutes. As the noise screen has a high entropy, the processor needs more computations to decode this, and this causes the temperature to increase. Therefore, the temperature trace has a periodic oscillating pattern.

2.4.3.1 Used Layers

The following paragraphs introduce all used NN layers and give a high level description of the functionality and the meaning of the parameters.

2.4.3.1.1 Convolutional Layer

Convolution layers are often used to find patterns in data-sets. Convolutional layers are trained on the weights which are contained in the filters. To train a convolution layer, a data vector is applied to the input and convoluted with all filters. As the weights are initialized randomly at the beginning, they do not fit the input data in anyway and therefore the difference between the output and the right label is large. This difference is called the loss and it is the measure which should be minimized during the training. This is done with a gradient descent based back-propagation algorithm which adapts all weights to better fit the input. When the back-propagation is finished, a new data vector is applied to the input and the procedure is repeated, but now with already better fitting weights. In this way, the algorithm converges to the filter weights with minimal loss.
There are three parameters which need to be specified in the convolution layer:

1. Kernel or window size: This determines how many samples are taken into account to calculate one filter output.
2. Stride length: This defines the number of skipped samples between two consecutive kernels.
3. Number of filters: This determines the number of different kernels which are trained in one window. This value can be considered the complexity of the layer.

The three parameters of the convolution layer are illustrated in Figure 2.9.

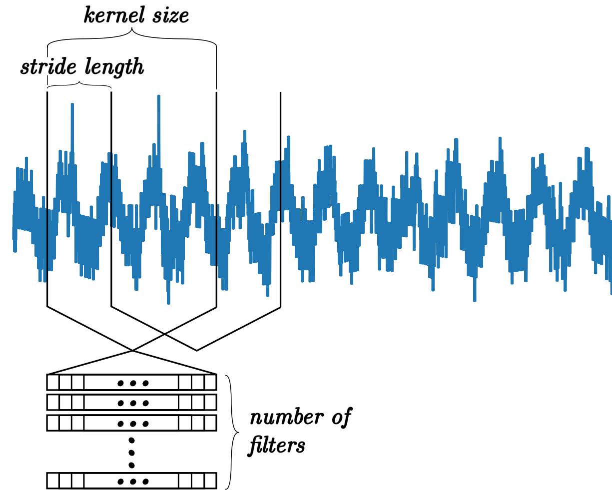


Figure 2.9: A one dimensional convolution layer showing all the parameters

2.4.3.1.2 Dense Layer

The dense or also called fully connected layer is the most basic layer in artificial NNs. It computes the output according to the scheme in Equation 2.23 and 2.24, where $\vec{x}$ is the input vector with dimension n, $w_{i,i}$ are the weights, $\vec{y}$ is the output vector with dimension m and $f(\vec{a})$ is the activation function. Similar to the convolution layer, the output is then compared to the true label and the loss is minimized by back-propagation.

$$\vec{a}^T = \begin{pmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \cdots & w_{m,n} \end{pmatrix} \vec{x} \tag{2.23}$$

$$\vec{y} = f(\vec{a}) \tag{2.24}$$

2.4.3.1.3 Long Short Term Memory

The LSTM was first proposed by Hochreiter and Schmidhuber in 1997 [44]. LSTM belong to the recurrent NN layers. Recurrent NN layers have, in contrast to standard NN layers, feedback connections. This creates a memory cell for each node, which allow the NNs to establish a temporal connection between the samples.
What distinguishes a LSTM layer from other recurrent layers is that the architecture enforces constant error flow, which solves the problem of vanishing or exploding back-propagation errors. With the ability of remembering previous samples, a LSTM is very well suited to detect temporal dependencies in a data stream. The only mandatory parameter which has to be specified is the dimension of the output space.

2.4.3.2 Implementation

The implementation of the NN is done in Python using the deep learning library Keras [23] with a Tensorflow [3] back end. We now explain the model of our NN and the two approaches used to identify videos.

2.4.3.2.1 Model

The first part of the NN consists of a one-dimensional convolution layer followed by a fully connected (dense) layer to extract the features from the thermal traces. The convolution layer takes as input a full trace of length $14\,000$ samples or 140 seconds. With a kernel size and a stride length of 1000 samples or 10 seconds each and an output dimension (number of filters) of 150, one trace gives 14 vectors with 150 elements. The output dimension is 150, since we empirically observed that for smaller dimensions, the network is not capable of learning the shape of the temperature trace. Higher dimension was not chosen to limit model complexity. The dense layer is then trained on labeling each vector with black or alternating video thermal traces. To make sure the convolution layer learns useful features, it is trained beforehand together with the dense layer on the traces shown in Figure 2.12 (a) and (b). Furthermore, to enforce normalized output vectors, a softmax activation function is applied to the output of the dense layer. After training this part of the network, the weights are fixed while training the second part.
The second part of the network can be realized in the following two ways:

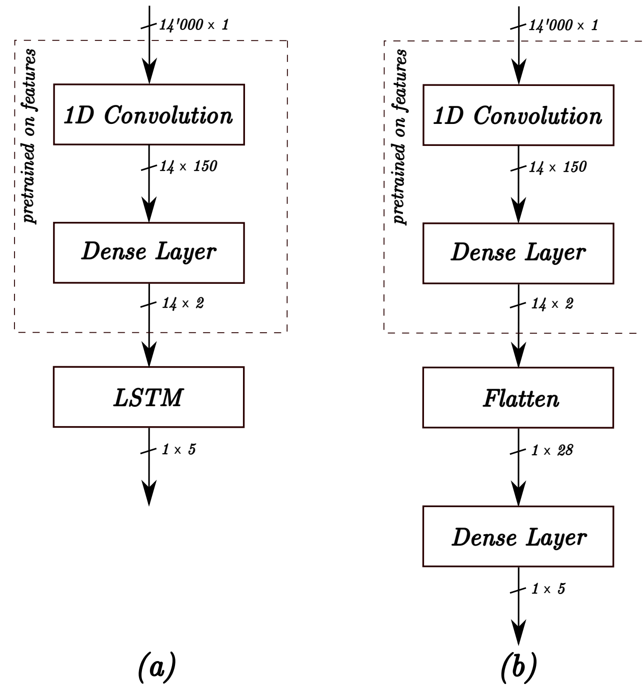


Figure 2.10: Two possibilities for the structure of the NN, where the dashed square shows which part is pretrained on the features. Afterwards the weights are fixed while training the LSTM (a) or the second dense layer (b).

1. LSTM based approach.
2. Flattening the vector space and using another dense layer.

Both approaches are trained on mixed traces of alternating and black video like the two examples shown in Figure 2.12 (c) and (d). The advantages of both approaches are described in the following two subsubsections.

2.4.3.2.1.1 LSTM Based Approach With a LSTM, the output of the pretrained part can directly be connected. The LSTM then computes the final labels. As the goal is to classify data with temporal dependencies, the LSTM is intuitively the right choice, but as the tests in subsection 2.4.4 show it has some serious disadvantages in our setting. The LSTM trains slower, as the internal structure is much more complex then the one of a dense layer. As we train it on short sequences of 14 features, the advantage of the memory cell does not take effect.

2.4.3.2.1.2 Dense Layer Approach For the dense layer approach the two-dimensional output generated by the first dense layer is converted into a one-dimensional vector using a flatten layer. This flattened vector is fed into another dense layer which computes then the labels of the mixed traces. The model is depicted in Figure 2.10 (a). One advantage of this approach is that it reaches a higher accuracy for the test set (difference of nearly 8 %) and converges faster than the LSTM.

2.4.3.3 Data Augmentation

For our initial tests we generate artificial traces from recorded thermal sequences to generate a large amount of data artificially. As stated in paragraph 2.4.3.2.1, the LSTM is trained on concatenated sequences consisting of 10 seconds snippets of the black and the alternating video. To generate these traces the algorithm randomly picks snippets from the original data set and places them in the right order. An illustration of this process is shown in Figure 2.11.

Finally, the convolution layer is trained on approximately 7000 traces of each video and the LSTM on 1000 traces each. Two sample traces for both training sets are shown in Figure 2.12. Both training sets have one-hot encoded labels, which means n labels are encoded in a vector with n rows and the only legal combination is one 1 entry and all the others 0.

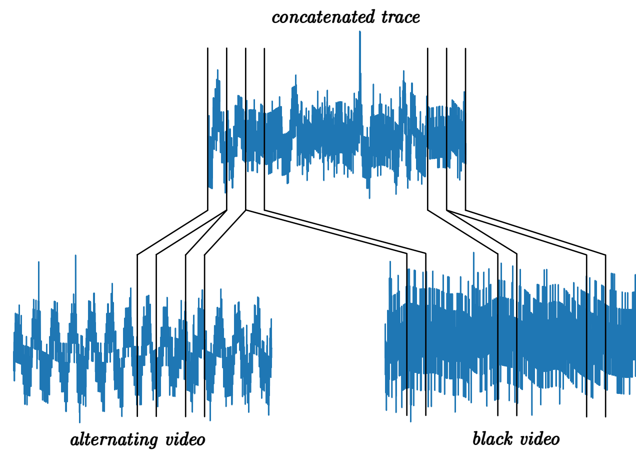


Figure 2.11: The procedure to generate the sequences for training the second part of the NN.

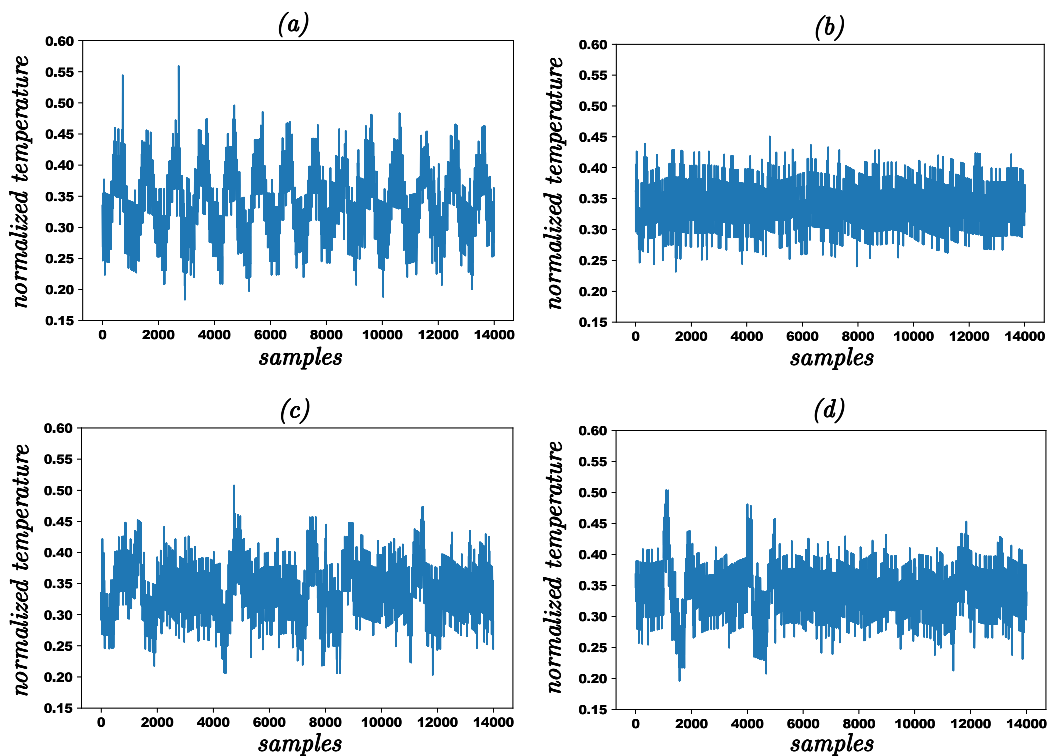


Figure 2.12: Traces to train the network. Sub-figures (a) and (b) are randomly shifted versions of the originally collected data. Sub-figures (c) and (d) are concatenated ten seconds snippets of the black and the alternating video in two different orderings, which serve as training data for the LSTM.

	# Labels	# Parameters	Relative Time to Train	Accuracy
LSTM	2	344	1.00	94.8%
	3	376	1.53	93.9%
	4	416	2.28	92.5%
	5	464	2.63	91.1%
Dense	2	362	0.23	99.6%
	3	391	0.34	98.9%
	4	420	0.47	98.8%
	5	449	0.63	98.7%

Table 2.3: The results of the two model types with two to five different sequences to classify. The time column is normalized on the time to train the two sequence LSTM.

	# Features	Model size	# Parameters
LSTM	2	632 kB	160
	5	633 kB	220
	20	644 kB	520
	100	700 kB	2120
	500	980 kB	10 120
	2000	2003 kB	40 120
Dense	2	633 kB	145
	5	636 kB	355
	20	648 kB	1405
	100	721 kB	7005
	500	1103 kB	55 005
	2000	2403 kB	140 005

Table 2.4: With increasing number of features computed by part 1, the number of parameters to train rises faster for the dense layer approach.

2.4.4 Results

Using the two models described in subsubsection 2.4.3.2, an accuracy of over 90% can be reached. With a rising number of different sequences the accuracy decreases, whereby the difference in accuracy for the compared approaches rises on nearly 8% for the more complex models. Also, the model complexity increases with the number of labels, which can be seen on the amount of trainable parameters and the needed time to train the models, again the LSTM has a steeper slope than the dense layer. As the LSTM needs more epochs to converge, the time to train it is drastically higher. To take into account that the time to train varies strongly for different machines and setups, it is normalized on the time of the two sequence LSTM, which took 321 seconds. The accuracy results are listed in Table 2.3. The second test focuses on how the complexity develops for more than two features. As expected, the number of trainable parameters rises faster for the dense layer approach which is caused by flattening the input matrix, but this does not seem to have a huge influence on the model size, as it is mostly 20% higher for the dense layer than for the LSTM. Due to the fact that there is not enough data to test a model on more than two features, it is not clear what effect this has on the accuracy of the two approaches. Table 2.4 shows all calculated model sizes.

2.4.5 Conclusion

The analysis presented in this section is preliminary; however, it is sufficient to make the determination that side-channel attacks (inferring platform activity based on temperature traces) is possible. Furthermore, it has been shown that basic video recognition based on temperature traces is possible using both the correlation based scheme presented in section 2.4.2 and the neural network based schemes presented in section 2.4.3. The extent/limit of these schemes has so far not been determined.

Chapter 3 Data Integrity

This chapter focuses on data integrity measures and their application in the domain of cyber-physical systems. It contains an introduction to the recently standardized KMAC algorithm and evaluation results for several data integrity algorithms. Furthermore, key management issues are described. Finally, design guidelines aimed at embedded developers and system designers are given. For an overview of current state-of-the-art techniques in data integrity, please refer to chapter 3 in D3.1.

3.1 Message Authentication Codes

Since the publication of D3.1, a new standard [50] for MACs based on SHA-3 (Keccak) has been released. In order to motivate the benefits of Keccak MAC (KMAC) compared to Keyed-Hash MAC (HMAC), we first present a more detailed view on HMAC (including an analysis of the design decision for using an outer and an inner loop) and continue to describe KMAC in this section.

3.1.1 Hash-based Message Authentication Codes (HMACs)

HMACs use a cryptographic hash function and a secret key to generate the checksum. Generally, the setup of an HMAC looks as follows [9]:

$$\mathtt{HMAC}(\mathtt{K},\mathtt{m}) = \mathtt{H}(\mathtt{K} \oplus opad \,||\, \mathtt{H}(\mathtt{K} \oplus ipad \,||\, \mathtt{m})), \tag{3.1}$$

where:

- $\mathtt{H}$ is a cryptographic hash function (e.g. SHA-256)
- $\mathtt{K}$ is a secret key
- $\mathtt{m}$ is the message
- $opad$ is the outer padding (i.e., the byte `0x5C` repeated until block length is reached)
- $ipad$ is the inner padding (i.e., the byte `0x36` repeated until block length is reached)
- $\oplus$ indicates an exclusive disjunction (XOR) operation
- $||$ indicates the concatenation of two bit strings

The HMAC construction from equation 3.1 uses two hash computations: firstly the *inner* loop using the secret key $\mathtt{K}$ with inner padding $ipad$ prepended to the message and secondly the *outer* loop using $\mathtt{K}$ with outer padding $opad$ prepended to the hash resulting from the inner loop. This construction protects against length-extension attacks, where the attacker can use $\mathtt{H}(m)$ to compute $\mathtt{H}(m')$ of a message m' that begins with the contents of m. All hash functions based on a Merkle-Damgård [62, 63, 25] construction (like MD5, SHA-1 and SHA-2) are prone to this attack.
The strength of the procedure depends on the strength of the underlying hash function. In [34] and [65], the applications of SHA- and MD5-based HMACs are further described.

3.1.2 Keccak Message Authentication Codes (KMACs)

The KMAC [50] algorithm is a keyed hash function based on the SHA-3 hash function (which is also known as Keccak). Keccak uses a very different construction compared to traditional hash functions, which are typically based on a Merkle-Damgård construction. In contrast, Keccak uses a sponge construction, in which data is "absorbed" into the sponge and then the result is "squeezed" out.

Keccak does not have the length-extension weakness and therefore does not need the HMAC construction from section 3.1.1. Recently, NIST standardized KMAC in [50], where two variants, KMAC128 and KMAC256, are specified as follows:

$$\mathtt{KMAC128}(\mathtt{K},\mathtt{X},\mathtt{L},\mathtt{S}) = \mathtt{cSHAKE128}(\mathtt{pad}(\mathtt{K},168) \mid\mid \mathtt{X} \mid\mid \mathtt{re}(\mathtt{L}),\mathtt{L},\text{"}\mathtt{KMAC}\text{"},\mathtt{S}) \text{ and} \quad (3.2)$$

$$\mathtt{KMAC256}(\mathtt{K},\mathtt{X},\mathtt{L},\mathtt{S}) = \mathtt{cSHAKE256}(\mathtt{pad}(\mathtt{K},136) \mid\mid \mathtt{X} \mid\mid \mathtt{re}(\mathtt{L}),\mathtt{L},\text{"}\mathtt{KMAC}\text{"},\mathtt{S}), \quad (3.3)$$

where

- `K` is a secret key of any length
- `X` is the message
- `L` is the requested output MAC length
- `S` is a optional customization string
- $||$ indicates the concatenation of two bit strings
- `cSHAKE()` is the hash function based on Keccak
- `pad()` is the byte-padding function
- `re()` is the right-encoding function

This construction allows to generate a MAC value with just one call of the underlying hash function. In contrast to other hash functions (including SHA-2, SHA-3), altering the requested output length generates a new, unrelated hash value. This effectively prevents length-extension attacks.
From an application point of view, KMAC is similar to HMAC, but more efficient. Furthermore, it supports variable-length output and hence can be used as a pseudo-random function.

3.2 Evaluation of Data Integrity Algorithms

Within SAFURE, we have implemented the following data integrity algorithms:

- AES-CCM (Advanced Encryption Standard in Counter with CBC-MAC mode)
- AES-GCM (Advanced Encryption Standard in Galois/Counter Mode)
- HMAC-SHA256 (Hash-based MAC based on SHA-2 with 256 bits)
- KMAC (Keccak CBC-MAC)
- Poly1305 (MAC based on polynomial $2^{130} - 5$)
- RSA digital signatures (Rivest-Shamir-Adleman cryptosystem)
- ECDSA (Elliptic Curve Digital Signature Algorithm)

We have evaluated these algorithms on the Infineon AURIX TC277T TriCore Microcontroller (see the "Algorithm" column in 3.1). The "ROM" column shows the code size, including machine code and static tables. The "RAM" column shows the memory consumption at run-time (e.g., stack usage). The "Time" column shows the performance with respect to speed. A lower value means a faster execution and thus a higher throughput of data. Finally, there is a brief description of the table.
As can be seen from Table 3.1, MAC algorithms are much faster than digital signatures. HMAC-SHA1 turns out to be the fastest MAC algorithm. However, if encryption is also desired, AES-GCM is the best choice. For the digital signature algorithms, RSA is the best choice if there are only a few signature generations and many signature verifications required. If the same amount of generation and verification is required, than ECDSA becomes the favourite choice. The memory requirements for all algorithms are quite moderate (less than 15kB ROM code).

Algorithm	ROM (bytes)	RAM (bytes)	Time (µs)	Description
AES-CCM	13697	28	4163	AES-128 in CCM mode (10 byte nonce, 16 byte add. data)
AES-GCM	13941	116	3107	AES-128 in GCM mode (12 byte IV, 20 byte add. data)
HMAC-SHA256	3402	208	1217	HMAC-SHA-256 with a 32 byte key
KMAC	2260	444	4513	KMAC-128
Poly1305	1937	272	4890	Poly1305-ChaCha20
RSASSA-PSS-Sign	11232	2256	1067560	PKCS#1-RSASSA-PSS signature generation, 1024 bit key, SHA-1
RSASSA-PSS-Ver	11232	1412	19779	PKCS#1-RSASSA-PSS signature verification, 1024 bit key, SHA-1
ECDSA-Sign	10346	1348	64123	ECDSA-secp192r1 signature generation
ECDSA-Ver	10346	944	89682	ECDSA-secp192r1 signature verification

Table 3.1: Evaluation results for Infineon AURIX TC277T TriCore Microcontroller

3.3 Key Management

This section gives an overview of different aspects of key management.

3.3.1 Key Generation

Cryptographic keys should not in any way be predictable for an attacker. Therefore, the key generation process must include some entropy from a non-deterministic source. Ideally, this could be a True Random Number Generator (TRNG) that uses a physical noise source to generate random numbers. Another option, if a TRNG is not always available or has only a limited bandwidth, would be to use a deterministic Pseudo-Random Number Generator (PRNG) that is seeded by a TRNG before use.
In many embedded scenarios, a TRNG is not available in the embedded device itself. For these, a dedicated Hardware Security Module (HSM) can be deployed. In this case, the key is generated in the HSM and then injected into the embedded device, usually at production time. Another option is to write an indivual seed (generated by a TRNG) into each device and let the device generate the key using a PRNG.
Some digital signature algorithms require a large amount of random data to generate the key (e.g., in the case of RSA, two large random prime numbers must be found, which requires several candidates to be checked for primality).
Furthermore, some algorithms (notably RSA with PSS padding) require random numbers during signature generation.

3.3.2 Key Distribution

In a network scenario with n entities, digital signatures have the advantage to only require each entity to publish its public key. The same scenario with MACs would require $n * (n-1)/2$ keys.
In scenarios that require only a limited amount of parties to communicate, MACs can be used. On embedded systems, the keys can either be injected at production time or distributed (and regularly updated) by a central gateway.
For multi-party scenarios, digital signatures almost always are preferred. In client-server computing, public keys are typically distributed using public key servers. These can also be used for embedded systems that have an online connection to the key server. However, many embedded systems do not have such a connection. For these systems, the public keys often are stored into the device at production time (either directly using hardware features or by using software functions). Renewal of the keys is then possible during maintenance.

3.3.3 Key Lengths

The optimal key length should be selected as a trade-off between security and performance, in the sense that it still has a sufficient security margin, but also does not sacrifice too much performance.
For symmetric algorithms like MACs, the typical choice for the key length is 128 bits. This is secure against brute-force attacks for several years.
For asymmetric algorithms like digital signatures, it is harder to classify the strength of a given key length. For traditional digital signature schemes like RSA and ElGamal, key lengths of at least 2048 bits are required and give a security margin similar to that of 128 bits for MACs. Algorithms based on elliptic curves, however, provide sufficient security with key lengths that is double to the respective symmetric algorithm. For example, ECDSA with a 256 bit key has roughly the same security margin than MACs with 128 bits.

3.3.4 Key Life-Cycle

The choice of key length should also consider the key's life-cycle (i.e., the time between generation and end of usage of a key).
Two time spans shall be considered in the evaluation:

- The life-cycle of the device generating the key.
- The usage time of each individual key.

These two time spans should be added in order to get the maximum life-cycle of the whole system. Then, the key length shall be chosen such that brute-force attacks are not possible within this time (assuming a growth in computing power of about two times every 18 months, according to Moore's Law).

In several applications, especially in embedded systems, the devices have a life-cycle of several years or even some decades. The usage time of individual keys depends on the application. Hence, the total life-cycle of a key could easily span two or three decades.
The choice of key management depends on several factors and influences the design of the Data Integrity solution. The next section on design guidelines summarizes the different design choices and tries to aid developers in their decisions.

3.4 Design Guidelines

In contrast to classical desktop computing systems (which are dominated by the Intel architecture), for cyber-physical systems there is a great variety of platforms, each having different properties:

- CPU architecture
- CPU frequency
- Number of cores
- Available memory (RAM and ROM)

Some platforms support cryptographic functions in hardware, e.g. as a cryptographic coprocessor or as an extension to the instruction set.
Similarly, there are many data integrity algorithms with different properties:

- CPU cycles
- RAM usage (stack and heap usage)
- ROM usage (typically FlashROM)

Unfortunately, there is not one solution that fits all cyber-physical systems. Further aspects that influence the decision about which algorithm to choose include: (i) use cases and misuse cases, (ii) potential threats, and (iii) attackers and attack trees.
It is recommended that a security analysis is performed in order to identify the threats with the largest risk. From these, the security objectives can be derived.

3.4.1 Guidelines

Here, we give some guidelines for the application of data integrity algorithms in the domain of cyber-physical systems.

1. **Symmetric vs. asymmetric:** Symmetric algorithms are much faster than their asymmetric counterparts. Therefore, it should be carefully evaluated whether it is necessary to use asymmetric cryptography. In cases where key distribution is a problem, asymmetric algorithms should be chosen. In most other cases, symmetric algorithms are the better choice.
2. **Space requirements:** If there is only little ROM space left on the device or if stack space (RAM) is very limited, symmetric algorithms usually better fit to these limitations. Also, it should be considered to use a space-efficient implementation of the respective algorithm.
3. **Speed:** Most speed optimizations come at the expense of a larger code size. When several algorithms are used in combination (e.g., hybrid encryption using AES with a key distributed using RSA), only the algorithms used for bulk data (AES in this case) should be optimized for speed.
4. **Key length:** The choice of key length influences the amount of (secure) storage and network traffic (for key distribution), but it also affects the run-time of algorithms. Therefore, the key length should be selected in order to offer good performance (speed, ROM and RAM usage) while still being secure enough for the application.

In the context of WP6, these design guidelines are applied to the SAFURE Use Cases. The results will be included in D6.4 and D6.6.

Chapter 4 Timing Integrity and Resource Sharing Integrity

This chapter focuses on **timing integrity** constraints associated with real-time applications, and the impact **of shared resources** on real-time constraints.

4.1 Timing Integrity for Multi-Cores: Challenge & Existing Solutions

In the Chapter 4 of Deliverable D3.1, we pointed out that safety-critical applications were characterized by stringent real-time constraints, and that time predictability was a major concern for the standards of the safety-critical industry.
In multi-core architectures, the major source for time variation comes from the **timing interferences** associated with concurrent accesses to shared hardware resources [12, 66]. Various Deterministic Platform Software (DPS) solutions [39] have been proposed to master these interferences and to ensure a deterministic usage of multi-core architectures: Control solutions aiming at eliminating all interferences and regulation solutions aiming at keeping the impact of interference below a harmful level.
The evaluation of these DPS solutions presented in Deliverable D3.1 versus different evaluation criteria has been performed in [39], and is summarized by Figure 4.1.

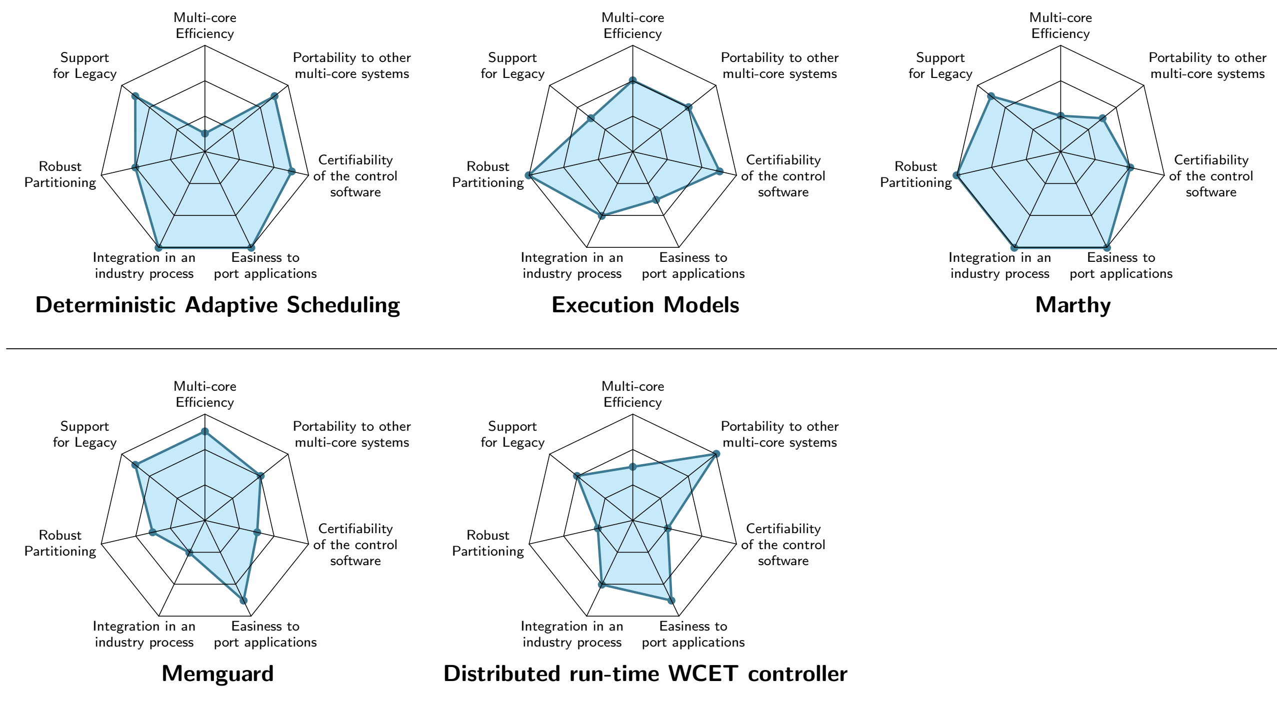


Figure 4.1: Evaluation of existing DPS solutions

The first row of Figure 4.1 represents the control-based solutions, while the second row presents the regulation-based solutions that are more adapted for mixed critical systems and quality-of-service support.
If all solutions are providing a different tradeoff between the different evaluation criteria, none of these solutions manages to fully exploit the multi-core efficiency (usually because of overprovisioning) nor to fully provide legacy support (running already certified / qualified application untouched).

4.2 Timing Integrity for Multi-Cores: SAFURE Solution

In the SAFURE project, we propose a new Deterministic Platform Software (DPS) solution relying on hardware resource budgeting, especially targeting mixed-critical systems. Both the architecture and the high-critical applications are characterized with regards to every hardware resources 1) to determine the total available resource budget; 2) to quantify the resource requirements of the high critical task; 3) to figure out the maximum level of extra external accesses to the resource before hampering the high critical tasks.
From this characterization information is inferred the maximum number of resource access allowed for non critical tasks, these tasks being suspended by a **Budget-Based RunTime Engine (BB-RTE)** once they reach this maximum number of total access during a given time slot.

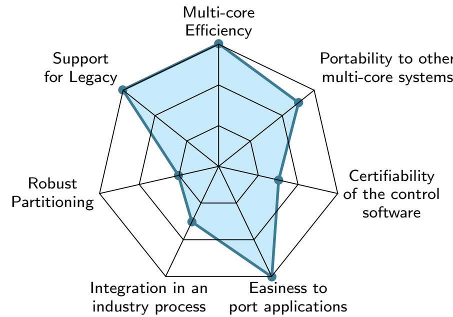


Figure 4.2: Budget-Based RunTime Engine (BB-RTE)

This technique particularly focuses on the ability to not modify neither the high-critical nor the low-critical applications, therefore not hampering the ability to run legacy applications, while exploiting most of the multi-core efficiency as depicted by Figure 4.2.

4.2.1 Principles

The principles of the approach consist in determining, per timeslot, a maximum budget to allocate to non critical applications in terms of resource accesses. When this budget is spent, non critical applications are suspended until the next timeslot, as depicted in first timeslot of Figure 4.3.

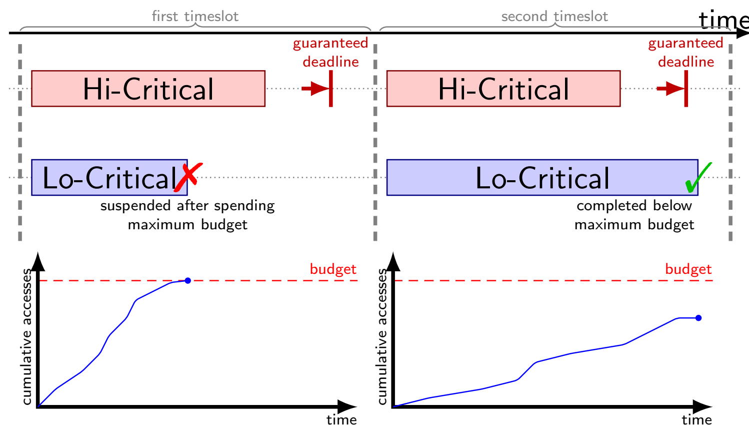


Figure 4.3: BB-RTE Principles

The budget is computed per timeslot and per hardware resource to ensure that the critical applications are matching their deadline in each particular timeslot.
During the second timeslot of Figure 4.3, the non-critical application manages to terminate without spending the whole allocated budget, again guaranteeing that the critical applications will match their deadlines during this timeslot.
With such a tactic based on budgeting, the major challenge consists in determining the budgets that guarantee the time behavior of high-critical applications. This way, time isolation principles and time integrity will be ensured for critical applications, while it will be relaxed for low critical tasks that could be suspended because of higher critical tasks.

4.2.2 Support & Tooling

To characterize the application behavior on the target hardware platform, to quantify timing interferences inherent to multi-core platforms, and to determine resource access budgets, we are relying on the METrICS measurement environment that aims at: 1) allowing an accurate estimation of embedded application performance; 2) characterizing the application behavior on the target hardware platform; 3) quantifying timing interference inherent to mutli-core platforms. This measurement environment is presented in details in Deliverable D3.3, and depicted by Figure 4.4.

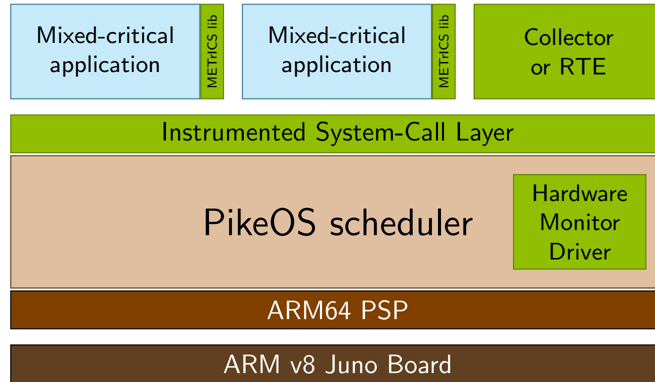


Figure 4.4: Measure Environment for TIme-Critical Systems (METrICS)

During the characterization phases, allowing us to characterize the critical application and to compute the budgets, we run the toolsuite together with the collector partition. During the runtime phases, we run the toolsuite together with the BB-RTE partition that is issuing scheduling decisions based on budget consumption.

4.2.3 Process to determine and ensure budgeting

The process to determine resource budgeting is performed with two major steps: First, an offline characterization step, and second a runtime regulation step relying on a runtime engine dedicated for mixed critical systems, as depicted in Figure 4.5.

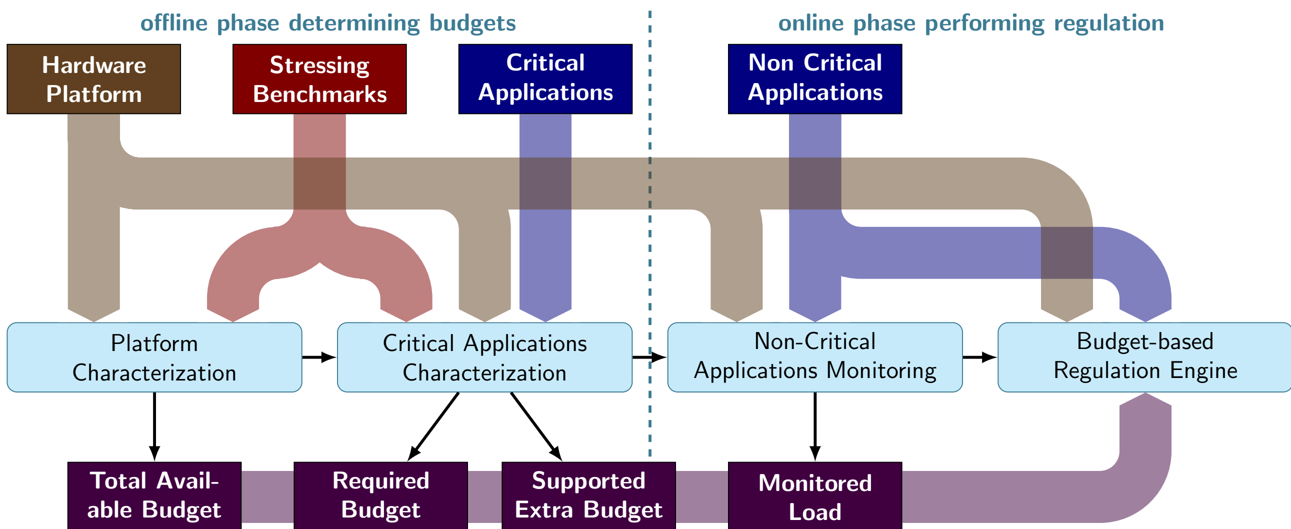


Figure 4.5: Timing integrity process for mixed time-critical systems

Characterization steps involve large-scale experimentation due to the limited monitoring resources of multi-core architectures. These architectures usually propose from tens to hundreds of hardware events that you can monitor, but only allow to monitor a few Performance Monitor Counters (PMC) at a given time (6 for ARM-v8 architectures for instance). As a consequence, testing all the countable PMC events involves days of experimentations, and this characterization phases have to be performed off-line. Anyhow this characterization only

needs to be performed once per critical application for a particular hardware target, and such a characterization could also provide useful information for the qualification or certification documents.
The second step is performed during the execution and consists in having a regulation system using the set of budgets extracted during the characterization phases. This regulation system is realized with the BB-RTE runtime engine described below.

4.2.4 Hardware characterization & budgeting

As introduced in Figure 4.5, in the first sub-step of the offline characterization phase, we perform a characterization of the target hardware platform. This hardware characterization consists in defining a set of low level (assembly code) Stressing Benchmarks (SB). Each of these stressing benchmarks is responsible for stressing a particular hardware resource of the selected multi-core target, by multiplying the number of accesses to this particular resource.
By progressively stressing each resource while monitoring both the execution time and the effective number of accesses to this resource thanks to PMC counters, we are able to determine the maximum available bandwidth in terms of accesses to this resource, and that corresponds to the **total available budget** for the resource.
By iterating over all the potentially shared hardware resources, we obtain a vector of such total budgets that fully characterize the hardware limitations of the selected platform.

4.2.5 Critical application characterization & budgeting

During the second characterization sub-step of Figure 4.5, we are characterizing the usage made by high critical applications of the shared hardware resources. To do so, we run the critical applications concurrently with the stressing benchmarks described above, progressively increasing the stressing level, and only monitoring the effect in terms of runtime on the critical application.
It allows us to extract two different kinds of information: First the **required per-resource budget** needed by our critical applications; and second, the level of extra resource access supported by our critical application before being significantly slowed down. This **supported extra accesses** is the access budget that can be safely used by the non-critical applications.
The process to determine the acceptable level of slowdown and the associated extra access budget is depicted in Figure 4.6. The y-axis represents the maximum observed runtime of the monitored application during the current timeslot. The x-axis represents the extra access load performed on the associated hardware resource by the stressing benchmarks.

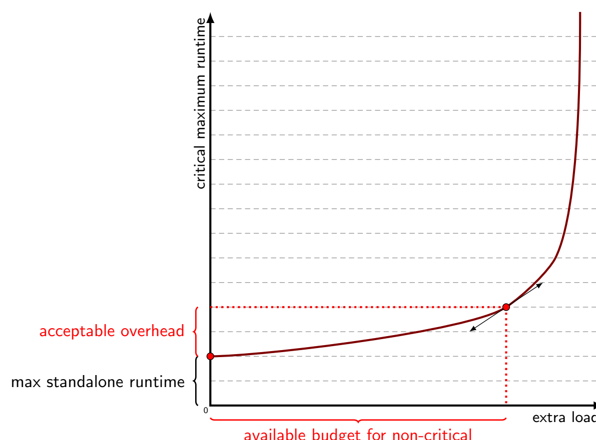


Figure 4.6: Determining an acceptable level of slowdown and the associated extra access budget

The leftmost point in the chart corresponds to when the application is running alone in isolation (with a null stressing benchmark activity). It therefore corresponds to the classical WCET of application running in isolation.
The rightmost point in the chart corresponds to a permanent maximum loads from the stressing benchmark actually nearly preventing the monitored application to access the required resource (the application being only protected by anti-starvation features of the hardware arbiter).
Selecting an acceptable level of slowdown and the associated extra access budget is performed by selecting a point on the chart. The projection of this point on the y-axis then corresponds to the acceptable level of

slowdown and the projection on the x-axis directly provides the extra access budget available for non-critical applications with an acceptable impact on critical applications.
We used two different techniques to select this point: First by directly selecting a maximum slowdown, but doing so for every hardware resource led us to too much overprovisioning, failing to fully exploit the multi-core efficiency. Second, we defined a maximum slope for the curve. This second solution allowed us to vary the level of slowdown relatively to the shape of the curve, focusing on the hardware resources the critical application was the most sensitive to.
By repeating this procedure for each of the shared hardware resources, we obtain again a budget in the form of a vector of both the required amount of access by the critical application (the leftmost point of the curve) and the number of supported extra access (the selected point of the curve).

4.2.6 The SAFURE Budget-Based RunTime Engine (BB-RTE)

Once the characterization phases are over and all budgets have been gathered, both high critical and low critical applications can be deployed on the hardware target together with the runtime engine. As shown in Figure 4.5, the online regulation phase consists itself into two substeps.
First, using the same process as per the characterization phases, the non critical applications are monitored with PMC counters, this time not to compute a budget, but to monitor the load in terms of hardware resource accesses.
Second, the BB-RTE runtime engine compares this load with the maximum extra budgets to decide if, during the current timeslot, non-critical tasks may continue executing or if they need to be suspended until the next timeslot. Doing so makes sure the slowdown of critical tasks does not hamper their ability to match their deadlines.
One of the specific challenges of the BB-RTE runtime engine for time-critical systems is that time intrusivity of the associated monitoring features has to be kept minimal, not to bias the time characterization results. Also the BB-RTE itself should make a minimal usage of shared hardware resources to not impact the resource access budgets. The final intrusivity footprint of both the monitoring features and the BB-RTE engine will be presented in Deliverable D4.2 (Analysis of runtime and software applications on multicore).

4.2.7 Results

As the characterization phase is performed offline, it does not have to focus on particular hardware resources nor any particular Performance Monitor Counters (PMC) as the number of experimentation can be freely expanded. During the regulation phase however, we cannot replay the same function several times to test all the possible PMC and their associated hardware components. As each core of the multi-core architecture is only allowing us to measure 6 different PMC at a given time plus the cycle timebase, one of the goals of the characterization phase is also to reduce the design space to identify the most meaningful counters.
Of course, the PMC corresponding to non-shared hardware resources (particularly all the one related to the core pipeline) could be eliminated, as we focus on those describing the behavior of shared hardware resources that are responsible from timing interference.
On the ARMv8 based Juno board, we empirically identified that, for our applications, the most meaningful counters were the ones related to the cache hierarchy (due to the shared L2 cache), and the ones related to the NoCs connecting the caches to the DDR memory.
More details on the characterization results and the BB-RTE measurements will be presented in Deliverable D4.2 with the mixed-critical applications composing the WP4 prototype. The specification of this prototype has been provided in Chapter 5 of Deliverable D4.1.
Also, the process described in this Section relies on budgeting hardware resource accesses. It can be easily extended to energy consumption or temperature as long as the hardware target provides the adhoc probes to be used in lieu of PMC counters. This way power or temperature can similarly be kept below a predefined threshold by suspending some non critical applications during over-spending time slots.

4.3 Timing Integrity in Overload Conditions

Many automotive systems, such as those in Safure, do not comply with the typical assumptions of hard real-time systems, that is where a single deadline miss has catastrophic consequences. However, in these systems, timing integrity should be still verified by guaranteeing an upper bound on the number of deadlines that can be missed.

Several control systems (including for example fule injection in automotive) are tolerant to individual deadline misses and treating them as hard real-time would result in unnecessary pessimism and possibly overprovisioning of resources. Of course, uncontrolled response times are also not desirable and even in case of deadline misses the designer may require some guarantees on the timing behavior of the system.

Since the seminal work of Liu and Layland [55], an overwhelming effort in real-time scheduling research has been dedicated to the question on whether there can be *a possible deadline miss in the system* according to the *hard* real-time analysis model. It is hard to completely identify the reasons for this disproportionate interest in hard analysis techniques. It is probably because of the simplicity of the model, its easier understandability and analyzability, the seemingly natural fit to safety-critical systems and, quite possibly, some incorrect judgement on the part of some researchers that believe most real-world systems are of hard type. The success of hard schedulability analysis also benefits from the existence of the critical instant, an activation scenario where all tasks are simultaneously activated, that leads to the worst-case response time for every task in the system. As a result, in a hard real-time system it is sufficient to investigate the particular task activation pattern that originates from the critical instant. More details on classic hard real-time schedulability analysis can be found in textbooks and surveys like [19] and [77].

While it is true that some safety-critical systems are vulnerable to a single violation of the temporal constraints, there are many more that can tolerate timing violations. In these cases, the hard schedulability analysis is too strict for a system's safety analysis. The *weakly* hard real-time schedulability analysis targets the problem of *bounding the maximum number of deadline misses over a number of task activations.* A dynamic priority assignment of priority for streams with m-k requirements was proposed originally by Hamdaoui et al. [58] to reduce the probability of m-K violations in time-sensitive applications. Hard real-time schedulability analysis can be traced back to the work of Bernat et al. [10] on the m-K model, in which no more than m deadline misses shall occur for any K consecutive activations of a task. The analysis in [10] and in other works assumes that there is an explicit initial state of the system, in which the initial offset of each task in the system is known. By restricting the analysis to a periodic task activation pattern, the weakly hard analysis can be conducted by checking task activations and interleavings within a large enough time span from the system initialization, so as to verify the m-K assumption. Periodic tasksets are quite common in real world applications, but the requirement of knowing all activation offsets may be too strict and undermine the robustness of the analysis: given a periodic task system with explicit initial offsets that passes the (weakly) hard test, a slightly change of the initial offset of some task may result in an unexpected time failure. The analysis is also very sensitive to a drift of the task periods. Some other techniques (in the next paragraph) and the work in this paper are more robust.

Another more recent direction on the research of weakly hard real-time analysis relaxes the requirement of knowing the initial state of the system. The approach consists in the application of *typical worst-case analysis* [69] to a system model represented as the superposition of a typical behavior (e.g., of periodic task activations) that is assumed feasible, and a sporadic overload (i.e., rare events). Under such an assumption, [42] and [91] proposed methods for weakly hard analysis that is composed by two phases: 1) the system is verified to be schedulable under the typical scenario (this can be done by the classical hard analysis), and 2) when the system is overloaded, it can be guaranteed that out of K successive activations of a task, at most m of them will miss the deadline. The sporadic behavior can be abstracted by observing and analyzing the system in runtime, and is regarded as rare events. A similar approach is considered in [52], where real-time calculus is used to analyze the worst-case busy period (in duration and lateness) that results from a temporary overload because of an error in the timing assumptions on the task worst-case execution times. Both these methods require the definition of a task model that may be in several cases artificial, since it requires the identification and separation of possible causes of overload. This line of works and the method in this paper are orthogonal and the two extend the weakly hard real-time system analysis at different directions. Combined use of the two (in the future) can make the safety analysis and system design more comprehensive.

Among the possible options, the *weakly hard* scheduling model has been proposed by several authors (Hamdaoui et al. [58] and Bernat et al. [10]) to check for a relaxed condition, that is, to analyze the number of temporal constraints violations given a time window or a sequence of task activations. This is also called m-K model, since a typical formulation consists in checking that no more than m deadlines are missed over a set of K activations. The rationale behind this model is that each deadline miss will bring the system closer to a faulty condition or state, and each completion in time will move the system back towards a safe working state.

The weakly hard model can be analyzed under several scheduling options, but due to its simplicity and effectiveness, fixed-priority scheduling is nowadays the de-facto standard for industrial real-time systems. Our analysis focuses on real-time systems consisting of periodic tasks scheduled with fixed priority.

The main problem with the existing analysis methods is that for a weakly hard system, as for any system that can be (at least temporarily) overloaded, the critical instant theorem does not hold and the system becomes

much more difficult to analyze. In the original work, this limitation has been overcome by restricting the study to offset-determined systems, that is, systems in which tasks are scheduled with a known initial offset.
This is a possibly serious limitation, not only because the designer may not be able to enforce or determine the system initial activation offsets, but especially because the analysis of all offset-determined systems is very sensitive to time drifts and errors in the task activation times, which are extremely hard to avoid in real systems.
As a result of the work in Safure, we provide a generalized framework for offset-free systems scheduled on uniprocessors. Our formulation is based on a MILP encoding and the problem of the critical instant determination is addressed by letting a variable represent the beginning of the busy period. The MILP formulation can be easily reused to check the system for a large set of m and K values, allowing the designer to explore a wide design range. The developed MILP model serves as an over-approximate analysis, that is, if our weakly hard analysis confirms the m-K property, it is guaranteed that there will be no more than m deadline misses out of any K successive job activations (of the target task), however, if the m-K property is not confirmed, we cannot conclude the opposite.
The result is the first weakly hard schedulability analysis for offset-free periodic real-time tasks. The analysis method includes the consideration of resource sharing and the task jitter. To solve the possible issues with the large number of integer variables counting the number of task interferences (as used, for example, in [13, 88, 51]), we relaxed these variables to real values, but we added binary variables expressing the possibility of job interferences for reducing or possibly eliminating the introduced pessimism. Surprisingly, there is no existing work that can cope with the weakly hard analysis of general (offset-free) periodic tasks, and this prevents a fair comparison between our solution and other relevant works. Thus, we evaluate our analysis method through extensive experiments to show its efficiency (expected runtime) and precision. In the special case in which $m = 1$, the analysis is always accurate as long as it validates the m-K property. With respect to accuracy, despite the relaxation to real valued job interference counters, the MILP analysis can still return exact results for a very high percentage of the tests.
The technical details of the work will be presented at the EMSOFT Conference in Seoul.

4.3.1 The System Model

A periodic real-time task is characterized by a tuple $\tau_i = (C_i, D_i, T_i)$ with $C_i \leq D_i \leq T_i$ such that C_i is its Worst-Case Execution Time (WCET), D_i is the relative deadline and T_i is the period for the activation of τ_i. A task's utilization is defined as $U_i = \frac{C_i}{T_i}$.
Each activation (instance) of τ_i is denoted by a job $J_{i,k}$ with $k = 1, 2, \ldots$ representing the job index (of its activations in time). A job $J_{i,k}$ can be further represented by its activation (or arrival) time $a_{i,k}$, its absolute deadline $d_{i,k} = a_{i,k} + D_i$, and its finish time $f_{i,k}$.
A job is schedulable if it can finish its execution before its deadline, i.e., $f_{i,k} \leq d_{i,k}$; and a task τ_i is schedulable if all its jobs are schedulable. The elapsed time between a job finish time and its activation time (i.e., $f_{i,k} - a_{i,k}$) is its *response time*. By definition, a task τ_i is schedulable if and only if the Worst-Case Response Time (WCRT) $R_i = \max_k \{f_{i,k} - a_{i,k}\}$ among all its jobs is not larger than its relative deadline D_i. A task Best-Case Response Time (BCRT) $r_i = \min_k \{f_{i,k} - a_{i,k}\}$ is the minimum time that it takes to complete the execution of the task.
In the periodic activation pattern $a_{k+1} - a_k = T_i$ for any two successive jobs of a task. As we do not require a specific initial offset for a task, the first job activation time $a_{i,1}$ of τ_i is unknown. A job (and the corresponding task) is said to be *active* if it has been activated but has not completed its execution.
The periodic taskset $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ executes upon a uniprocessor platform. Each task in $\mathcal{T}$ is assigned a unique and static priority, and tasks are scheduled by a *fixed priority preemptive scheduler*. Tasks are ordered in $\mathcal{T}$ from higher to lower priority. That is, τ_j has a higher priority than τ_i, if $j < i$. If a task does not always finish before its deadline, it can have multiple active jobs at the same time. In such cases, these jobs are served in FIFO order, meaning that the latter job cannot execute until all its predecessors are completed.
A *level-i busy period* is defined as a time interval during which the processor is always occupied by the execution of tasks with priority higher than or equal to τ_i. For example, in Figure 4.7, $[s_0, f_2)$ and $[a_3, f_3)$ are two level-3 busy periods: Because the focus of this paper is not the single task WCRT, our definition of a busy period does not strictly follow its original meaning in [53], where a busy period corresponds to the maximal level-i busy period in this paper.
The execution of any job of τ_i can only be affected by the workload of interfering tasks (including τ_i itself) within the same level-i busy period. According to [53], the WCRT R_i of a task τ_i is found within the longest level-i busy period, which starts at the critical instant (i.e., when all tasks are activated at the same time). In case a task always completes before its next activation, the task schedulability can be easily checked by computing the response time of the first task instance inside it.

However, this condition does not hold for weakly hard real-time systems when deadlines can be missed and multiple instances can be active at the same time. In this case, the WCRT of a task τ_i does not necessarily happen for the first job in a level-i busy period. However, the BCRT is still occurring for the last job in a level-i busy period. Algorithms to compute the BCRT can be found in [72, 18]. In this work, we trivially assume that the BCRT of a task does not exceed its period: $r_i \leq T_i$. Otherwise, the task simply misses all its deadlines. Also, we assume that the BCRT r_i and the WCRT R_i of each task are computed in advance using established techniques such as in [53]. Once computed, these values can be used as parameters in the MILP formulation. The analysis of weakly hard systems is performed by analyzing each individual (arbitrary) task $\tau_i \in \mathcal{T}$, also defined as *target task*.
An arbitrary sequence of K successive activations of τ_i is considered, with the objective of checking whether there are more than m deadline misses for τ_i in this sequence.
For simplicity, the jobs of τ_i in the activation sequence are denoted by $J_1, \ldots, J_k, \ldots, J_K$ (without the task index). Given a job J_k, its activation time and finish time are defined as a_k and f_k, respectively. The time interval $[a_k, a_{k+1}[$ is called the *kth job window* of τ_i, and the *problem window* for the analysis is $[s_0, f_K[$, where s_0 (also considered as the time reference $s_0 = 0$) is the earliest time instant such that the processor is fully occupied by the execution of higher priority tasks from s_0 to a_1.
As an example, consider a system of 3 tasks: $(C_1 = 1, D_1 = 3, T_1 = 3)$, $(C_2 = 3, D_2 = 5, T_2 = 15)$ and $(C_3 = 2, D_3 = 6, T_3 = 6)$, with $K = 3$ and τ_3 is the target task. Figure 4.7 shows a scenario where 2 (J_1 and J_3) out of 3 jobs in the problem window $[s_0, f_3[$ miss the deadline. In this case, $s_0 = 0$ and $a_1 = 0.5$. If the problem window starts at the critical instant, that is, when all tasks are synchronously activated in s_0, only J_1 misses its deadline.

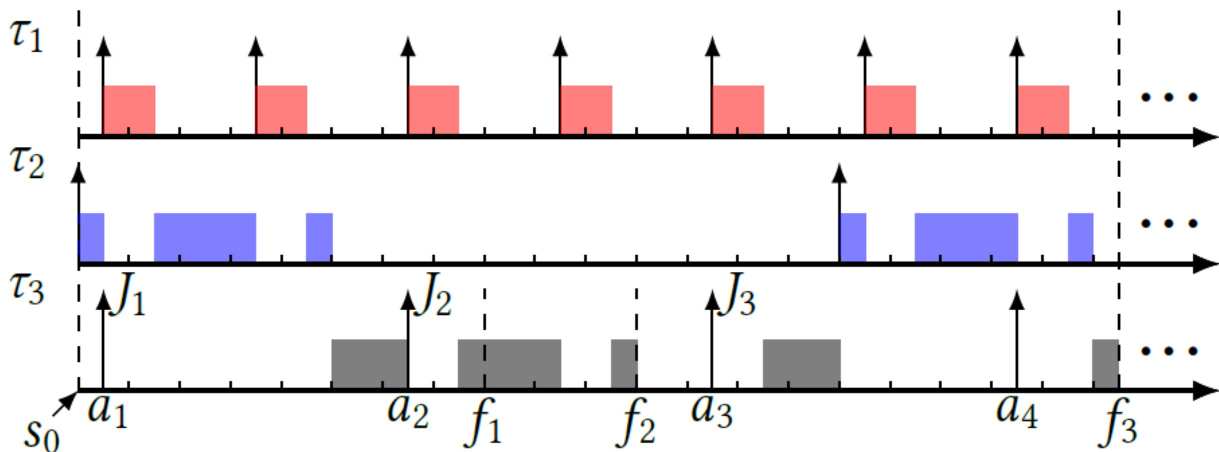


Figure 4.7: A problem window with 3 job windows

4.3.2 The Solution Model

We provide a Mixed Integer Linear Programming (MILP) formulation for the weakly hard analysis of a set of offset-free periodic tasks under fixed priority scheduling. Two observations allow to reduce the problem space by considering only the problem windows that maximize the number of deadline misses for τ_i.
(**O1**) The worst-case number of deadline misses occurs for problem windows such that the last job of τ_i before the beginning of the problem window (indicated as J_0) is schedulable.
If J_0 is not schedulable, any problem window of k instances starting with J_0 has at least as many misses as the window starting with the first job J_1. The two windows have all the jobs from J_1 to J_{k-1} in common, but J_0 misses its deadline, therefore, in the best case the two windows have the same number of misses if also J_k is a deadline miss. In Figure 4.7, consider the problem window with J_2, J_3 and the following instance (not shown) J_4. Depending on the schedulability of J_4, there will be 1 or 2 deadline misses in this window. However, since J_1 is non-schedulable, there are 2 deadline misses for the problem window including J_1, J_2 and J_3.
(**O2**) The worst-case number of deadline misses occurs for problem windows such that the the first job is non-schedulable.
Consider a window of K instances that starts with a set of schedulable jobs of arbitrary length $J_1, \ldots J_n$ (with $n < K$; if $n = K$ the proof is trivial) and the window that starts with J_{n+1}; the latter has at least as many deadlines misses as the window starting with J_1 (the proof is similar to the previous case).

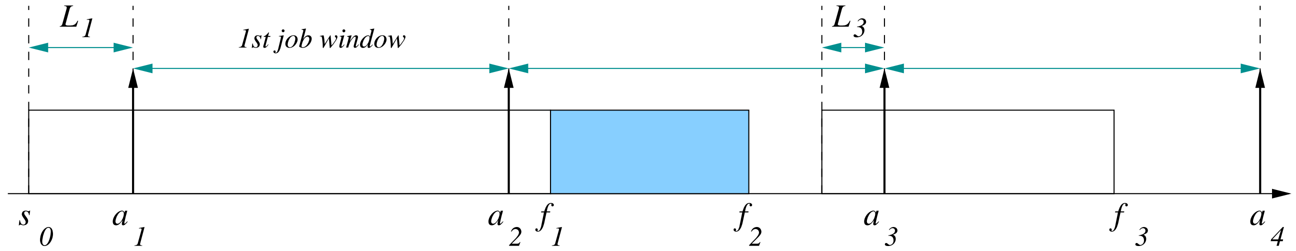


Figure 4.8: Notation for the definition of a problem window

4.3.2.1 Variables and basic constraints

In this subsection, we introduce the (real and Boolean) variables defined in our MILP model, together with some basic constraints on them. Real valued variables are labeled by $\mathbb{R}$ and Boolean variables by $\mathbb{B}$. M is a big enough constant value used to encode conditional constraints (a standard technique known as big-M). A brief summary of all the optimization variables is in Table 4.1.

Type	Variables	Annotations
$\mathbb{R}$	L_k	Segment of busy execution of higher priority tasks in front of (before) a job window, when $f_{k-1} \leq a_k$.
	α_j	Activation time of the 1st job of each higher priority task τ_i with respect to the start time s_0 of the analysis.
	f_k	Finish time of J_k.
	ι_k	Processor idle time inside the job window.
	$If_{j,k}$	#jobs of τ_j within the time interval $[0, f_k[$.
	$IL_{j,k}$	#job of τ_j inside $[0, a_k - L_k[$.
$\mathbb{B}$	b_k	$b_k = 0$ if J_k is schedulable; $b_k = 1$ if J_k misses its deadline. The number of deadline misses is $\sum_k b_k$.
	β_k	$\beta_k = 0$ if J_k completes before a_{k+1}; otherwise $\beta_k = 1$ and J_k interferes with J_{k+1}.
	$\Gamma f_{j,k,p}$	$\sum_p \Gamma f_{j,k,p}$ is #jobs of τ_j inside: 1) $[a_k - L_k, f_k[$ when $k = 1$ or $\beta_{k-1} = 0$; 2) $[f_{k-1}, f_k[$ when $k > 1$ and $\beta_{k-1} = 1$.
	$\Gamma L_{j,k,p'}$	$\sum_{p'} \Gamma L_{j,k,p'}$ is #jobs of τ_j inside $[f_{k-1}, a_k - L_k[$, if it exists.

Table 4.1: A summary of variables defined

4.3.2.2 Busy periods

Each job of τ_i inside the problem window can be interfered by pending executions of higher priority tasks that are requested before its activation but have not completed. The real valued L_k (as in Figure 4.8) indicates the

portion of the level-i busy period for job J_k that extends to the earliest such activation, when $f_{k-1} \leq a_k$. That is, if J_{k-1} finishes execution not later than a_k (that is, J_{k-1} does not interfere with the execution of J_k), $a_k - L_k$ is the earliest time instant such that the processor in $[a_k - L_k, a_k)$ is fully occupied by higher priority tasks.
The start time $s_0 = 0$ of our problem window is $a_1 - L_1$, and the arrival time a_k of the kth job of τ_i in the problem window is $a_k = L_1 + (k-1) \cdot T_i$. A trivial constraint that applies to all L_k is

$$\forall k \;\; 0 \leq L_k \leq T_i - r_i \tag{4.1}$$

Throughout our analysis, we only use the value of L_k, when $f_{k-1} \leq a_k$. If L_k is larger than $T_i - r_i$ then $f_{k-1} > a_k$ (any job of τ_i needs at least r_i to complete). In Figure 4.7, $L_1 = 0.5$ and $L_3 = 0$. Because J_1 interferes with the execution of J_2, L_2 is not relevant to our analysis.

4.3.2.3 Offsets

The offset of a higher priority task within the problem window refers to its first job activation time with respect to the start time s_0. The first job activation that happens no earlier than s_0 for each higher priority task τ_j is denoted by $\alpha_j \in \mathbb{R}$

$$\forall j < i \;\; 0 \leq \alpha_j \leq T_j - r_j \tag{4.2}$$

Assume that the first job $J_{j,1}$ of τ_j in the window arrives at time $s_0 + T_j - r_j + \epsilon$ with $\epsilon > 0$. This implies that the previous job $J_{j,0}$ is activated at time $s_0 - r_j + \epsilon$. Because any job of τ_j needs at least r_j time to finish, $J_{j,0}$ will be still active at time instant s_0, which contradicts the hypothesis that s_0 is the earliest time instant such that from s_0 to a_1 the processor is fully occupied by higher priority tasks. Hence, the upper bound for α_j is $T_j - r_j$.

4.3.2.4 Finish times

For each job J_k of τ_i, its finish time is denoted by $f_k \in \mathbb{R}$

$$\forall k \;\; r_i \leq f_k - a_k \leq R_i$$

Because jobs from the same task are executed sequentially, for any two consecutive jobs of τ_i, this precedence constraint is encoded as

$$\forall k \;\; C_i \leq f_{k+1} - f_k \leq R_i + (T_i - r_i) \tag{4.3}$$

4.3.2.5 Level-i idle time inside a job window

The level-i processor idle time refers to the time when the processor is not occupied for execution by τ_i or any other higher priority task (in a given time interval). Given an arbitrary job window $[a_k, a_{k+1}[$ of J_k, we define $\iota_k \in \mathbb{R}$ as the amount of processor idle time inside this kth job window

$$\forall k \;\; 0 \leq \iota_k \leq a_{k+1} - a_k - r_i$$

4.3.2.6 Schedulability of each job of τ_i

For each job J_k of τ_i inside the problem window, a Boolean variable $b_k \in \mathbb{B}$ indicates whether the job misses its deadline:

- $b_k = 0$ if J_k finishes its execution no later than its deadline;
- $b_k = 1$ otherwise.

The value of b_k is defined by the comparison between the finish time f_k of J_k and its absolute deadline $a_k + D_i$: $b_k = 0 \Leftrightarrow f_k \leq a_k + D_i$, which is encoded by the following linear constraint.

$$\forall k \;\; -M \cdot b_k \leq a_k + D_i - f_k < M \cdot (1 - b_k) \tag{4.4}$$

Being M a very large value, the conditional constraint in (4.4) forces $b_k = 0$ if the job J_k meets its deadline (i.e., $f_k \leq a_k + D_i$) and $b_k = 1$ otherwise. As in observation **O2** at the beginning of Section 4.3.2, we require that J_1 misses its deadline, that is, $b_1 = 1$ (schedulable tasks can be ruled out by simply performing a traditional hard schedulability test in advance).
The *total number of deadline misses* of τ_i inside the problem window is denoted by $\sum_k b_k$.

4.3.2.7 Interference from the previous jobs of the same task

A job J_k of τ_i interferes with the execution of the next job J_{k+1} in case $f_k > a_{k+1}$. The Boolean variable β_k encodes this condition.

- $\beta_k = 0$ if J_k finishes its execution within its own job window;
- $\beta_k = 1$ if J_k completes after a_{k+1}.

Similarly as in (4.4), the constraint $\beta_k = 0 \Leftrightarrow f_k \leq a_{k+1}$ over β_k, f_k and a_{k+1} can be formulated as

$$\forall k \quad -M \cdot \beta_k \leq a_{k+1} - f_k < M \cdot (1 - \beta_k) \tag{4.5}$$

If there is idle processor time inside the job window $[a_k, a_{k+1}[$ of J_k, then J_k must terminate within its window and does not interfere with J_{k+1} (i.e., $\beta_k = 1 \Rightarrow \iota_k = 0$).

$$\forall k \ \ \iota_k \leq M \cdot (1 - \beta_k)$$

4.3.2.8 Number of interfering jobs from higher priority tasks

When modeling a schedulability problem in MILP, the major complexity comes from computing the interference from higher priority tasks. A common approach (as in [13, 88, 51]) is to count the number of jobs from each higher priority task that interfere with the execution of the task under analysis. Different from previous works, we explore the relaxation of this integer count to a real value. Table 4.2 summarizes the variables defined for counting the higher priority interferences for the example system in Figure 4.7

Given a job J_k of τ_i and a higher priority task τ_j, $If_{j,k}$ is the number of job instances of τ_j within the time interval $[0, f_k[$. By definition, $If_{j,k} = \lceil \frac{f_k - \alpha_j}{T_j} \rceil$ is an integer number. However, we relax the definition of $If_{j,k}$ allowing it to be a real value and we linearize the constraint on $If_{j,k}$ as (by the definition in [53])

$$\forall j < i \ \ 0 \leq If_{j,k} - \frac{f_k - \alpha_j}{T_j} < 1 \tag{4.6}$$

Moreover, we define $IL_{j,k} \in \mathbb{R}$ as the number of jobs of τ_j ($\forall j < i$) within the time interval $[0, a_k - L_k[$, when $\beta_{k-1} = 0$. In this case, If J_k is not interfered by its predecessor J_{k-1}, then the number of jobs from τ_j that interfere with the execution of J_k is $If_{j,k} - IL_{j,k}$. We remind that we only use the value of L_k, and thus the interval $[0, a_k - L_k[$, when $\beta_{k-1} = 0$.

Formally, if J_{k-1} completes before the activation of J_k (i.e., $\beta_{k-1} = 0$), then $IL_{j,k} = \lceil \frac{a_k - L_k - \alpha_j}{T_j} \rceil$. That is, $\beta_{k-1} = 0 \Rightarrow 0 \leq IL_{j,k} - \frac{a_k - L_k - \alpha_j}{T_j} < 1$. In other words, $\forall j < i$

$$-M \cdot \beta_{k-1} \leq IL_{j,k} - \frac{a_k - L_k - \alpha_j}{T_j} < 1 + M \cdot \beta_{k-1} \tag{4.7}$$

In case $k = 1$, by the definition of the starting time instant $s_0 = 0$, it must be $\forall j < i : IL_{j,1} = 0$.
For simplicity, when $\beta_{k-1} = 1$, we force $IL_{j,k} = If_{j,k-1}$:

$$\forall j < i \quad -M \cdot (1 - \beta_{k-1}) \leq IL_{j,k} - If_{j,k-1} \leq M \cdot (1 - \beta_{k-1}) \tag{4.8}$$

	$j=1$			$j=2$		
	$k=1$	$k=2$	$k=3$	$k=1$	$k=2$	$k=3$
$If_{j,k}$	3	4	7	1	1	2
$IL_{j,k}$	0	3	4	0	1	1
$\Delta_{j,k}$	3	1	3	1	0	1
$\Gamma_{j,k}$	0	0	0	0	0	0

Table 4.2: The counting of higher priority jobs

4.3.2.9 Refining the interferences from higher priority tasks

Both $If_{j,k}$ and $IL_{j,k}$ are real variables. This is efficient but inaccurate. To restore a correct formulation, we define two classes of Boolean variables to constrain the values of $If_{j,k}$ and $IL_{j,k}$.
Given a job J_k of τ_i and a higher priority task τ_j, an array of Boolean variables $\Gamma f_{j,k}[p] \in \mathbb{B}$ counts the number of jobs (i.e., job releases) of τ_j inside the time interval (p indexes these jobs).

- $[a_k - L_k, f_k[$ if J_{k-1} does not interfere with J_k, i.e., $\beta_{k-1} = 0$;
- $[f_{k-1}, f_k[$ if J_{k-1} does interfere with J_k, i.e., $\beta_{k-1} = 1$.

A rough bound for the size of $\Gamma f_{j,k}[\cdot]$ (the number of instances of τ_j in the interval) is

$$\lceil \frac{R_i + T_i - r_i}{T_j} \rceil \tag{4.9}$$

$\Gamma f_{j,k}[p] = 1$ indicates that the pth job activation of τ_j in the specified time interval can interfere with the execution of J_k (the pth job is activated before J_k completes, otherwise, $\Gamma f_{j,k}[p] = 0$). The total number of activations of jobs of τ_j, interfering with the execution of J_k in the specified time interval is

$$\Delta_{j,k} := \sum_p \Gamma f_{j,k}[p]$$

As shown in Table 4.2, for the example in Figure 4.7, when $j = 1$ and $k = 3$ it is $L_3 = 0$ and $\Delta_{1,3}$ jobs from τ_1 within the time interval $[a_3 - L_3, f_3[$. As $\beta_1 = 1$, from f_1 to f_2, $\Delta_{1,2} = 1$.
In case J_{k-1} does not delay the execution of J_k, it is $IL_{j,k} + \Delta_{j,k} = If_{j,k}$. In the other case (i.e., when $\beta_{k-1} = 1$) $If_{j,k-1} + \Delta_{j,k} = If_{j,k}$ and (4.8) enforces $IL_{j,k} = If_{j,k-1}$. Consequently,

$$\forall j < i \;\; IL_{j,k} + \Delta_{j,k} = If_{j,k} \tag{4.10}$$

If a higher priority job $J_{j,p}$ does not interfere with the job J_k of the target task (i.e., $\Gamma f_{j,k}[p] = 0$), this implies that J_k completes before $J_{j,p}$, then no later job $J_{j,p'}$ $(p' > p)$ can interfere with J_k. This results in the precedence constraint between elements in $\Gamma f_{j,k}$.

$$\forall j < i \;\; \Gamma f_{j,k}[p+1] \leq \Gamma f_{j,k}[p]$$

Similarly, given a job J_k and a higher priority task τ_j, we define an array of Boolean variables $\Gamma L_{j,k}[\cdot]$ to count the number of job instances of τ_j inside the time interval $[f_{k-1}, a_k - L_k)$. The size of $\Gamma L_{j,k}[\cdot]$ can also be bounded (e.g., by $\lceil \frac{T_i - r_i}{T_j} \rceil$), and the total number can be computed as

$$\Lambda_{j,k} := \sum_p \Gamma L_{j,k}[p]$$

$\Lambda_{j,k}$ counts the number of jobs from a higher priority task τ_j that are guaranteed not to interfere with J_{k-1} or J_k since they are activated after J_{k-1} finishes and are not in the same busy period with J_k. For instance, during the interval $[f_2, a_3 - L_3[$ in Figure 4.7, no higher priority jobs are released: $\Gamma_{1,3} = \Gamma_{2,3} = 0$.
When $\beta_{k-1} = 0$, it is $If_{j,k-1} + \Lambda_{j,k} = IL_{j,k}$:

$$\forall j < i \;\; -M \cdot \beta_{k-1} \leq IL_{j,k} - \Lambda_{j,k} - If_{j,k-1} \leq M \cdot \beta_{k-1} \tag{4.11}$$

In case $\beta_{k-1} = 1$, the interval $[f_{k-1}, a_k - L_k[$ is not relevant to our analysis and we force $\Lambda_{j,k} = 0$. Using the big-M formulation.

$$\forall j < i \;\; -M \cdot (1 - \beta_{k-1}) \leq \Gamma L_{j,k}[p] \leq M \cdot (1 - \beta_{k-1}) \tag{4.12}$$

The constraint between variables in $\Gamma L_{j,k}[\cdot]$ resulting from the execution in FIFO order of the jobs from the same task can be encoded as

$$\forall j < i \;\; \Gamma L_{j,k}[p+1] \leq \Gamma L_{j,k}[p]$$

4.3.2.10 Constraints on the idle time and workload

In this subsection, we present the constraints that bound the processor idle times and the time spent executing by the tasks (i.e., workload) inside the problem window and its sub parts (e.g., one or multiple job windows). For short, we first define several terms: $\rho_k = f_k - a_k$, $\lambda_k = f_k - (a_k - L_k)$ and $\lambda'_k = f_k - f_{k-1}$. As an example, in Figure 4.7, $\rho_1 = 7.5$, $\lambda_1 = 7$ and $\lambda'_2 = 3$.

4.3.2.11 Minimum level-i idle time

To analyze the target task τ_i under fixed priority scheduling, it is sufficient to consider the taskset composed by τ_i and its higher priority tasks: $\mathcal{T}_i = \{\tau_1, \dots, \tau_i\}$. Given an arbitrary time interval of length X, we use $\text{minIdle}(\mathcal{T}_i, X)$ to denote the minimum amount of (level-i) processor idle time that is available within it (left unused by the tasks in $\mathcal{T}_i$).
Then, for any number x of consecutive job windows inside the problem window (of length $x \cdot T_i$), the total amount of idle time is lower bounded by $\text{minIdle}(\mathcal{T}_i, x \cdot T_i)$. That is, $\forall 1 \leq x \leq K, 1 \leq y \leq K - x + 1:$

$$\text{minIdle}(\mathcal{T}_i, x \cdot T_i) \leq \sum_{y \leq k \leq y+x-1} \iota_k \tag{C1}$$

To compute $\text{minIdle}(\mathcal{T}_i, X)$, we define a virtual task $\tau_* = (-, X, X)$ that has relative deadline and period equal to the interval length X and lowest priority. $\text{minIdle}(\mathcal{T}_i, X)$ is estimated as the maximum execution time C of τ_* that still guarantees its schedulability: if C is not the minimum level-i idle time, then there should exist a combination of job activations for tasks in $\mathcal{T}_i$ that leads to a deadline miss for τ_* (easily demonstrated by contradiction; slack stealing algorithms [26] provide methods to estimate the processor idle time).

4.3.2.12 Idle time inside a job window

Consider the job window $[a_k, a_{k+1}[$ of J_k, if $\beta_k = 0$, ι_k is in fact the idle time the interval $[f_k, a_{k+1} - L_{k+1}[$, as exemplified by the 2nd job window in Figure 4.7. The total amount of higher priority workload in $[f_k, a_{k+1} - L_{k+1}[$ can be represented as

$$\Theta_k := \sum_{j<i} (IL_{j,k+1} - If_{j,k}) \cdot C_j$$

As a result, the idle time in the kth job window is $\iota_k = (a_{k+1} - a_k) - \rho_k - L_{k+1} - \Theta_k$. This equivalence only applies when $\beta_k = 0$ and can be encoded in a MILP formulation with the following constraint (trivially true for $\beta_k = 1$).

$$\forall k \quad -M \cdot \beta_k \leq \iota_k + \Theta_k - (a_{k+1} - a_k - \rho_k - L_{k+1}) \leq M \cdot \beta_k \tag{C2}$$

4.3.2.13 Formulation of the busy period $[a_k - L_k, f_k)$ when $\beta_{k-1} = 0$

If $\beta_{k-1} = 0$, J_{k-1} does not interfere with the execution of J_k, and $[a_k - L_k, f_k[$ is a busy period with length λ_k. The total amount of workload from higher priority tasks inside $[a_k - L_k, f_k[$ is

$$\Phi_k := \sum_{j<i} (If_{j,k} - IL_{j,k}) \cdot C_j$$

For the first instance $k = 1$, it is

$$\Phi_1 + C_i - \lambda_1 = 0 \tag{C3}$$

Otherwise, $\beta_{k-1} = 0$ implies that $\Phi_k + C_i = \lambda_k$. To apply the constraint only to the case $\beta_{k-1} = 0$, the formulation is

$$\forall k \quad -M \cdot \beta_{k-1} \leq \Phi_k + C_i - \lambda_k \leq M \cdot \beta_{k-1} \tag{C4}$$

4.3.2.14 Formulation of the busy period $[f_{k-1}, f_k[$ when $\beta_{k-1} = 1$

If $\beta_{k-1} = 1$, the interval between f_{k-1} and f_k is a busy period with length λ'_k. The total amount of workload from higher priority tasks inside $[f_{k-1}, f_k[$ is

$$\Phi'_k := \sum_{j<i} (If_{j,k} - If_{j,k-1}) \cdot C_j$$

Thus, the length λ'_k of busy period $[f_{k-1}, f_k[$ can be represented as $\Phi'_k + C_i$ and the MILP constraint becomes

$$\forall k \quad -M \cdot (1 - \beta_{k-1}) \leq \Phi'_k + C_i - \lambda'_k \leq M \cdot (1 - \beta_{k-1}) \tag{C5}$$

4.3.2.15 Formulation of f_k by accumulating the idle time and workload

If we consider each job J_k, from $s_0 = 0$ to its finish time f_k, the time interval $[0, f_k[$ consists of multiple busy periods and processor idle times, which can be summed up as in

$$\forall k \sum_{j<i} If_{j,k} \cdot C_j + k \cdot C_i + \sum_{k'<k} \iota_{k'} = f_k \tag{C6}$$

4.3.2.15.1 Refining the arrival time of a higher priority job before the beginning or the end of a busy period

At the beginning or the end of a level-i busy period, a higher priority task must have completed any previously requested execution time. As a result, it must be

$$\forall k, j < i \;\; \alpha_j + (IL_{j,k} - 1) \cdot T_j + r_j - M \cdot \beta_{k-1} < a_k - L_k \tag{C7}$$

The latest activation time of a higher priority job (from τ_j) before the beginning of a busy period starting in $a_k - L_k$ is $\alpha_j + (IL_{j,k} - 1) \cdot T_j$. This job must complete before the start of the busy period in $a_k - L_k$ after at least r_j time units. The term $-M \cdot \beta_{k-1}$ is used in the constraint (C7) because L_k is only relevant when $\beta_{k-1} = 0$.
Likewise, at the end of a busy period

$$\forall k, j < i \;\; \alpha_j + (If_{j,k} - 1) \cdot T_j + r_j < f_k \tag{C8}$$

4.3.2.16 Length of a busy period

We use BP to denote the length of longest level-i busy period, and $N_i = \lceil \frac{BP}{T_i} \rceil$ is the number of jobs of τ_i within that busy period. As long as there is a busy period that spans N_i jobs of τ_i, the total task execution within it cannot exceed BP. Therefore, $\forall 1 \leq x \leq K - N_i + 1$:

$$\begin{aligned} &L_x + (a_{x+1} - a_x) - M \cdot (1 - \beta_x) + \\ &\sum_{x<k<x+N_i-1} (a_{k+1} - a_k - M \cdot (1 - \beta_k)) + \rho_{x+N_i-1} \leq BP \end{aligned} \tag{C9}$$

For arbitrarily N_i successive jobs inide an arbitrary problem window, we do not know if they are inside the same busy period, however, $\beta_k = 1$ is a sufficient condition for two jobs J_k and J_{k+1} to be in the same busy period (the same for $\beta_x = 1$) and this explains the big-M terms in (C9). For the scenario in Figure 4.7 where $BP = 11$ and $N_3 = 2$, there is a busy period $[s_0, f_2)$ that spans two jobs J_1 and J_2: $L_1 + (a_2 - a_1) + \rho_2 \leq BP$.

4.3.3 Weakly hard schedulability analysis

Given an arbitrary sequence of K successive jobs of τ_i inside the problem window, the weakly hard property specifies that the maximum number of deadline misses (among these K jobs) should be bounded by m $(< K)$. The total number of deadline misses can be computed as

$$\mathsf{nDmiss} := \sum_k b_k \tag{C10}$$

The number of deadline misses of τ_i within the problem window is bounded by m, *if the addition of the constraint*

$$m + 1 \leq \mathsf{nDmiss} \tag{C11}$$

makes the formulation non feasible.
Another option is to use the formulation of the number of deadline misses in (C10) as an optimization (maximization) function, and check what is the maximum number of misses for a given number of activations, or even the other way around, to find what is the minimum value of K given a number of deadline misses.

4.4 Extensions of the Solution Model

The weakly-hard analysis framework proposed in Section 4.3.2 can be easily adapted to a more general task model. In particular, to shared resources and tasks with jitter.

4.4.1 Shared resources

In this part, we show an extension to the case of resource sharing using the Immediate Priority Ceiling Protocol (PCP) [76] as used in the OSEK and AUTOSAR operating system standards.
A set of shared resources $\mathcal{R}_1, \dots, \mathcal{R}_G$ are accessed by tasks in mutual exclusive mode. For any task τ_i and for any resource $\mathcal{R}_g$, $\mathcal{S}_{i,g} = \{cs_{i,g,1}, cs_{i,j,2}, \dots\}$ is a finite **multiset** (a set that allows multiple instances of its elements) of worst case execution times for the critical sections executed by τ_i on $\mathcal{R}_g$.
The *priority ceiling* $pc(\mathcal{R}_g) := \min\{i : \mathcal{S}_{i,g} \neq \emptyset\}$ of $\mathcal{R}_g$ is defined as the highest priority of any task that accesses it. Every time a task accesses $\mathcal{R}_g$, its priority is boosted to the priority ceiling of $\mathcal{R}_g$. In this way, any job of τ_i can be blocked at most once by one lower priority job executing a critical section on a resource with priority ceiling $pc(\mathcal{R}_g) \leq i$. This guarantees a predictable worst-case blocking time.
For simplicity, in the following, we will assume the Rate Monotonic (RM) system such that τ_i has a higher priority than τ_j if $T_i < T_j$ and for any τ_i, $D_i = T_i$.
An arbitrary sequence of x consecutive job activations of τ_i, can be (directly or indirectly) blocked by at most x critical section executions on resources with ceiling higher than or equal to the priority of the job.

$$\mathcal{S}_i^x := \bigcup_{pc(\mathcal{R}_g) \leq i} \bigcup_{i \leq j} \overbrace{\mathcal{S}_{j,g} \cup \dots \cup \mathcal{S}_{j,g}}^{\lceil \frac{BP + x \cdot T_i}{T_j} \rceil \text{ times}}$$

Hence, for any x consecutive job windows of τ_i, the maximum blocking time is defined as the sum of the x largest elements in the multiset $\mathcal{S}_i^x$:

$$\mathcal{B}_i^x := \sum_{\text{the } x \text{ largest}} \mathcal{S}_i^x$$

To apply these blocking times to the MILP model in Section 4.3.2, we follow the common approach that adds the blocking time to the execution time when considering the possible interference.
For any $1 \leq k \leq K$, the real variable $c_{i,k}$ indicates the execution time which includes the blocking time that the kth job of τ_i, within the problem window, can suffer.

$$C_i \leq c_{i,k} \leq C_i + \mathcal{B}_i^1 \tag{4.13}$$

For any number of consecutive job windows, we can bound the sum of all these execution variables: $\forall 1 < x \leq K\ \forall 1 \leq y \leq K - x + 1$

$$\sum_{y \leq k \leq y+x-1} c_{i,k} - x \cdot C_i \leq \mathcal{B}_i^x \tag{C12}$$

To extend the original problem formulation to the case of resource sharing, all instances of C_i in constraints (C3)~(C6) should be replaced by the corresponding variable $c_{i,k}$ Also the definition of the minimum processor idle time needs to be modified and the constraint (C1) is then updated as follows

$$\text{minIdle}(\mathcal{T}_i, x \cdot T_i) - \mathcal{B}_i^x \leq \sum_{y \leq k \leq y+x-1} \iota_k \tag{C1*}$$

4.4.2 Jitter

The jitter [18] of a periodic task represents the maximum possible delay of the task actual activation times with respect to the ideal periodic activations. Given a periodic task $\tau_l = (C_l, D_l, T_l)$, we denote its jitter as $\mathcal{J}_l$ with $\mathcal{J}_l + C_l \leq D_l$.
Because of jitter, the distance between the activation times of two jobs J_k and J_{k+1} of the target task τ_i inside the problem window is not a fixed value T_i, but can be any value within the range $[T_i - \mathcal{J}_i, T_i + \mathcal{J}_i]$. More generally, there is $\forall 1 \leq k < K, 1 \leq N \leq K - k$,

$$N \cdot T_i - \mathcal{J}_i \leq a_{k+N} - a_k \leq N \cdot T_i + \mathcal{J}_i$$

The jitter of a higher priority task τ_j also affects its interference upon the task under analysis. For example, the number of jobs of τ_j that arrive before the finish time of the kth job of τ_i within the problem window becomes: $\lceil\frac{f_k-\alpha_j-\mathcal{J}_j}{T_j}\rceil \le If_{j,k} \le \lceil\frac{f_k-\alpha_j+\mathcal{J}_j}{T_j}\rceil$. This is encoded by the constraint below, as a replacement of (4.6).

$$\forall j < i \;\; \frac{f_k-\alpha_j-\mathcal{J}_j}{T_j} \le If_{j,k} < \frac{f_k-\alpha_j+\mathcal{J}_j}{T_j} + 1 \tag{4.14}$$

For $IL_{j,k}$, when $\beta_k = 0$, it is now $\lceil\frac{a_k-L_k-\alpha_j-\mathcal{J}_j}{T_j}\rceil \le IL_{j,k} \le \lceil\frac{a_k-L_k-\alpha_j+\mathcal{J}_j}{T_j}\rceil$, and the big-$M$ constraint in (4.7) is updated to $\forall j < i$

$$\frac{a_k-L_k-\alpha_j-\mathcal{J}_j}{T_j} - M\cdot\beta_{k-1} \le IL_{j,k} \tag{4.15}$$

$$IL_{j,k} < \frac{a_k-L_k-\alpha_j+\mathcal{J}_j}{T_j} + 1 + M\cdot\beta_{k-1} \tag{4.16}$$

To take into account jitter, several equations in the MILP formulation also need to be updated (the jitter mostly result in a modifier applied to periods). Summarizing (the full justification is omitted for space reasons), Equations (4.1), (4.2), (4.3), (4.4), (4.9), (C1) (C7), (C8) are replaced with the following

$$\forall k \;\; 0 \le L_k \le T_i - r_i + \mathcal{J}_i \tag{4.17}$$

$$\forall j < i \;\; 0 \le \alpha_j \le T_j - r_j + \mathcal{J}_j \tag{4.18}$$

$$\forall k \;\; C_i \le f_{k+1} - f_k \le R_i + (T_i - r_i) + \mathcal{J}_i \tag{4.19}$$

$$\forall k \;\; -M\cdot b_k \le a_k + D_i - f_k - \mathcal{J}_i < M\cdot(1-b_k) \tag{4.20}$$

$$\lceil\frac{R_i+T_i-r_i+\mathcal{J}_j}{T_j}\rceil \tag{4.21}$$

$$\text{minIdle}(\mathcal{T}_i, x\cdot T_i - \mathcal{J}_i) \le \sum_{y\le k\le y+x-1} \iota_k \tag{4.22}$$

$$\forall k, j < i \;\; \alpha_j + (IL_{j,k}-1)\cdot T_j + r_j - M\cdot\beta_{k-1} - \mathcal{J}_j < a_k - L_k \tag{4.23}$$

$$\forall k, j < i \;\; \alpha_j + (If_{j,k}-1)\cdot T_j + r_j - \mathcal{J}_j < f_k \tag{4.24}$$

4.4.3 Experiments

In the section, we apply the proposed weakly hard schedulability analysis to an automotive engine control application and a set of randomly generated system configurations. All experiments are conducted on a machine with 8 GB memory and 8 cores: Intel(R) Xeon(R) CPU X3460 @ 2.80GHz, using CPLEX 12.6.3 as the MILP solver. The MILP formulation is encoded in C++ using the CPLEX library and is available for download[1].

The fuel injection case study At first, we apply the MILP weakly hard schedulability analysis with the shared resource extension (Section 4.4), to the fuel injection application described in [15].

According to the AUTOSAR standard, an automotive application is composed by a set of functions called runnables, which are executed by tasks scheduled by fixed priority. The runnable-to-task mapping and the task scheduling are defined at the system integration phase.

For the fuel injection application in [15], a heuristic strategy is applied to allocate approximately 1000 runnables to tasks with 280 critical sections. The resulting taskset has 15 tasks with priorities assigned according to the Rate Monotonic rule (all times in microseconds)

Due to the blocking from τ_{15}, τ_{14} is not (hard real-time) schedulable.

To verify the weakly hard schedulability property, we tested a series of m-K parameters: $\{(1,5),(2,5),(2,10),(3,10),(3,15),(4,15)\}$. According to our weakly hard schedulability analysis, it is guaranteed that there will be at most $m = 2$ (resp. 3 and 4) deadline misses out of any $K = 5$ (resp. 10 and 15) consecutive jobs of τ_{14}.

Regarding the runtime cost, except for the case $m = 3$ and $K = 15$, all tests complete within 2 minutes. It takes the CPLEX solver almost 30 minutes to make a decision when $m = 3$ and $K = 15$. **Runtime performance** In this subsection, we apply the weakly hard real-time analysis in Section 4.3.2 to a set of randomly generated tasksets for an empirical evaluation of the runtime performance, with a large variety of configurations: $n \in \{10 \sim 15, 20, 30, 50\}$, $U \in \{0.8, 0.85, 0.9, 0.95\}$, $m \in \{1,2,3\}$ and $K \in \{5, 10 \sim 15, 20\}$. Each configuration in

[1] https://github.com/m-k-wsa/

Task	C_i	T_i	Task	C_i	T_i
τ_1	1015.83	$2 \cdot 10^4$	τ_8	5296.84	$1 \cdot 10^5$
τ_2	2309.5	$2 \cdot 10^4$	τ_9	325.64	$2 \cdot 10^5$
τ_3	1148.64	$2.5 \cdot 10^4$	τ_{10}	3285.24	$2 \cdot 10^5$
τ_4	2419.6	$3 \cdot 10^4$	τ_{11}	208.67	$5 \cdot 10^5$
τ_5	287.5	$5 \cdot 10^4$	τ_{12}	539.5	$5 \cdot 10^5$
τ_6	51.072	$6 \cdot 10^4$	τ_{13}	47616.3	$1 \cdot 10^6$
τ_7	2318.42	$1 \cdot 10^5$	τ_{14}	799006	$2 \cdot 10^6$
			τ_{15}	$1.0005 \cdot 10^6$	$1 \cdot 10^7$

Table 4.3: An automotive case study

the experiment is specified by a tuple (n, U, m, K), where n is the taskset size, U is the taskset utilization, and m-K is the weakly hard property to be checked.
Overall, 6253 task systems are tested. For each taskset with a pair n and U: 1) the utilization U_i of tasks is generated using the Randfixedsum algorithm in [35]; 2) the task period T_i is uniformly sampled in the range $[10, 1000]$; each task has an implicit deadline, i.e., $D_i = T_i$; 3) and each task WCET is computed as $C_i = T_i \cdot U_i$.
Tasks are assigned priorities according to the Rate Monotonic rule. If the lowest priority task τ_n in the taskset is schedulable, the taskset is abandoned; otherwise, we proceed with the weakly hard real-time analysis on τ_n. This configuration is designed to stress the weakly hard analysis, since even if the lowest priority task τ_n is schedulable there may exist other non-schedulable tasks with a smaller number of interfering higher priority tasks for the m-K analysis.
In the analysis of each taskset we defined a runtime limit of 1800 seconds: if the analysis takes more than 1800 seconds without terminating, we stop and report a failure of the m-K analysis.
Deadline misses in a row
The m-K model discussed so far concerns the upper bound on the number of deadline misses (m) out of any K consecutive task activations. Another popular pattern for weakly hard schedulability analysis is to check if there are **more than m deadline misses in a row**, which is equivalent to analyze the m-K model with $K = m + 1$.
In the following, we evaluate the number of cases in which there are $K = 2$ and $K = 3$ consecutive deadline misses, when $U = 0.95$ and $n \in \{10, 20, 30, 50\}$. Results are shown in Table 4.4 and for every test case, the MILP solver returns its decision in less than a minute. Consecutive deadline misses seldom happen even when the total utilization is as high as 0.95. Another observation is that the fraction of cases with consecutive deadline misses is not sensitive with respect to the number of tasks in the set.

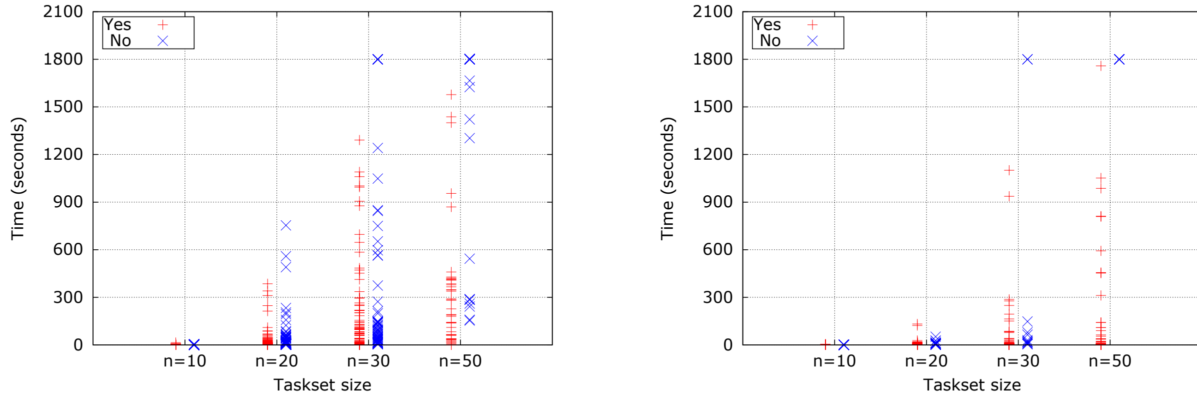
	$K = 2$	$K = 3$		$K = 2$	$K = 3$
$n = 10$	80.3%	98.6%	$n = 30$	85.1%	99.9%
$n = 20$	84.8%	99.6%	$n = 50$	84.9%	99.9%

Table 4.4: Percentage of sets with K consecutive deadline misses

Varying the taskset size
Table 4.5 and Figure 4.9 show the experimental results with a variable taskset size, when $U = 0.85$. The weakly hard analysis confirms that a large portion of these non-schedulable tasks will never miss more than 1 deadline out of any 5 its consecutive activations. For example, when the taskset size is 20 or 30, the percentage of *1-5* feasible sets is around 50%. When m is increased to 2, more than 90% of the tested tasksets satisfy the specified m-K property in all cases.
On the other side, when the taskset size is very large ($n = 50$), for $m = 1$ a significant amount of tests exceed the runtime limit of 1800 seconds, which implies that a longer runtime is needed for such cases. Figure 4.9 depicts the time spent on the weakly hard analysis of each taskset: *Yes* labels that the corresponding m-K property is verified (*No* in the other case). The majority of analyses return the decision within 10 minutes.
Varying the problem window size
Table 4.6 contains the experiment results when varying the problem window size K, with $n = 10$ and $U = 0.85$. The problem window size K is a dominant factor with respect to the complexity of the analysis. Still, the results are promising and for more than one third of the tasksets, the number of deadline misses is bounded by at most $m = 2$.
Varying taskset utilization.

	$m = 1, K = 5$		$m = 2, K = 5$	
	confirmed	n/a	confirmed	n/a
$n = 10$	42.8%	0%	90.6%	0%
$n = 20$	49.0%	0%	91.9%	0%
$n = 30$	54.4%	6.5%	92.9%	1.2%
$n = 50$	42.7%	41.9%	94.0%	6.0%

Table 4.5: Experiments that confirm the m-K property and run out of time limit (n/a) with variable n



(a) $m = 1$ Figure 4.9: Runtime results for $K = 5$ (b) $m = 2$

	$m = 2$	$m = 3$		$m = 2$	$m = 3$
$K = 11$	54.4%	84.8%	$K = 14$	47.0%	73.6%
$K = 12$	51.3%	81.5%	$K = 15$	45.8%	66.4%
$K = 13$	49.2%	76.5%	$K = 20$	33.6%	56.1%

Table 4.6: Percentage of valid m-K property with variable K

	$m = 2, K = 10$	$m = 3, K = 10$
$U = 0.80$	90.8%	99.1%
$U = 0.85$	60.8%	86.1%
$U = 0.90$	30.2%	50.6%
$U = 0.95$	6.8%	17.6%

Table 4.7: Percentage of valid m-K property with variable U

Table 4.7 shows the percentage of tasksets that satisfy the m-K property with variable taskset utilization levels and a fixed taskset size $n = 10$.
Even when the taskset utilization is very high ($U = 0.90$), more than 30% of the non-schedulable tasks will not miss more than $m = 2$ deadlines within any sequence of $K = 10$ successive task activations. If we further increase m to 3, the tasksets satisfying the weakly hard property become half of the generated sets.

4.5 Optimizing the placement of time-critical automotive tasks in multicores

Several application developers are currently faced with the problem of moving a complex system from a single-core to a multicore platform. The problem encompasses several issues that go from modeling issues (the need to represent the system features of interest with sufficient accuracy) to analysis and optimization techniques, to the selection of the right formulations for constraints that relate to time. We report on the initial findings in a case study in which the application of interest is a fuel injection system. We provide an analysis on the

limitations of AUTOSAR and the existing modeling tools with respect to the representation of the parameters of interest for timing analysis, and we discuss applicable optimization methods and analysis algorithms.

Several application developers are currently faced with the problem of moving a complex system from a single-core to a multicore platform. The problem encompasses several issues that go from modeling issues (the need to represent the system features of interest with sufficient accuracy) to analysis and optimization techniques, to the selection of the right formulations for constraints that relate to time. We report on the initial findings in a case study in which the application of interest is a fuel injection system. We provide an analysis on the limitations of AUTOSAR and the existing modeling tools with respect to the representation of the parameters of interest for timing analysis, and we discuss applicable optimization methods and analysis algorithms.

The problem of partitioning a real-time application onto a multicore platform is very well studied. For most metric and constraints of interest, finding an optimal allocation is an NP-hard problem in the strong sense, and solutions derived based on (often bin-packing) heuristics are proposed [54, 29, 67]. In the case of our application of interest, the constraint that we consider for the partitioning objective is time. With respect to real-time schedulability, most available results assume tasks as the units of placement and scheduling [64, 36]. When the partitioning problem is addressed at the task level, the variables that need to be determined are the placement and the assignment of priorities, and the decision is based on considerations on the worst case completion times of tasks, which are affected by preemption and the possible blocking on shared resources. Shared resources can be physical (such as I/O devices) or logical (such as communication buffers). In both cases, the critical sections runtimes are additional possible blocking times to be considered. Since the cost of intercore sharing is typically much more than the intracore blocking time, tasks are often clustered based on their resource and communication dependencies. Automotive applications use the AUTomotive Open System ARchitecture (AUTOSAR) [81], standard for the representation of the application architecture and the execution platform. In AUTOSAR, applications and threads are composed by a set of functions, called runnables, that are executed periodically or in response to an event (physical, such as an interrupt, or logical). Runnables are grouped into tasks, which are the unit of scheduling of the AUTOSAR Operating System [81]. The runnable-to-task mapping and the task scheduling of an application is defined as application configuration, to be performed statically, at system integration time. Ideally, all dependent runnables should be grouped into a single task, reducing as much as possible the inter-task communications and resource dependencies. This is the approach that is recommended by AUTOSAR for multi-core systems [81]. Following this model, Monot et al. [64] presented a scheduling algorithm for AUTOSAR applications on multi-core ECUs.

However, in a fuel injection application separation is not possible because of the large number of interdependencies, and a task-driven allocation using the tasks defined for the legacy single-core application is clearly ineffective, because of the large number of runnables mapped onto the same task and the limited degree of freedom for the placement. Faragardi et al. [36] presented a scheduler for AUTOSAR applications on multi-core ECUs having as scheduling criteria to minimize the worst-case communication delays among runnables scheduled in different cores. This solution assumes that the runnable-to-task mapping that minimizes the communication among cores is already given, and so it focuses only on the task scheduling. In our approach, we start from the runnables model and the construction of the set of tasks is one of the objectives of the procedure. To evaluate the impact of shared resource on the timing properties, we will assume a lock-based and spin-based resource locking mechanism, based on MSRP [38], which provides a predictable worst-case blocking time. As mandated by AUTOSAR, MSRP uses non-preemptive spin locks for managing the access to global resources, but it also enforces FIFO waiting queues to obtain tighter worst-case blocking times. Other mechanisms for resource sharing that are suitable for implementation on AUTOSAR systems are discussed in [16] and [16, 17].

Our approach builds on optimization approaches for task partitioning that consider MSRP blocking times such as the algorithm proposed in [89], which works at the task level and introduces a mixed-integer programming solution to the problem. Also, for priority assignment, we build on the results of the optimization algorithm presented in [89].

The algorithms proposed in these papers assume a task characterization based on a periodic or sporadic (i.e. with minimum interarrival time) activation. Control tasks for fuel injection applications are not only composed of periodic tasks since some of them are activated in correspondence with the phase of the crankshaft and are often implemented to adapt their execution time to the engine speed [20]. This model, denominated Adaptive Variable Rate (AVR) is challenging for the objective of timing analysis, and recent works present algorithm for accurate analysis [28, 14]]. Finally, the timing properties of tasks executing on multicores are also dependent on the architecture features of the execution platform, including caches and communication architectures. The impact of these features on time predictability is treated as part of the project for the definition of Single Core Equivalent Virtual Machines [75, 74].

4.5.1 Modeling the Functionality and the Platform

The objective of our work is to build models that allow to analyze the partitioning solutions with respect to time and then to define optimization algorithms that can find possible partitioning options for further manual refinement by the designers. Application and BSW models have been constructed by extending existing AUTOSAR models of the legacy application with the description of all the runnables in the application model, as well as in the BSW, together with a model of the other BSW components, including the drivers and the OS, their time characteristics and the structure of the target microcontroller with sufficient detail to analyze the impact on the response time of the tasks of the sharing of physical resources.

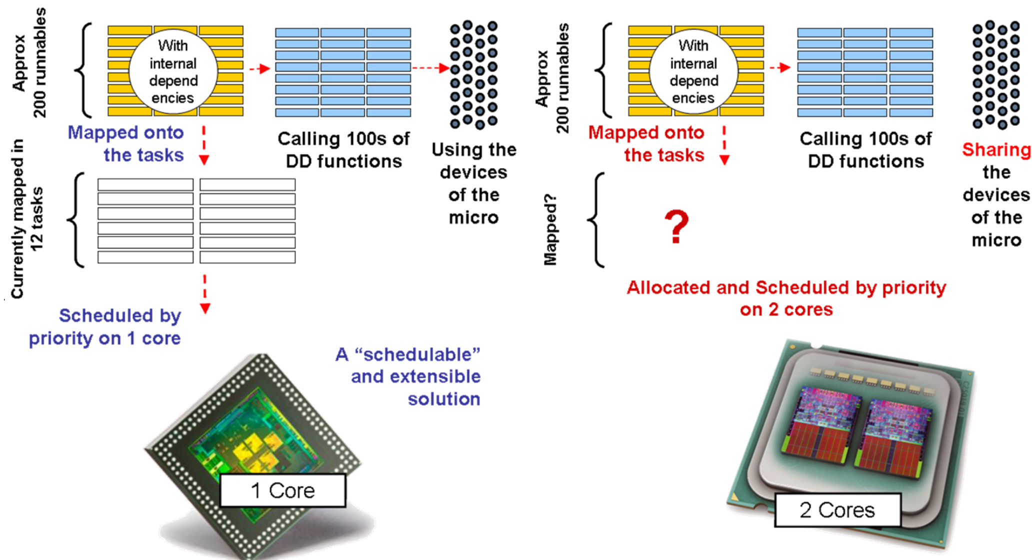


Figure 4.10: The main characteristics of the mapping problem.

The model of the fuel injection application and its BSW has been defined in Rhapsody (with its AUTOSAR 4.0 modeling extensions) and consists of approximately 200 application-level runnables plus 200 BSW runnables and functions (Figure 4.10). The entities that need to be considered for the mapping problem are the runnables (application and BSW) and the allocation of the BSW device driver (DD) functions, which are in turn dependent on the use of the microcontroller peripheral devices by the runnables executing on them. The objective of the mapping is to

- define the set of tasks to be executed on each core;
- assign a priority to each task;
- define the sequence of runnables to be executed by each task;
- define the allocation of the device driver functions to the cores;

with the following constraints:

- each runnable is executed by a task with a period that is the same of its trigger event, in case it is a timed event, or it is executed by a task activated by the same event. Both constraints are actually more restrictive than needed, in order to avoid pessimism in the following timing analysis;
- each task must complete before it is activated again (implicit deadline).

In the definition of our problem, we also made use of the following approximations:

- the execution time of each task is simply computed as the sum of the execution time of all runnables mapped onto it;
- the execution time of a runnable does not depend on where it is allocated and each access to memory globally shared requires the same time as access to local memory.

In essence, the partitioning algorithm requires knowledge of all dependencies among runnables because of communication and synchronization, as well as all the dependencies between runnables and logical and physical resources. Communication and synchronization dependencies can be extracted from the AUTOSAR model

by identifying the data dependencies on send-receive ports and the calls on client-server ports. The communication dependencies between runnables on data ports are characterized by the size (type) of the data item flowing on these ports. Dependencies on logical and physical resources can be found by tracing the chain of calls from runnables to BSW runnable to device drivers and the type of HW resources that each device driver needs or accesses. Finally, we need a time characterization of each runnable in terms of its worst-case execution time and the event that triggers it (for application runnables) and the set of order of execution dependencies among runnables (when defined). Given all the approximations and assumptions, the final result of the algorithm is not meant to be guaranteed as optimal, neither are the estimated response times guaranteed to be safe with sufficient accuracy. Neverthless, the purpose of the evaluation is not the evaluation of the solutions in absolute terms, but rather the comparative evaluation and the selection of a mapping of good quality for a first exploration and possible further manual improvement.

4.5.2 Fuel injection applications and AVR tasks

The application tasks of the legacy application (and the runnables mapped onto them) are of two types: periodic tasks, activated at different rates, and tasks activated at specific angles of the engine crankshaft (with respect to the Top Dead Center -TDC- of a reference cylinder). The timing analysis of the crankshaft tasks is especially challenging. Traditional schedulability theory deals with the periodic or sporadic execution model, in which any two task activations must be separated by a minimum time interval. In this case, for each periodic or sporadic task τ_i having priority i, its worst-case response time can be computed using [38]:

$$R_i = C_i^* + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^* \tag{4.25}$$

Where C_i^* is the worst-case execution time of the task τ_i (obtained as the sum of the runnables executed by it plus it worst-case spin time, as explained in the following), $j \in hp(i)$ indicates the set of all the tasks τ_j with priority j higher than or equal to i allocated on the same core of τ_i, T_j indicates the period (or minimum interarrival time) of τ_j, and Bi is the worst-case blocking time. A blocking time is defined as the time spent by a higher priority task waiting on a resource (such as waiting a critical section for accessing a shared port or inside a device driver) that is currently locked by a lower priority task. In the case of MSRP, the term B_i can be computed as

$$B_i = \max(B_{li}, B_{gi}).$$

B_{li} is the worst-case local blocking time computed as the longest local critical section (shared device driver or data access functions) that can be executed by a (runnable of a) task with priority lower than τ_i on a resource with a ceiling higher than or equal to i, where the ceiling of a resource is the highest priority of any task that can possibly use it. Global resources are assigned with a ceiling that is higher than that of any local resource. A task that fails to lock a global resource (in use by another task) spins on the resource lock until it is freed, keeping the processor busy. B_{gi} is the global blocking time and it is bound in the worst case to be no larger than the sum of the spin time $L_{k,i}$ plus that worst case length of any possible blocking critical section $CS_{j,k}$ executed by a task τ_j allocated on the same core as τ_i on a global resource (with index k).

$$B_{gi} = \max_{j,k}\{CS_{j,k} + L_{k,c(i)}\} \tag{4.26}$$

The spin time $L_{k,c}$ is computed for each global resource and core c as the sum of the maximum length of any critical section using the resource executed by any task τ_w on a different core (at most one term of the sum for each core).

$$L_{k,c} = \sum_{w,c(w) \neq c} \max_{w,k}\{CS_{w,k}\}$$

For a dual core system, the sum only consists of one element (relative to the other core). The sum of all the spin times for all the resources accessed by τ_i (indicated with the notation $k \in R(i)$) needs to be computed as part of C_i^*, that is

$$C_i^* = C_i + \sum_{k \in R(i)} L_{k,c(i)}$$

A more accurate blocking analysis has been presented in [88]. The sporadic model is quite pessimistic in the case of tasks activated by the crankshaft and would practically consist in the evaluation of the maximum

worst-case execution time at the largest rotation speed (highest rpm) only. In addition, several fuel injection applications are programmed for a behavior that is adaptive with respect to the execution rate (and the available computation time). This means that these tasks are executed in different modes according to the speed of the crankshaft. At lower rpm, when more time is available between any two activations, a more complex control algorithm may be executed. At higher rpm, a simplified implementatio n is used to avoid overloading the processor and trespassing the deadlines. The worst case response time analysis of AVR tasks can be performed using the model in [14]. Because of space limitations, we do not explain the algorithm here, the interested reader can refer to the reference. However, the exact analysis is not expressed in closed form as the analysis in (1), but requires the execution of an algorithm.

4.5.3 Analysis Models

The objective of our analysis have been defined in the Introduction. In the following, we outline the choices that we made with respect to the analysis algorithm, and the customized extensions that we added to the AUTOSAR model of the application in Rhapsody to represent the custom application with higher accuracy. Approximating the AVR behavior with modes The available real-time analysis methods (both exact and with a good degree of approximation) for a set of AVR tasks cannot be expressed in closed form. For the purpose of our early analysis we chose to evaluate the mapping for 6 different ranges of speeds, from 500 to 6500 rpm, assigning the mid-value of the speed range to the AVR tasks and treating them as sporadic. This means that in the formulation of the optimization problem, the analysis is performed using (1). AUTOSAR extensions The analysis of the application for the purpose of partitioning requires several extensions to the modeling elements currently available in AUTOSAR. Among the extensions that are required, we found that

- worst case execution times should be associated with runnables, not tasks. In addition, many application and BSW runnables are executed in response to multiple events or invoked for different services and their execution time varies in a significant way according to the trigger event, the type of service or even the execution mode;
- the type of I/O device that is accessed by a DD function (a runnable in a BSW component) must be represented to address the partitioning of the runnables on the basis of the HW resources available at each core (and the possible contention/blocking on their DD functions).

Modeling entities to represent these features are not available in AUTOSAR (and therefore are not found in the corresponding profile offered by Rhapsody). However, the AUTOSAR modeling concept are available in Rhapsody as stereotyped UML elements. This allows to define a custom profile and additional stereotypes to provide the needed attributes to our elements of interest. For example, to add mode-dependent worst-case execution times (WCET) to runnables the stereotype shown in 4.11 has been defined with three values to define the WCET value, it units and the mode (trace id) where it is measured. In addition, to identify exactly the dependencies for shared hardware resources and shared DD functions, we had to define stereotypes for all types of devices that are needed from the platform (typically later mapped onto microcontroller pins). Each runnable call of a BSW or DD service was then associated with the stereotype identifying the device type for which the call was issued. This stereotype ≪IDN≫ has a single property of type itemized that describes the physical IO type that is requested. Figures 4.11 and 4.12 show Rhapsody screenshots of the stereotypes and itemized types for the specification of hardware resources and for the assignment of execution time to runnables.
Another limitation of AUTOSAR models is the lack of sufficient accuracy to model cache and options for the placement of code in the cache. In the real-time execution of runnables in multicores, the impact of caches (or scratchpads) is going to be very important. However, in our preliminary analysis we skipped considerations about the placement of code in cache and caching effects. With the exception of the previous extensions, all the required information for our synthesis problem was encoded using standard AUTOSAR features (ports, interfaces, data values and their types, call dependencies and so on). The AUTOSAR model has been constructed in part manually, in part by automatic analysis of the program features and its timing characteristics using static code analysis and trace analysis tools. The data has been loaded in the AUTOSAR model using the Rhapsody Java API and scripts. The model has been then reviewed and completed manually. The AUTOSAR model contains the input data for the analysis but does not contain the definition of the optimization constraints or the analysis algorithm. The optimization problem is encoded and solved by an external tool.

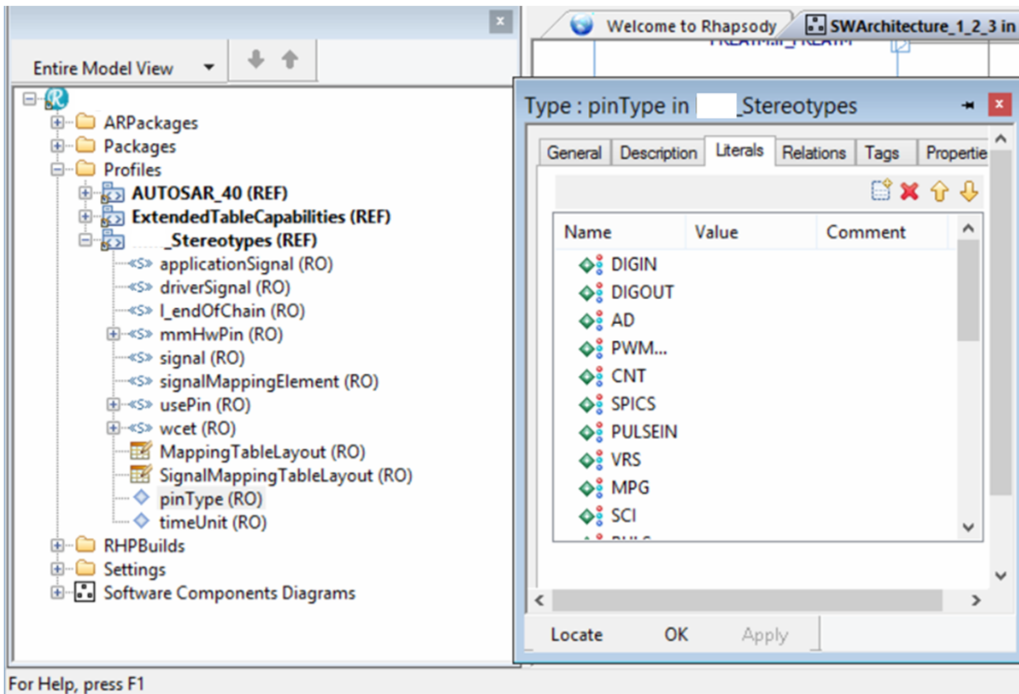


Figure 4.11: Stereotypes in Rhapsody for modeling hardware (IO) resources in AUTOSAR.

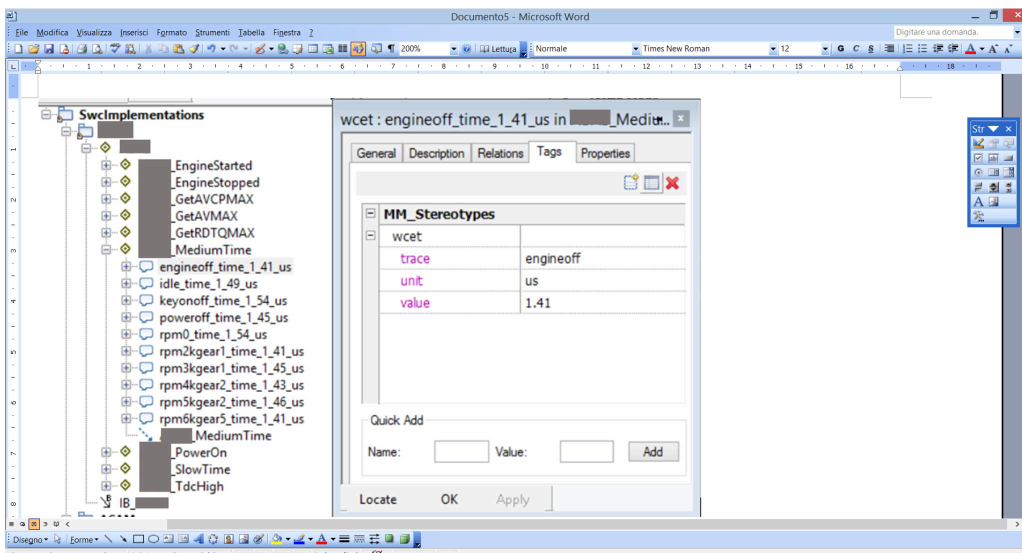


Figure 4.12: Stereotypes in Rhapsody for the representation of the execution time of runnables (extracted from traces) as a function of the activation event and the execution mode.

4.5.4 Optimization Algorithms

Our analysis draws input data from the AUTOSAR model and then passes the data to an optimization engine that computes possible solutions. The input data is extracted from AUTOSAR using the Rhapsody API and saving the features of interest in a custom format. The first optimization engine that we tried is a simulated annealing function. The next step will consists of exploring a formulation and an optimization algorithm based on mathematical optimization using a MILP encoding. Simulated Annealing The initial approach to the problem is to analyze each range of crankshaft speeds separately and to detect whether there are significant differences in the computed allocation solutions for each mode. The algorithm that has been selected for this initial investigation is a simple Simulated Annealing solution (a stochastic optimization method). The method draws its name from the way liquid metals anneal and reach a configuration with low energy content (close to optimal) when the temperature is lowered and they go back to the solid state. Simulated annealing is in essence a local optimization method that is modified to improve the chances of escaping from local optima. The modification consists in the selective acceptance of higher cost and possibly even infeasible solution during the search. At

each point in time in the algorithm maintains a current solution, the best solution found up to that point, and a temperature parameter T that is slowly lowered with time. The algorithm uses a transition operator to compute a new solution starting from the current one, and a cost evaluation function to compute the cost C of each solution. At each iteration, the newly computed solution is always accepted if it has lower cost (it improves upon) the existing solution and it is also accepted with probability

$$p = e^{-\frac{\Delta C}{T}}$$

if its cost is higher than the existing solution. The probability tends to zero when T approaches zero and is lower for higher cost differences ΔC. A pseudocode implementation of the main algorithm is shown in Figure 4.13.

```
void anneal(init_temp, final_temp, coolrate, MAXTRY,
MAXCHANGE)
   cur_sol = Compute_First_Allocation();
   cur_val = Evaluate_Solution(cur_sol);
   while temp > final_temp do
      for k = 0; k < MAXTRY; k + + do
         new_sol = ComputeNewAllocation();
         new_val = EvalSolution(new_sol);
         valid = ischange(new_val-cur_val, temp);
         if valid then
            cur_val=new_val; cur_sol=new_sol;
         UpdateBestSolution(cur_sol, cur_val);
      end for
      temp -= coolrate;
   end while
```

Figure 4.13: Pseudo-code of the Simulated Annealing routine.

The algorithm requires the definition of the transition operator that generates a new allocation solution from an existing one and a metric to evaluate the quality of solutions. The transition operator consists in the random selection of one of the following operations:

- a runnable is extracted from an existing task, and a destination task with the same period is selected on a randomly selected core. If there is no possible destination task, a new one is created and a random priority level is assigned to it. The position of the runnable is selected in accordance with the order of execution.
- Two tasks are randomly selected (on the same core) and their priorities are swapped;
- Two tasks are randomly selected (on different cores) and their allocation is swapped.
- A task is randomly selected. A partition of its runnables is randomly selected and moved to a different core as a new task with a randomly assigned priority

The evaluation (performance) function is quite simple. It consists of computing the worst case response time of all tasks on the cores and then the laxity, defined as the difference between the deadline and the response time. The minimum laxity among all the tasks is the performance function and the cost function is simply obtained by changing its sign.

$$C = \max_i (R_i - D_i)$$

where the index i spans over all the tasks. This definition allows to select also non-feasible solutions in the intermediate steps of the algorithm. The application of the simulated annealing algorithm to our problem was indeed able to find a solution, at least within the limitations of our approximations.

4.5.5 Linear Optimization

The main problems with most stochastic optimization approaches are the need to trade-off execution time with the quality of (or even the chance of finding) solutions, and especially the lack of any measure of the quality of the solution that is returned by the algorithm. Optimization algorithms based on linear or convex formal mathematical formulations offer several advantages. At each step in the algorithm, the duality theorem provides an upper bound on the distance of the current solution from the optimal one, and when optimality is achieved, it can be formally guaranteed. In addition, and probably most important, once the problem is formulated as a linear or convex optimization, it can be solved using one of the solver programs that are widely

available as commercial or open source, such as CPLEX from IBM [45]. These programs have been finely tuned to achieve extremely good performance and leverage a great amount of research work in the field of mathematical optimization. For the allocation problem at the level of tasks, a formulation that accounts for the effect of blocking times has been proposed in [89] and quickly discarded in favor of a heuristic, given that the authors in [89] found that for four cores, the MILP formulation could not be solved for more than 20 or 30 tasks. When shifting the problem to the optimization of runnables, clearly 20 or 30 is too small a number, even if the platform that needs to be optimized is a simpler dual core.

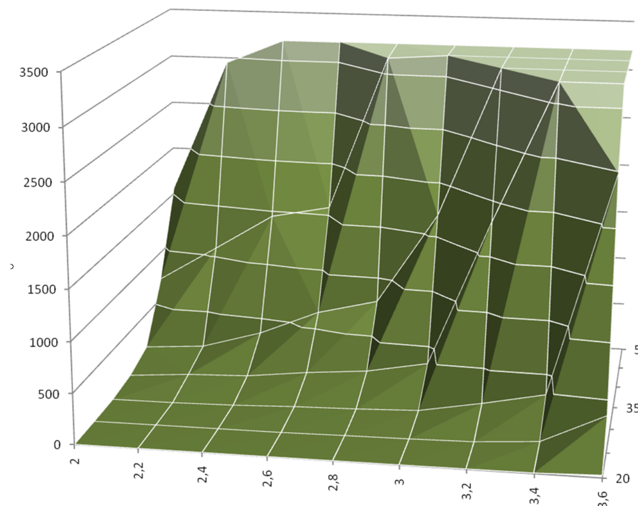


Figure 4.14: Runtime of the MILP Optimization problem.

However, our preliminary findings on the formulation in [89] reveal that it is most likely quite pessimistic. By removing unnecessary variables and simplifying the formulation (for priority encoding, using the approach in [92] and [30]), by adding very simple constraints on the utilization of each core (trivially forced to be less than 100%) and improving the definition of other constraints, we were able to speed up the solution to more than two orders of magnitude. Initial results on the allocation problem on four cores show the average runtimes (on 50 runs) shown in Figure 4.14. The runtimes (in seconds) are plot against the number of runnables/tasks and the overall utilization on the cores. The experiment setting is the same as in [89]. Sets of 50 runnables/tasks can be handled in less than one hour for utilizations of 2.6 or less. Considering that the load of the truly time critical tasks may be in the same range (60 to 70%) and that the two core optimization problem is significantly simpler (the number of many variables is proportionally reduced), we expect to be close to the practical applicability to our problem.

4.6 Vulnerability Detection for Multi-Cores

The target platform for multicore contention modelling was the one of the Telecom use case . However, finally this methodology will be ported to the Telecom use case through extrapolation in WP4 and to the Automotive multicore use case in WP4 and WP6. In WP4 the relevant platform for the Telecom use case and an internal avionics prototype is the ARM Juno board, which is a complex multicore processor, whereas the Automotive multicore use case uses a simpler Infineon AURIX TC27x board. Therefore, methods in WP3 will target the ARM Juno board platform given that the AURIX one just poses a subset of the challenges posed by the Juno board. Thus, solutions valid for the latter are a superset of the solutions needed for the former.
Hence, this section identifies the main hardware shared resources in the Juno board and how their sharing can affect execution time of the different tasks running. This challenges the estimation of execution time bounds, which is a form of vulnerability since safety real-time tasks need reliable execution time bounds for being properly scheduled. Therefore, a methodology is provided to upper bound the impact in execution time of contention when accessing hardware shared resources.
Note that Sections 4.6.1 and 4.6.2 have been only slightly modified w.r.t. the contents reported in D3.1 to reflect the shift from the DragonBoard to the Juno board. Instead, Section 4.6.3 is a completely new contribution not reported before.

4.6.1 Shared Hardware Resources in the Juno board

The hardware architecture of the ARM Juno board is depicted in Figure 4.15. It is composed of two clusters. The first one is a **high-performance** A57 dual-core cluster, with a 2MB L2 cache shared between the 2 cores, and private L1 caches. The second one is a **low energy** A53 quad-core cluster sharing a smaller 512KB L2 cache between the cores, also encompassing private L1 caches (though being smaller than for the other cluster).

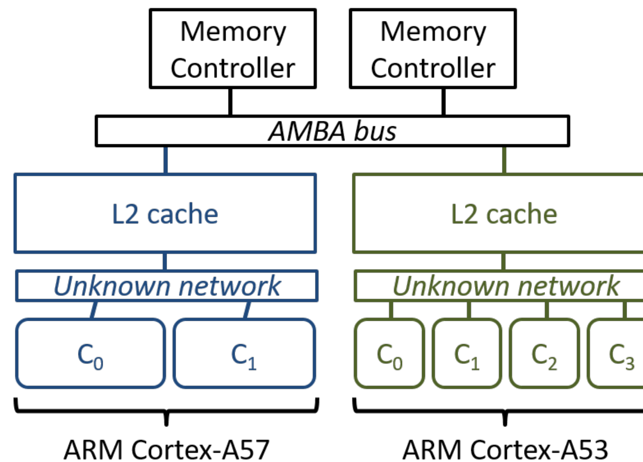


Figure 4.15: Shared Hardware resources alongside the memory path on the ARM Juno board. As shown each of the two clusters includes a shared L2 cache and communication channels towards the DDR memory controllers.

The clusters are connected to the main memory through a shared CCI400 AMBA bus and then through an Application Fabric interconnect, itself connected to a dual port 32-bit DDR controller.
These interconnect shared hardware resources are also shared by other hardware resources, like some low-level peripherals for the AMBA bus, the multi-media subsystem for the Application Fabric, as well as the System Bus connecting all the peripherals (PCI-X, UART, SPI, ...).

4.6.2 On-Chip Cacheless Resource Sharing

The research on timing analysis for multi-core processors is still in its infancy. Especially so for Commercial off-the-shelf (COTS) multi-cores, whose timing analysis is a complex challenge that needs to be solved before their adoption in safety-critical real-time systems industry may become viable. Deriving an Execution Time Bound (ETB)[2] for tasks running on multi-cores is challenged by the contention, also known as inter-task interference, occurring on access to hardware shared resources. Unless otherwise restrained, contention causes the execution time of any one task, hence its ETB, to depend on its co-runners. This has a severe impact on system design and validation, as it conflicts with the incremental development and verification model that industry pursues to contain qualification costs and development risks. This industrial goal is solved by allowing individual subsystems to be developed in parallel, qualified in isolation, and incrementally integrated, with great reduction in the risk of functional regression at system level. In the timing domain, incremental integration and qualification postulate *composability* in the timing behavior of individual parts, whereby the ETB derived for a task determined in isolation, should not change on composition with other tasks.
Two main approaches have been followed so far to deal with contention for multi-core on-chip resources, which place at the opposite ends of a conceptual spectrum of solutions. On the one end, some authors propose computing ETBs so that they upper bound the effect of any possible inter-task interference a task may suffer on access to hardware shared resources. ETBs computed this way are *fully time composable* and enable incremental integration and qualification, but at the cost of pessimism that may cause untenable over-provisioning, as the timing behavior occurring in operation may fall much below the level determined considering the worst-case interference possible in theory [70, 66, 37]. On the opposite end, other authors [22] propose – currently only for research platforms – to determine ETBs simultaneously for multiple tasks in specific configurations. Those ETBs are *non-time composable*, as they only hold valid for the tasks being analyzed and

[2]Due to the lack of definitive Worst-Case Execution Time (WCET) estimation methods for COTS multi-cores, we use the term "execution time bound" (ETB) instead of WCET.

for their specific configuration. If any such parameter changes, all ETBs become invalid and the entire analysis has to be repeated.
In this study we tackle resource contention in multi-cores by proposing the new concepts of **resource usage signature** (*RUs* or $\mathcal{S}$) **and template** (*RUl* or $\mathcal{L}$). A task τ is exposed to contention in the access to the hardware shared resources because co-runner tasks can interfere τ's execution. The usage of the hardware shared resources made by co-runner tasks is referred to as u. *RUs* and *RUl* aim at making the ETB derived for τ, time composable with respect to u. The tasks' ETBs are derived for a particular set of utilizations $\mathcal{U}$ such that the ETB derived for any $u \in \mathcal{U}$ upper bounds τ's execution time under any workload so long as the co-runners of τ make a resource usage smaller than u. We explain later what "smaller" means and how this can be determined. This abstraction allows deriving time-composable ETBs for individual tasks in isolation for each $u \in \mathcal{U}$, so that the system integrator can safely pull those (interfering) tasks together as long as the resource usage made by their individual set of co-runners is upper-bounded by some u. All that the system integrator has to care in that regard is to characterize the tasks' accesses to hardware shared resources (a low-cost abstraction of the task execution time), ignoring any finer-grained detail of that access behavior. In this section we present an approach to produce ETBs in that manner, using measurement-based timing analysis techniques.
RUs and *RUl* are devised to be agnostic to the particular timing distribution of the resource access requests to be considered. Hence, two tasks generating the same number of accesses to a resource, though with different patterns, have the same signature. The challenge in the proposed method is in determining an effect on the interfered task that upper bounds the interference caused by contending accesses, regardless of the time distribution of those accesses as made by the interfered and the interfering tasks. In this work we make the following main contributions:

1. We develop the novel concepts of *RUs* and *RUl* for the timing analysis of COTS multi-cores and sketch an algebra of operators over *RUs*/*RUl* to enable their practical use.
2. We provide exemplary *RUs* and *RUl* for the cases when requests accessing shared resources incur either fixed or variable response latency.
3. We present a strategy to implement *RUs* and *RUl* for the ARM Juno processor representative for the Telecom use case focusing on the bus and the memory controller as exemplars of on-chip shared resources.

4.6.2.1 Formalization of RUs and RUl

RUs and *RUl* allow analyzing, for the most part in isolation, the timing behavior of tasks, by abstracting the interference that they may suffer due to contention when accessing hardware shared resources on a multi-core caused by co-runner tasks.

4.6.2.1.1 Resource usage signature (*RUs*)

Given an interfered task, τ_A, a *RUs* abstracts its use of the hardware shared resources. Once computed, it will be used for τ_A's multi-core timing analysis instead of τ_A itself.
We describe the use of a hardware shared resource through a set of features, which correspond to quantitative values. A *RUs* for task τ_A, is a vector $\mathcal{S}_A = (a_1, a_2, ..., a_n)$ that contains the list of relevant features that characterize all the hardware shared resources, for the evaluation of contention effects. Since *RUs* are quantitative, the *RUs* of distinct tasks are comparable and can also be combined together to form a joint *RUs*. How to operate and compare multiple *RUs* is explained later.
Consider the reference multi-core architecture shown in Figure 4.15, where the bus and the memory are shared. Further consider two types of accesses to those shared resources, for read and write operations respectively. In this case, *RUs* have at most 4 features: bus reads (n_{rd}^{bus}) and writes (n_{wr}^{bus}); memory reads (n_{rd}^{mem}) and writes (n_{wr}^{mem}). *RUs* are thus defined as $\mathcal{S}_A = (n_{rd}^{bus}, n_{wr}^{bus}, n_{rd}^{mem}, n_{wr}^{mem}) = (a_1, a_2, a_3, a_4)$.
If the bus was the only shared resource, the *RUs* of a task τ_A would be abstracted as a *RUs* with two features: n_{rd}^{bus} and n_{wr}^{bus}. If both types of requests hold the bus for the same duration, the *RUs* would consist of a single feature corresponding to the sum of n_{rd}^{bus} and n_{wr}^{bus}, i.e., $\mathcal{S}_A = (n_{rd}^{bus} + n_{wr}^{bus}) = (a_1 + a_2)$. The addition of $\mathcal{S}_B$ to $\mathcal{S}_A$ is given by $\mathcal{S}_A + \mathcal{S}_B = (a_1 + a_2 + b_1 + b_2)$. For comparison, instead, we say that $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \succsim \mathcal{S}_B$, if the interference by the former is greater than that by the latter: $a_1 + a_2 \geq b_1 + b_2$.
This reasoning easily extends to the more realistic scenario in which the bus holding times are asymmetric; for example, with reads holding the bus longer than writes. In that case, the *RUs* for τ_A could be either single-feature, considering all accesses as "long" accesses (counting writes as reads in the example), or multi-feature (two, in the example), i.e., $\mathcal{S}_A = (a_1, a_2) = (n_{rd}^{bus}, n_{wr}^{bus})$. In the latter formulation, addition and comparison

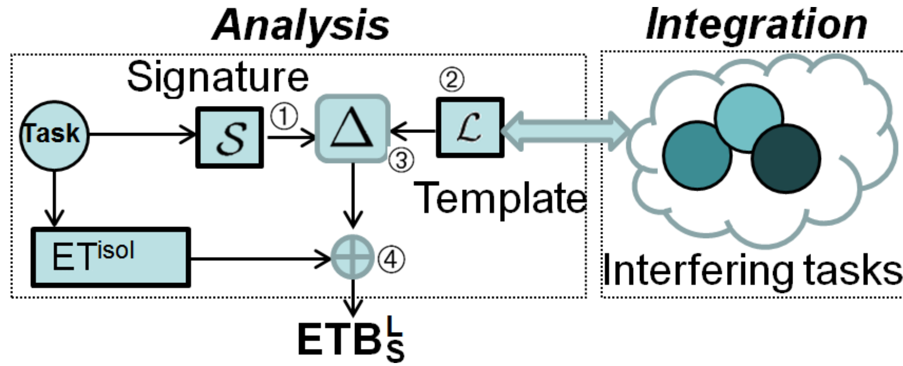


Figure 4.16: Main steps in the *RUs* and *RUl* methodology.

change as follows: addition is defined as vector addition, i.e., $\mathcal{S}_A + \mathcal{S}_B = (a_1 + b_1, a_2 + b_2)$; for comparison, $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \gtrsim \mathcal{S}_B$ if $(a_1 \geq b_1) \wedge (a_2 \geq b_2)$.
Note that it can be the case that neither $\mathcal{S}_A$ dominates $\mathcal{S}_B$ nor $\mathcal{S}_B$ dominates $\mathcal{S}_A$ if, for instance, $(a_1 > b_1) \wedge (a_2 < b_2)$. In that case one should resort either to single-feature *RUs* or to having a new multi-feature *RUs* $\mathcal{S}_C = (max(a_1, b_1), max(a_2, b_2))$ so that $\mathcal{S}_C$ dominates both $\mathcal{S}_A$ and $\mathcal{S}_B$.

4.6.2.1.2 Resource usage template (*RUl*)

RUl have the same form as *RUs*, namely, a vector of features $\mathcal{L}_K = (k_1, k_2, ...k_n)$, but with a different use. *RUs* abstracts tasks according to their use of the shared resources while *RUl* upper bounds the use of the shared resources made by co-runner tasks. A *RUl* is a true upper bound if $\mathcal{L}_K \gtrsim \mathcal{S}_i$, where $\mathcal{S}_i$ is the *RUs* for any task τ_i that can be a co-runner of the task under analysis (i.e. $\mathcal{S}_i$ is dominated by $\mathcal{L}_K$).
Tasks are made time composable against some *RUl* $\mathcal{L}_K$ so that the ETB derived for a given task τ_A and for that *RUl*, denoted ETB_A^K, upper bounds τ_A's execution time inclusive of the interference that the contenders of τ_A, whose *RUs* do not exceed $\mathcal{L}_K$, may cause.
Returning to the example in which the bus is the sole shared resource with all accesses to it incurring the same contention effect: for a $\mathcal{L}_K$ that captures a given number of accesses to the shared bus, we want to determine the highest impact by $\mathcal{L}_K$ on ETB_A, so that ETB_A^K can be regarded as a time-composable bound for τ_A in any workload in which $\mathcal{L}_K \gtrsim \sum_i \mathcal{S}_i$ for all co-runner tasks τ_i of interest.
A maximally time-composable template $\mathcal{L}_{TC}$ exists, which is an upper bound for all potential workloads (i.e. including the worst possible workload). $\mathcal{L}_{TC}$ corresponds to the case in which all accesses from the signature suffer the highest contention from the $N_c - 1$ contending cores[3]. In that case, every access from $\mathcal{S}_A$ contends with $N_c - 1$ other accesses, i.e., $\mathcal{L}_{TC} = (N_c - 1) \times \mathcal{S}_A$. Any $\mathcal{L}_K \gtrsim \mathcal{L}_{TC}$ would produce exactly the same result as $\mathcal{L}_{TC}$, since τ_A cannot be interfered more than the accesses in its signature $\mathcal{S}_A$.

4.6.2.1.3 RUs and RUl through an example

In this section we return to the case in which the bus is the sole shared resource and all accesses to it incur the same contention effect. For now we limit our attention to two cores. The task under analysis, τ_A, runs in one of the two cores. The contending requests from the two cores are arbitrated with the round-robin policy.
Figure 4.16 depicts the process we follow when the proposed approach is applied to this case. First, we obtain the *RUs* of τ_A, denoted $\mathcal{S}_A$. In the example architecture, the *RUs* of tasks using the shared resource is the number of accesses they make, a for τ_A, hence $\mathcal{S}_A = (a)$. Our approach treats contention such that the ETB of τ_A can be derived by upper bounding τ_A's execution time considering the interfering effect that it incurs when its co-runner task makes up to k contending accesses to the shared resource. To this end we define a *RUl* $\mathcal{L}_K$, which is the system integration parameter that defines the inter-task interference to be considered in the determination of τ_A's ETB. The abstraction captured by $\mathcal{L}_K$ with $\mathcal{L}_K = (k)$ is a *RUl*.
Once the $\mathcal{S}_A$ and $\mathcal{L}_K$ are defined, we determine Δ_A^K, the increment to be applied to the execution time that τ_A may incur, to capture the contention effect from $\mathcal{L}_K$. This corresponds to step 3 in Figure 4.16. More precisely, Δ_A^K upper bounds the increment that the execution time of a task τ_A with at most a accesses to

[3]Note that the highest contention for a given access can be upper-bounded in modern embedded processors [70, 66, 37], so such highest contention can be computed in general. If, for instance, one core had priority over the others on the access to hardware shared resources, then the highest contention possible would be infinite and an ETB would not exist. Then such hardware platform would not allow running any critical task in the low-priority cores and our analysis would be valid only for the high-priority cores.

a shared resource may suffer from k contending requests. ETB_A^K (i.e τ_A's ETB determined under the *RUI* $\mathcal{L}_K$) is computed as the summation of ET_A^{isol}, the execution time of τ_A when running in isolation, without contention, and Δ_A^K, the increment that upper bounds the contention effects from any k interfering accesses. This corresponds to step 4 in Figure 4.16. Overall, ETB_A^K is time composable against any co-runner task τ_B with signature $\mathcal{S}_B = (b)$, as long as $\mathcal{L}_K$ dominates the *RUs* of the co-runner, which means that τ_B makes $b \leq k$ contending accesses. We denote this as $tc(ETB_A^K, \tau_B)$, which holds if $b \leq k$.

RUs abstract the distribution of requests over time. Taking into account the exact distribution of requests over time, for instance in the form of requests arrival curves [68], would potentially enable deriving tighter ETB. However, deriving such distributions is complex, as programs normally have multiple paths of execution, each with its own access pattern (distribution). And, paradoxically, considering these particular distributions would decrease timing composability. Instead, our approach only requires the tasks' access count for every individual shared resource, as well as ET_i^{isol} (execution time in isolation) for each individual task τ_i. Notably, both can be obtained with high accuracy by state-of-the-art technology, e.g., [71]. With our approach, the ability to abstract away from the need to know the exact points in time at which requests would be made to shared resources releases the system integrator from the obligation of adopting rigid and inflexible scheduling decisions (which fares poorly with the development unknowns of novel systems) or from the labour-intensive cost of exact analysis.

Our approach requires the user to set the *RUI* to capture the potential co-runner tasks precisely. The spectrum of this capture has two ends. On one extreme we find the time-composable templates, $\mathcal{L}_{TC}$, which represent an upper bound for *RUI*. However, if *RUI* is close to that template, the ETB of tasks might be unnecessarily increased. On the opposite extreme, if *RUI* is too small, it constrains the choice of tasks that may be allowed to run in parallel. A simple solution consists in deriving for each task an ETB under different *RUI*, such that at integration time, the smallest *RUI* that upper bounds the signature of the actual co-runner tasks is used. With this, the residual part of the timing verification at system integration is small and simple. Selecting the proper number of *RUI* represents a trade-off between effort and accuracy: the higher the number of *RUI* the lower the over-estimation of ETB and the greater the analysis time, and vice-versa. Finding appropriate *RUI* is a standard optimization problem that is part of our future work.

In the example considered in this section we have made several simplifications to facilitate understanding: two cores, one single type of access, synchronous accesses (i.e. the core stalls when the access occurs until served) and a single shared resource. In real processors we have different types of accesses to the shared resource (synchronous and asynchronous), each with a distinct access latency. Hence, simply bounding the effect of contention by adding access counts is not enough.

4.6.2.2 RUs & RUI for Measurement-Based Timing Analysis

Next we present one concrete realization of *RUs* and *RUI* for use with measurement-based timing analysis, specifically for a processor with those shared resources in the ARM Juno board.

4.6.2.2.1 Methodology

Our approach uses *micro-benchmarks* [70, 66, 37], a set of single-phase user-level programs with a single execution behavior designed so that all their operations access a given shared resource, e.g. the bus. Micro-benchmarks consist of a main loop whose body includes a substantial number (e.g. 256) of instructions designed to generate a steady stress load on target resources. The fact that the loop body executes repeatedly the same instruction causes the target resource to be continuously accessed. Moreover, placing a high number of identical instructions in the loop body drastically reduces the impact of control instructions (down to 2-4%) [37]. For the architecture in Figure 4.15, a loop body including load instructions that hit in the L2 cache stresses the bus. We consider two types of micro-benchmarks:

Resource stressing benchmarks, *RStB*, place a configurable load on a given shared resource, so that running a task against a *RStB* may represent contention scenarios of interest.

In theory, one could design a *worst-contender* benchmark that generates the maximum contention that a task τ_i can suffer. However, such benchmark would be specific for the task to be interfered and for the target processor [70]. Consider for example, a single shared resource arbitrated by a least-recently-used policy, where the task that accessed the resource last gets the least priority. In that case, the worst-contender benchmark should generate a request in exactly the same cycle as the task of interest, so that every request from that task gets delayed by the contender, and for the next round of arbitration the task has the lowest priority again. The level of control required on the application behavior and the granularity of intervention are too fine-grained and laborious to be used in practice [70].

Resource sensitive benchmarks, *RSeB*, are designed to upper bound the execution time increase suffered by any other task, with a smaller or equal signature, owing to the interference from a given template $\mathcal{L}_K$. Consider a scenario in which bus accesses hold the bus for a constant duration. Further assume that we want to determine Δ_A^K for τ_A, i.e its ETB increment due to a template $\mathcal{L}_K$ with k accesses. Intuitively, one could get an estimate of it by running τ_A several times against a *RStB* that makes k accesses. However, in order to gain confidence in the ETB obtained, the experiment should be repeated with different *alignments* of the *RStB*, so that the interleaving of accesses varies enough and the worst case can be observed in a measurement. In practice, this may require excessive experimentation effort. The need for repeating the experiments with different alignments stems from the uncertainty on the time distribution of accesses, which is hard, if at all possible, to measure and control by timing analysis technology. We can therefore conclude that studying the task under analysis against micro-benchmarks is not viable. Instead, we use micro-benchmarks *to model both the interfered and the (set of) interfering tasks*: *RStB* and *RSeB* are designed to account for bad alignments of requests: *RSeB* is made of instructions that cause accesses to the shared resource and that continuously contend with *RStB* requests.
We define $\Delta_{RSeB}^{RStB} = ET_{RSeB}^{RStB} - ET_{RSeB}^{isol}$, where ET_{RSeB}^{RStB} is the execution time when a given *RSeB* with the same signature as task τ_A runs against a *RStB* implementing a template $\mathcal{L}_K$ with k accesses; and ET_{RSeB}^{isol} the execution time when the *RSeB* runs in isolation. For task τ_A, let $\Delta_A^K = ET_A^K - ET_A^{isol}$ be the execution time increase τ_A suffers when it runs against $\mathcal{L}_K$. *RSeB* and *RStB* are designed so that $\Delta_{RSeB}^{RStB} \geq \Delta_A^K$ holds for any request alignment of τ_A under $\mathcal{L}_K$ contention. To that end, we run the *RSeB* in isolation and then against $N_c - 1$ copies of *RStB* so that all *RSeB*'s accesses to the shared resource suffer high contention, causing a measurable Δ_{RSeB}^{RStB} to emerge. In the next section we show how to derive the number of accesses of the *RSeB* and the *RStB*, based on the number of accesses of the template and signature under consideration.
Δ_{RSeB}^{RStB} is used to compute the ETB estimate for τ_A as follows: $ETB_A^K = ET_A^{isol} + \Delta_{RSeB}^{RStB}$. ETB_A^K is composable with any set of interfering tasks against which τ_A runs in parallel, if their total number of accesses is lower or equal to k. That is, the addition of the signatures of the interfering tasks is dominated by $\mathcal{L}_K$: $(S_i + S_j + ... + S_l) \precsim \mathcal{L}_K$. Interestingly, given a task τ_B whose signature is dominated by τ_A, i.e. $S_B \precsim S_A$, the obtained Δ_{RSeB}^{RStB} for τ_A can be used to upper bound τ_B's execution time: $ETB_B^K = ET_B^{isol} + \Delta_{RSeB}^{RStB}$.
Overall, *RUs* and *RUl* provide powerful abstractions for the interfered and the interfering tasks, which simplify the integration of multiple tasks by combining their signatures.

4.6.2.2.2 The case of a ARM big.LITTLE (Juno) architecture

Our reference multi-core architecture comprises $N_c = 2$ and $N_c = 4$ symmetric cores in each of the 2 multicores (ARM Cortex-A57 and ARM Cortex-A53 respectively), see Figure 4.15, each equipped with private instruction cache (IC) and data cache (DC). Some of the hardware characteristics described next are derived from the manuals while others are inferred from limited information in those manuals. Those that may affect the methodology and cannot be obtained unambiguously from the processor manual will be studied empirically as part of BSC work in T4.1.

Relevant hardware characteristics
ARM Cortex-A57 and ARM Cortex-A53 multi-cores differ. However, the ARM Cortex-A57 is more powerful than the Cortex-A53 one, so the features of the Cortex-A57 are a superset of those in the Cortex-A53. Hence, we refer to the particular hardware characteristics of the Cortex-A57 multi-core given that they already include those of the Cortex-A53 one. IC and DC can be used in many different ways including write-allocate, write-no-allocate, write-through, etc. This has a direct influence on the number of accesses issued to the L2 cache, and so on the contention suffered. Analogously, speculation on branches and prefetch operations may issue further requests to the shared resources that may also suffer contention (or increase the contention suffered by non-speculative requests). The L2 cache can process up to 2 requests simultaneously if they access different tag/data Random Access Memory (RAM) arrays. L2 is accessed through an Advanced Microcontroller Bus Architecture (AMBA) AXI interface which, in principle, may receive requests from the different cores and process them in parallel. Therefore, it is the L2 cache the resource serializing requests, not the AMBA AXI interface.
It is, therefore, unclear what type of network is implemented. We will assume that core-to-L2 networks are also implemented using AMBA buses, as it is the case for the L2-to-memory network, since this is the worst case because requests are fully serialized and so contention is the highest. As the work in the project progresses we will confirm or reject our assumption. However, if it is rejected, our methodology should be flexible enough to adapt to any common network available in small multi-cores such as the 4-core ARM Cortex-A57 and Cortex-A53.

Bus

interfered	st			l2h			l2m		
interfering	l2h	l2m	st	l2h	l2m	st	l2h	l2m	st
Impact	7	2	2	7	2	2	2x7=14	2x2=4	2x2=4

Figure 4.17: Hypothetical impact (in cycles) from/to the different access types to the bus. $l2h$, $l2m$ and st refer to L2 load hits, L2 load misses and stores respectively.

For the sake of this discussion we assume that our target processor implements round-robin bus arbitration so that if, in a given round, core $c_i, i \in \{1, .., N_c\}$ is granted access to the bus, the priority ordering in the next round is: $c_{i+1}, c_{i+2}, ..., c_{N_c}, c_1, c_2, ..., c_i$. A lower priority core can use the bus when all higher priority cores do not use it. Due to the incomplete documentation available for the Juno processor (and for the previously targeted SnapDragon 810), it turns out to be almost impossible determining with enough confidence the arbitration policy. In any case, as part of the work in T4.1, BSC has developed a method that, among others, is able to identify whether the arbitration policy in shared resources is round-robin or FIFO.
The *bus access jitter* that a task incurs on access to the bus, depends not only on the number of co-runners but also on the way their requests interleave. The worst contention situation happens when a task τ_B assigned to core c_i requests the bus in a given round of arbitration, simultaneously with tasks in all other cores and the previous round was assigned to c_i.

L2 cache
A shared L2 cache poses many difficulties to upper-bound the impact of contenders since the amount of contention depends on many fine-grain parameters such as: amount of data accessed, time elapsed between accesses to the same cache line, frequency of access, cache sets accessed, etc. Therefore, shared L2 caches do not fit well within the concept of signatures and templates, so alternative approaches are needed to quantify the contention due to shared L2 caches. Such approach is later detailed in Section 4.6.3.

Memory controller
The L2 sends a request to a memory controller on every L2 miss. Requests are stored in a FIFO request queue, with one entry per core. Whether only one memory controller is in place or each multi-core has a separate one needs to be investigated. For the sake of this discussion we will assume that each multi-core uses one different memory controller. In future enhancements of our methodology we will relax this constraint.

4.6.2.2.3 Bus

The AMBA bus handles three distinct request types, which differ in the contention they induce and suffer. Stores (*st*) regardless of whether they hit or miss on the L2, are served immediately by the L2 and hold the bus for few cycles (e.g. 2 cycles). L2 load hits (*l2h*) hold the bus for few more cycles (e.g., 7 cycles) because the bus is retained while retrieving the data from L2. L2 load misses (*l2m*) release the bus once the request reaches L2, and perform a new arbitration whenever the L2 responds to the miss, holding the bus for as many cycles as L2 store accesses (e.g., 2 cycles) in each arbitration. Figure 4.17 shows the contention suffered by a source (interfered) request by another (interfering) request for all request types assuming some hypothetical latencies of 2 cycles for short L2 stores and L2 load misses, and 7 cycles for L2 load hits. $l2h$ generate the highest contention and $l2m$ are the most affected since they suffer two rounds of arbitration: $l2m$ can therefore be interfered twice by two concurrent contending requests, one round of arbitration per each such request.
Our approach based on *RUs* and *RUI* does not require knowing the exact time of request issue, but whether they have asymmetric timing behavior in the impact they suffer and they cause to other request types so that *RStB* and *RSeB* can be designed with the appropriate request types. The *RStB* and *RSeB* for the bus are called *BStB* and *BSeB*:

BSeB (abstracting interfered task bus usage). The signature of a task τ_A running in this architecture may take different forms, with different levels of tightness and experimentation effort. The canonical signature for the bus contains the number of accesses of each type made by the task. That is: $\mathcal{S}_A^{bus} = (a_{st}, a_{l2h}, a_{l2m})$. This can be simplified by realizing that $l2h$ and st access the bus once whereas $l2m$ do it twice. Moreover, the delay suffered by an access does not vary whether the access was generated by a $l2h$, st or $l2m$. Hence, signatures have the form: $\mathcal{S}_A^{bus} = (a_{st} + a_{l2h} + 2 \times a_{l2m})$.
BSeB can be implemented with either $l2h$ or st. Conversely, $l2m$ are not appropriate as it is not possible to place high pressure on the bus with $l2m$ since they miss in cache and take long to be served from memory, leaving the bus idle in the meantime. Instead, $l2h$ and st can place very high pressure on the bus. Our approach considers *BSeB* to only have st operations.

BStB (abstracting interfering task(s) bus usage). Templates can be mono- ($\mathcal{L}_{1D}$) or bi-dimensional ($\mathcal{L}_{2D}$).

$\mathcal{L}_{2D}$. Accesses of type st and $l2h$ generate different impact on the bus (recall that $l2m$ are equated to 2 st). In particular, $l2h$ produces the highest impact and st the lowest. This allows generating bi-dimensional templates: $\mathcal{L}_{2D} = (k_{l2h}, k_{2\times l2m+st})$, whereby *BStB*s comprises load L2 hit accesses and store accesses to generate each respective type of interference.

$\mathcal{L}_{1D}$ templates comprise only $l2h$, which generate the highest interference. A given $\mathcal{L}_{1D} = (k_{l2h})$ with k $l2h$ accesses upper bounds the impact that one or several tasks, whose bus access count is lesser or equal to k, can generate on any other interfered task. $\mathcal{L}_{1D}$ are easier to generate and simplify experimentation, but they increase the pessimism of ETBs, since st are considered to generate the same impact as $l2h$.

Putting it all together. Deriving the access count for *BSeB* and *BStB* varies for $\mathcal{L}_{1D}$ or $\mathcal{L}_{2D}$ as we show next.

$\mathcal{S}_A - \mathcal{L}_{1D}$. Let a and k be the number of accesses in the signature $\mathcal{S}_A$ and the template $\mathcal{L}_K$ respectively. Running *BSeB* and *BStB* concurrently, we derive an upper bound to the increase in execution time (Δ^{BStB}_{BSeB}) that k accesses of the template can have on the a accesses of the signature. If $k \geq (N_c - 1) \times a$ then each request of $\mathcal{S}_A$ suffers the impact of $N_c - 1$ contenting requests. If this is not the case, only $\lceil k/(N_c - 1)\rceil$ requests from $\mathcal{S}_A$ suffer the impact of $N_c - 1$ contenting requests.
The number of request accesses generated by the *BSeB* is given by $N = min(a, \lceil k/(N_c - 1)\rceil)$. By running this *BSeB* against $N_c - 1$ *BStB* copies, each having a number of accesses largely above N, we derive an upper bound to the impact that $\mathcal{L}_K$ has on $\mathcal{S}_A$. The impact that a task can suffer due to a template $\mathcal{L}_K$ with k $l2h$ is upper bounded as: $\Delta^{BStB}_{BSeB} = ET^{BStB}_{BSeB} - ET^{isol}_{BSeB}$. The ETB derived for a given task τ_A and template $\mathcal{L}_K$ is: $ETB^K_A = ET^{isol}_A + \Delta^{BStB}_{BSeB}$.

$\mathcal{S}_A - \mathcal{L}_{2D}$. In the case of 2-dimensional signatures and templates we account for the fact that requests sent by the interfered task, τ_A, suffer different interference by the $l2h$ and $l2m/st$ sent by the interfering tasks, abstracted in $\mathcal{L}_{2D}$. In this approach we pair up every request in τ_A with $N_c - 1$ requests in $\mathcal{L}_{2D}$ causing the highest interference ($l2h$) on the former. If the number of those requests in $\mathcal{L}_{2D}$ is exhausted, we pair up τ_A requests with those in $\mathcal{L}_{2D}$ causing the second worst interference (st).
We generate two *BSeB* and *BStB* pairs to capture the impact that accesses in $\mathcal{S}_A$ suffer from $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ so that:

$$\Delta^{BStB}_{BSeB} = \left(\Delta^{BStB_l}_{BSeB_1} + \Delta^{BStB_s}_{BSeB_2}\right) \tag{4.27}$$

$BSeB_1/BStB_l$ and $BSeB_2/BStB_s$ capture the interference on τ_A's accesses caused by the $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ respectively. $BSeB_1$ and $BSeB_2$ have different number of st operations, N_1 and N_2. $BStB_l$ comprises $l2h$ operations whereas $BStB_s$ comprises st operations.
Let us assume for example $a = 30$, $k_{l2h} = 60$, and $k_{st} = 80$. In this case, $BSeB_1$ has $N_1 = min(30, \lceil 60/3\rceil) = 20$ st, which we pair up with 20 accesses in $\mathcal{S}_A$; and $BSeB_2$ has the rest of accesses in $\mathcal{S}_A$, $N_2 = 30 - 20 = 10$ st, which we pair up with 3×10 requests out of the 80 accesses in k_{st}. The remaining 50 st in k_{st} are not paired since they will not cause further impact on $\mathcal{S}_A$. Overall, an upper bound to the impact that an application can suffer due to $\mathcal{L}_{2D}$ is given by:

$$ETB^K_A = ET^{isol}_A + \left(\Delta^{BStB_l}_{BSeB_1} + \Delta^{BStB_s}_{BSeB_2}\right) \tag{4.28}$$

For the memory controller we follow the same principles as for the bus, with the particularity that the impact from/to the read/write request types is homogeneous. Hence we only need $\mathcal{L}_{1D}$ templates. The *RStB* and *RSeB* for the memory are called *MStB* and *MSeB*.

4.6.2.2.4 Multi-resource signatures

In the presence of several shared resources, the signatures and templates must cover the features to upper bound contention in each of them. For the reference architecture considered in this work, signatures and templates are as follows: $S^{bus+mc}_A = (a_{st} + a_{l2h} + 2a_{l2m}, a_{mem})$ and $\mathcal{L}^{bus+mc}_K = (k_{st} + 2k_{l2m}, k_{l2h}, k_{mem})$.
It is possible that a task suffers contention in several shared resources simultaneously, so that the impact of the contention does not accumulate but rather overlaps. However, determining trustworthy bounds to the degree of overlap in the contention suffered on requests to different resources is hard. Signatures and templates are intentionally made agnostic to the distribution of requests over time. As we focus on the number of requests to each resource rather than on their timing, it is difficult to determine how contending requests overlap. Our current approach assumes no overlap in contention, which in a time-anomaly free processor design [57] is a safe assumption on the maximum impact of contention. It needs to be investigated to what extent the ARM Juno processor is subject to timing anomalies and to what extent those timing anomalies break our assumptions.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Isolation	A_1^∞	B_1^∞	C_1^∞	D_1^∞	A_2^3			A_3^0	D_2^1	C_2^2	B_2^3		C_3^1
Task A h/m	miss	miss	miss	miss	miss			hit	hit	miss	miss		hit
Multicore	A_1^∞	B_1^∞	C_1^∞	D_1^∞	A_2^3	X_1^∞	Y_1^∞	A_3^2	D_2^3	C_2^4	B_2^5	Y_1^4	C_3^2
Task A h/m	miss	miss	miss	miss	miss			MISS	MISS	miss	miss		MISS

Table 4.8: Illustrative example of how stack distance helps capturing misses due to contention

Overall, in the presence of a template for the bus $\mathcal{L}^{bus}$ and the memory $\mathcal{L}^{mc}$ (a.k.a. $\mathcal{L}^{bus+mc}$), a task is assumed to suffer the sum of the contention generated by both templates:
$ETB_A^{\mathcal{L}_K^{bus}+\mathcal{L}_K^{mc}} = ET_A^{isol} + \Delta_{BSeB}^{BStB} + \Delta_{MSeB}^{MStB}$

4.6.3 OnChip Cache Sharing

Given two applications, A and B, executed concurrently in a processor with a shared L2 cache, how the latter impacts the former due to shared cache interference depends on several parameters. These include:

1. The L2 cache access frequency that, in turn, is determined by the miss rate in DL1 and IL1. In the case of a write-through DL1, the number of stores also affects the access frequency to the L2, since they are forwarded to L2 regardless of whether they hit or miss in DL1. Note that, since DL1 and IL1 are private to each core, their behavior is essentially unaffected by contender tasks.
2. The L2 cache miss frequency, which serves as a proxy metric of the cache requirements of the applications. In general, the more an application misses in L2, the higher its L2 cache requirements are.
3. How applications' L2 accesses interleave, that is, the particular order in which accesses occur to the L2 which affects A an B behavior in cache.
4. The memory mapping, aka the memory addresses where their code/data are mapped. With modulo cache placement this determines the cache sets that A and B use.

Capturing all these low-level factors is challenging due to the high frequency at which they occur. In the extreme case, one would need the sequence of addresses accessed by each application, as well as the cycle in which L2 accesses occur. However, such sequence is not only time consuming to collect, but also hard to reproduce automatically with surrogate applications. Mimicking their access times of individual accesses further difficults the creation of surrogate applications.
A more workable approach requires better balancing the accuracy it can be obtained when mimicking the cache events of a given task, and the information to be extracted from the application to do so. In the remaining of this section, we propose such an approach based on the *stack distance*.

4.6.3.1 Introduction to Stack Distance

Common eviction cache policies such as least-recently used (LRU) have the stack property [61]. Each set in a cache can be seen as a LRU stack, where lines are sorted by their last access cycle: the first line of the LRU stack is the most recently used (MRU) line, whereas the last line is the LRU. The position of a line in the LRU stack defines its stack distance. Further, those accesses with a stack distance smaller than or equal to the number of cache ways (w) result in a hit and vice versa, as shown in Equation 4.29 and Equation 4.30, where sd_i is the number of accesses with stack distance i.

$$hits = \sum_{i=0}^{w-1} sd_i \quad (4.29)$$

$$misses = \sum_{i=w}^{+\infty} sd_i \quad (4.30)$$

Overall, in the context of LRU, the stack distance of an access $@A_k$ is the number of unique (i.e. non-repeated) addresses mapped to the same set where $@A_k$ is mapped and that are accessed between $@A_k$ and the previous access to it, $@A_{k-1}$. Note that stack distance is similar to the concept of reuse distance, though the latter does not break down accesses per set.
While we focus on LRU, a similar analysis could be conducted in other replacement policies such as, for instance, First-In First-Out (FIFO). In the case of FIFO, the stack distance is defined w.r.t. the last access fetching an address rather than the absolute last access to that address.

4.6.3.2 Stack Distance as Proxy for Cache Contention

In our baseline platform, hit accesses in IL1 and DL1 are not affected by cache contention since both IL1 and DL1 are private and non-inclusive, thus meaning that a L2 eviction cannot create a cascade effect in L1 caches. With respect to the L2, L2 misses are not affected either by contention. Since we focus on independent applications that do not share data or instructions, if an access of a given application results in a miss when the application runs in isolation, it will also result in a miss in any workload in which the application runs. Hence, only hit accesses can change their behavior. In particular hits to the L2 can become misses due to evictions of contender tasks.
Building on stack distance, we can formulate the case where an access (e.g. $@A_k$) that hits in L2 when the application runs in isolation, becomes a miss when the application runs in multicore mode as follows. The stack distance of $@A_k$ when the application runs in isolation is smaller than w and in multicore it becomes equal or larger than w, see Equation 4.31. $sd_{isol}(@X)$ defines the stack distance of an access $@X$ when the application runs in isolation, and $sd_{muc}(@X)$ when the application runs in multicore under a given workload.

$$sd_{isol}(@A_k) \leq w \quad and \quad sd_{muc}(@A_k) > w \tag{4.31}$$

The increase in the stack distance occurs due to the accesses performed by contender tasks to the same set where $@A_k$ is mapped to. This is better illustrated with the simple example presented in Table 4.8. The first row corresponds to the sequence of accesses of a given task T_1 to an arbitrary set. The second row shows the sequence of accesses to the same set when task T_1 runs simultaneously with another task T_2. Note that application T_1 accesses addresses $@A$, $@B$, $@C$, and $@D$ whereas application T_2 accesses addresses $@X$, $@Y$, $@Z$. For the sake of simplicity all addresses are assumed to belong to different cache lines. For each address the super index shows its stack distance. For each address the subindex describes its stack index and the superindex the number of times that access has repeated. Assuming a 2-way cache we observe that T_2 accesses increase T_1 accesses' stack distance making that some of them become miss, marked as 'MISS' in the table.

4.6.3.3 sdki

From the previous analysis it follows that mimicking stack distances of the accesses of a given application is critically important to accurately reproduce the impact that such application can have on other applications. While this is not necessarily the only parameter that needs to be mimicked, as shown in our evaluation section it is enough to achieve high accuracy in the applications evaluated.
In our approach we define stack distance k per thousand (kilo) of instructions or $sdki_k$ as the number of accesses with stack distance k every 1,000 executed instructions. Note that the normalization to thousands of accesses is just done since stack distance values per instruction are naturally very low (below 1). For each application we collect the stack distance vector (SDV) that has w entries as shown in Equation 4.32.

$$SDV = [sdki_0, sdki_1, sdki_2, ..., sdki_w] \tag{4.32}$$

Note that under $sdki_w$ we count all values with stack distance $\geq w$, since all them result in misses.
Interestingly, with the $sdki$ formulation we can derive other well-known cache parameters, such as accesses per kilo instruction and misses per kilo instruction, $apki$ and $mpki$ respectively.

$$apki = \sum_{i=0}^{w} sdki_i \tag{4.33}$$

$$mpki = sdki_w \tag{4.34}$$

While the $sdki$ provides a powerful abstraction of an application's cache utilization, it misses some parameters that also affect cache contention. *Burstiness* covers whether cache accesses (and/or misses) occur in bursts during execution or, on the contrary, distribute quite evenly over time. *Set distribution* covers whether accesses/misses spread across few or many cache sets. While both can be taken into account with our approach, with the aim of maintaining the SurApp generator simple, in this work we assume homogeneity in both parameters and provide some guidelines on how both could be factored in when generating SurApp. Our results show that the accuracy achieved is very high, thus proving the effectiveness of our approach without accounting for those parameters. However, as part of our future work we plan to identify applications particularly sensitive to those parameters and extend our generation of SurApps accordingly.

Parameter	Definition
$il1_size$	Number of instructions that fit in L1
w	Number of ways
w_size	way size in bytes
$inst$	Number of desired executed instructions
SDV	Vector that represents the proportion of the accesses that has certain SD
$otype$	Type of accesses to memory, this parameter can be "LOAD" or "STORE"
$Nnops$	Number of nop instructions to inject in each burst
str	Stride among accesses to $dvec[]$

Table 4.9: Parameters used by SurAppGen

4.6.3.4 Obtaining sdki

The SurApp approach requires deriving $sdki_k$ values from target applications when run in isolation. To that end we can make use of standard tracing facilities existing on many architectures. For instance, the LEON processor family allows collecting instruction and data addresses, opcode and timestamp of all instructions. This information is dumped via the debug interface (DSU). Other processors provide similar support, e.g. the Nexus Interface Nexus Interface [2] for NXP (formerly Freescale) or the Coresight [1] for ARM, which are increasingly being considered for measurement-based and static timing analysis solutions [32, 33].
Overall, obtaining the information needed to compute $sdki_k$ values is doable in many architectures using existing tracing capabilities. When those capabilities do not exist, emulation or simulation tools can be used instead.

4.6.3.5 Surrogate Cache Application Generator

The Surrogate Cache Application Generator (SurAppGen) generates applications in C code. When those applications are compiled and executed they mimic the cache behavior of the target applications described with the input parameters passed to the SurAppGen. In this section we describe the main structure of a SurApp as well as the parameters that the SurAppGen uses to control its behavior.
The main data structure of the SurApp is a vector ($dvec[]$) whose size is given by $dvec_size = (w + 1) \times wsize$, where w is the number of ways and $wsize$ the size of a cache way. For instance, for a $32KB$ 4-way cache $dvec_size = (4 + 1) \times 8KB = 40KB$. SurApp accesses this vector appropriately so that those accesses match the $sdki$ described in SDV.
A SurApp contains two well differentiated phases: an initialization phase and an execution phase. While during the initialization phase some internal parameters are updated based on the input parameters, most of the initializations are carried out by the SurAppGen reducing as much as possible the code in the initialization of the SurApp. For instance $dvec[]$ size is initialized before the SurApp is compiled, thus avoiding the execution of memory allocation operations.
For accessing cache, the SurApp can be configured to use either loads or stores. SurApp generates activity in the L2 via data accesses, which are easier to control explicitly than instruction accesses. On the other hand, its code is specifically designed to fit in the IL1 cache so that it does not create uncontrolled interferences.

4.6.3.5.1 Initialization and Pre-Initialization phases

The input parameter $otype$ determines whether the type of accesses used by the SurApp are loads or stores, which in fact, affects the initialization phase.
When $otype = load$, we follow a pointer chasing approach in which in each entry of the vector we store the address of the next element to access, see Figure 4.18. This allows traversing the vector during the execution phase without additional control instructions to compute the address of the next element to access. The elements of the vector that are accessed are str bytes apart. By playing with the $dvec_size$ and str we can force all accesses to hit/miss in a desired cache level and be accessed with specific stack distances in the range [0,w].
When $otype = store$, we cannot follow the same pointer-chaising approach. In this case, the vector is just initialized to zero leaving the computation of the addresses of the elements to access to the execution phase.

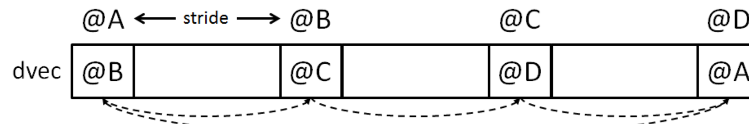


Figure 4.18: Block Diagram of the pointer chaising approach followed for loads.

4.6.3.5.2 Execution Phase

The execution phase works with different sub-phases for each stack distance, meaning that different code is generated for each stack distance. These code blocks are defined by the `block-operation()` function that is invoked once per stack distance as shown Algorithm 1. As input parameters, `block-operation()` has the stack distance sd, the number of nops[4] $Nnops$ to generate and the total number of instructions to generate $inst$. Note that $sdki$ is an input parameter to the SurAppGen, whereas the number of instructions to execute for each stack distance value is computed by SurAppGen.

Algorithm 1 Baseline structure of the generated SurApp.

```
1: procedure EXECUTE
2:     for (i = 0; i <= w; i++; ) do
3:         block-operation(i, iter, Nnops);
4:     end for
5: end procedure
```

The function `block-operation()`, see Algorithm 2, executes memory operations with a stack distance i and also core operations – in particular nop operations. The number of nop operations $Nnops$ to execute and the number of iterations of the main loop in `block-operation()` is computed by the SurAppGen so that the desired $sdki$ is achieved: $(iter, Nnops) = compute_params(sdki, inst)$, where $inst$ defines the number of operations to execute. The `reset()` function properly resets the access pointer to $dvec[]$ as needed.
The main body of `block-operation()` invokes memory operations and nop operations (line 6 and 7). For the former, the $memory_operation(addr)$ is invoked and for the latter, $nop_operation(Nnops)$ is invoked.

Algorithm 2 Code executed for each stack distance

```
1: procedure BLOCK_OPERATION(sd,iter, Nnops)
2:     reset(addr);
3:     for (i = 0; i < iter; i++; ) do
4:         for (j = 0; j <= sd; j++; ) do
5:             for (k = 0; k < w_size; k++; ) do
6:                 memory_operation(addr);
7:                 nop_operation(Nnops);
8:             end for
9:         end for
10:    end for
11: end procedure
```

The code of $memory_operation(addr)$ varies for loads and stores, while the functionality is roughly the same. In particular, the first $(sd + 1) \times w_size$ bytes of $dvec[]$ are accessed so that after the first iteration all accesses have the desired stack distance. Note that in the first invocation of $memory_operation(addr)$ all accesses to $dvec[]$ have $sd = w$, which is accounted by the SurAppGen when computing the number of memory operations with $sd = w$.
When the memory operations to be generate are loads, the body of $memory_operation(addr)$, as shown in Algorithm 3, basically loads in a variable the current position in $dvec[]$. The contents of this variable is, in fact, the address of the next address to be accessed, since it $dvec[]$ has been initialized following a pointer chaising approach by the SurAppGen.

[4] A nop (not operation) is an instruction type existing in most architectures, which has no functional effect and executes typically in a single cycle. If nops do not exist in a given architecture, alternatives can be used such as performing simple

Algorithm 3 Algorithm for memory operations with loads

1: **procedure** MEMORY-OPERATION(addr)
2: $addr = dvec[addr];$
3: **end procedure**

When the type of memory operations to be generated is store, the body of $memory_operation(addr)$ has some differences w.r.t. the case of loads. In particular, after storing a value in the target address some control operations are needed to compute the next address to access, see Algorithm 4.

Algorithm 4 Algorithm for memory operations with stores

1: **procedure** MEMORY_OPERATION(addr)
2: $dvec[addr] = 0;$
3: $addr+ = stride;$
4: **end procedure**

Finally, the code of the `core-operation()` function is given in Algorithm 5. It basically executes $Nnops$ nop operations.

Algorithm 5 Algorithm for core operations

1: **procedure** CORE_OPERATION(Nnops)
2: **for** $(i = 0; i < Nnops; i + +;)$ **do**
3: nop;
4: **end for**
5: **end procedure**

Note that for sake of clarity in the explanations we have encapsulated all code in functions, which can be however inlined to reduce the overhead of control operations (i.e. `call` and `return`).
In the same line, some loops can be simply removed unrolling the body as many times as needed – keeping in mind the restriction that the SurApp code must fit in IL1. If loops are used, the number of control (core) operations they generate is factored in by the SurAppGen to achieve the desired $sdki$ defined in the SDV.
It is also the case that we generate the code in C since this improves portability, though it requires certain control on compiler flags so the assembly code actually reaches the desired goal. Alternatively, SurApp can be generated directly in assembly code, which needs however the use of specific assembler instructions for the target architecture.

4.7 Timing Integrity for Network: SAFURE Solution

4.7.1 Timing Integrity for Network: Challenges & Existing Solutions

Ethernet has long been identified as the replacement for Flexray as the communication backbone in the automotive domain. On the one hand, Ethernet can provide the bandwidth and flexibilty required for modern high-througput applications which Flexray is lacking. On the other hand, Ethernet was originally not developed with any real-time constrains in mind and hence lacks any inherent real-time properties. Embedded applications, however, are often required to comply with strict real-time constraints. This becomes especially challenging when certified high critical data and low critical data have to share a medium, such as the communication network. With a limited number of priorites, standard Ethernet offers only a limited inherint isolation. However, for these mixed-critical networks, sufficient isolation between data streams of different criticalies has to be ensured.
In the context of SAFURE, we focused on Time-Sensitive Networking, a set of upcoming Ethernet extensions aimed at providing methods to integrate aspects of saftey and timinig criticality into Ethernet. Older standards, such as switched Ethernet (IEEE802.1Q)[86, 85, 31], Ethernet Audio-Video Bridging (AVB) (IEEE802.1Qav)[49,

arithmetic operations and storing results in innocuous destinations such as the always-zero register or an unused one.

31] or Avionics Full-Duplex Switched Ethernet (AFDX) [41, 4] have been covered by several analyses, see section 4.3 in D3.1. Additionally to switch-level frame transmission schemes, network management techniques were evaluted. Modern embedded systems need to be capable to adapt to the ever changing environment, be it a change in the system objective, or simply to adapt to unforeseen events. One possible candidate for such a real-time supporting network managment is the *Software Defined Networking* (SDN).
In the following, we present the work performed on real-time Ethernet standards, as well as SDN, in the context of SAFURE.
TSN is utilising its underlying Ethernet structure. Therefore, we also provide analysis methods for standard switched Ethernet (IEEE802.1Q) as a reference. All models and analyses are build on the compositional performance analysis (CPA) approach.

4.7.2 Compositional Performance Analysis for Ethernet

In this section we give a brief overview of the compositional performance analsis (CPA), which is used to derive the worst-case timing behaviour of the Ethernet switches and, therefore, end-to-end delays of frames sent through a network. A more detailled description of CPA can be found in [43].
The CPA model consists of three elemental parts: *resources, tasks* and *event models*. Resources provide an abstract service according to a specified scheduling policy which is consumed by tasks being executed on these resources. The tasks are activated according to their input event models and produce an output event model, which describes their termination behaviour. In contrast to a single event trace, event models only consider the best- and worst-case behavior, which is bounded by the set of arrival functions $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, bounding the minimum and maximum number of task activations within the given, half-open, time interval $[t, t + \Delta t]$. There also exist the pseudo-inverse functions $\delta^-(n)$ and $\delta^+(n)$, which yield the minimum and maximum (time) distance between any n consecute events.
Tasks can form a linked path, if the output event model of one task is utilised as the input event model for another.
For Ethernet, CPA resources represent switch output ports which provide transmission time for frames, whose frame transmissions are modelled as tasks. An Ethernet stream is defined as a sequence of frames being transmitted via a defined path, and is hence modelled as chain of tasks as described above. The frame transmission duration, i.e. the amount of consumed service, depends on the the port speed (r_{TX})of the output port and length of the frame, determined by its payload (p_i):

$$C_i^{-/+} = \left(42\text{byte} + \max\{42\text{byte}, p_i^{-/+}\}\right) / r_{TX}. \tag{4.35}$$

The longest transmission time (C_i^+) is needed, if the largest payload (p_i^+) is transmitted.
We need to add the protocol overhead for our Ethernet frames of 42 byte for the Ethernet frame, as well as UDP/IP packaging. Additionally, we consider the minimum valid Ethernet frame length and add padding if necessary.
The analysis in CPA [43] is performed in two steps: 1) A local analysis is performed which derives the best-and worst case frame latencies on each indivial switch output port. 2) A global propagation of the resulting output event models from the first step. Event models are propagated by a loop, which terminates if all event models have converged and become stable, otherwise, the cycle is repeated.
In order to determine the worst-case end-to-end latencies for the path of Ethernet frames, the local worst-case transmission latencies on each switch port are accumulated.
In the following sections, we concentrate on the local analysis of switch ports, since they are dependent on the scheduling policy of the respective switch.

4.7.3 Switched Ethernet

The goal of the local analysis is to determine the worst-case transmission latency of a frame at a particular switch. The transmission latency of a frame is the duration from when the frame has been received at the switch input port until it has been fully transmitted via an output port. Within each switch the transmission of a frame is affected by several stages: contention at the input port, forwarding delay in the switching fabric, queueing delay at the output port buffers, and transmission delay on the link. The first two are impelementation dependent and typically in the order of a few clock cycles, i.e. they are significantly lower than actual transmission latencies. The transmission delay on the wire can be considered constant, e.g. a 10m copper wire causes a delay of roughly 50 ns. Therefore, we assume that the transmission latency of a frame consists only of the output port's queueing delay, as the other delays only have negligible impact on the transmission latency or can be added as

constants to the other delays. The output port queueing delay accounts for all delays caused by an output port scheduler (including the various delays and shaper effects of Time Sensitive Networking (TSN) in the following sections).
Ethernet frames are generally transmitted non-preemptivley. Therefore, the worst-case transmisssion latency R_i^+ of a frame for stream i can be computed via its worst-case queueing delay within its corresponding output port. The worst case queueing delay of the q-th frame is computed via the busy period:

Definition 1. *The worst-case queueing delay $w_i(q, a_i^q)$ of frame q of stream i arriving at a_i^q at the output port, is the the time interval from the beginning of the level-i busy period [27] (the arrival of the first frame within i) until the transmission of frame q starts.*

Note that a_i^q is measured relatively to the start of the busy period and that $w_i(q, a_i^q)$ depends on the arrival time a_i^q of the q-th frame, which will be addressed below.
Frames of different traffic streams interfere at the switch output ports. There, different blocking effects can occur:
Lower-priority blocking: If a lower priority frame can block a higher priority frame. This can happen at most once, if the lower priorty frame started transmissing just before the higher priority one arrives:

$$I_i^{LPB} = \max_{j \in lp(i)} \{C_j^+\} \tag{4.36}$$

where lp(i) is the set of all traffic streams with a priority lower than that of stream i.
Higher-priority blocking: In any time interval Δt, a frame of stream i can be blocked by all higher-priority frames arriving before this frame can start its transmission:

$$I_i^{HPB}(\Delta t) = \sum_{j \in hp(i)} \eta_j^+(\Delta t) C_j^+ \tag{4.37}$$

where hp(i) is the set of all traffic streams with a priority higher than that of stream i.
Same-priority blocking: A frame of stream i suffers also from blocking of frames with the same priority FIFO queue. The q-th arrival of a frame of stream i, arriving at a_i^q , is queued behind $q-1$ predecessors of its own stream and for all frames of other same-priority streams, which have been queued previous to its arrival.

$$I_i^{SPB}(q, a_i^q) = (q-1)C_i^+ + \sum_{j \in sp(i)} \eta^{+]}(a_i^q) C_j^+ \tag{4.38}$$

Here sp(i) is the set of all traffic streams with a priority equal to that of stream i (excluding stream i). The function $\eta^{+]}(\Delta t)$ conservatively covers the worst-case order of frame arrivals and returns the number of frame arrivals in any closed time interval $[t, t+\Delta t]$. As Ethernet typically serves frames of equal priority in FIFO order, a candidate search is required to compute the worst-case blocking [31]. If a frame arrives as early as possible, it might have to wait longer for a currently transmitted frame to finish. If it, however, arrives as late as possible, more frames of the same priority might have been queued before its arrival, increasing the blocking. In [31] the authors have shown, that the candidates which need to be considered, can be reduced to the instances, where a_i^q coincides with arrivals of interfering same-priority frames.

$$A_i^q = \bigcup_{j \in sp(i)} \left\{ \delta_j^-(n) | \delta_i^-(q) \leq \delta_j^-(n) < \delta_i(q+1) \right\}_{n \geq 1} \tag{4.39}$$

With this, the worst-case queueing delay $w_i(q, a_i^q)$ for the q-th arrival of a frame of stream i, arriving at time a_i^q, can be computed with:

$$w_i(q, a_i^q) = I_i^{LPB} + I_i^{SPB}(q, a_i^q) + I_i^{HPB}(w_i(q, a_i^q)) \tag{4.40}$$

This equation represents a fixed-point problem, which can be solved by iteration, as all terms are monotonically increasing. As a starting point, a conservative lower bound can be chosen, e.g. $w_i(q, a_i^q) = (q-1)C_i^+$.
The largest transmission latency $R_i(q)$ of frame q can be derived from its queueling delay, transmission latency and taking the arrival time of the frame into account, as shown in Fig. 4.19:

$$R_i(q) = \max_{a_i^q \in A_i^q} \left\{ w_i(q, a_i^q) + C_i^+ - a_i^q \right\} \tag{4.41}$$

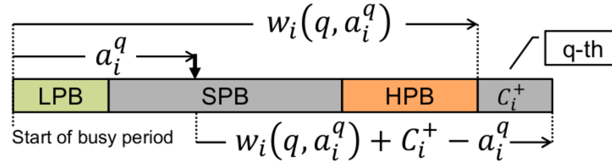


Figure 4.19: Example queuing delay

The worst-case frame transmission latency for all frames within stream i can therefore obtained with:

$$R_i^+ = \max_{1 \leq q \leq \hat{q}_i} \{R_i(q)\} \tag{4.42}$$

where $\hat{q}_i$ is the largest number of frame arrivals of stream i which have to be evaluated. This value is equal to the largest number of frame tranmsisions of stream i in the longest level-i buy period for non-preemptive static priority (SPNP) scheduling [27]. For the SPNP transmission in IEEE802.1Q, this upper bounds for the longest level-i busy period can be computed - similar to the queueing delay - with:

$$\hat{w}_i = I_i^{LPB} + \hat{I}_i^{SPB}(\hat{w}_i) + I_i^{HPB}(\hat{w}_i) \tag{4.43}$$

This equation is again a fixed point problem to be solved iteratively.
In contrast to Eq. 4.38, we do not need to distinguish between individual frames arrivals q. Therefore, $\hat{I}_i^{SPB}(\delta t)$ is defined as:

$$\hat{I}_i^{SPB}(\delta t) = \sum_{j \in sp(i) \cup \{i\}} \eta_j^+(\Delta t) C_j^+ \tag{4.44}$$

The maximum number of frame arrivals of stream i, which have to be considered in Eq. 4.42, is $\hat{q}_i = \eta^+(\hat{w}_i)$.

4.7.4 Ethernet TSN - Time Sensitive Networking

Ethernet Time-Sensitive Networking is a set of upcoming Ethernet standards, which, among other things, introduces new qualtiy-of-service mechanisms to Ethernet. Among these mechanisms are three traffic shapers and a mechanism which allows to preempt frames being sent. One basic principle of TSN is that all traffic shapers use time-driven scheduling. In the scope of SAFURE, analysis methods to evalute the listed mechanisms have been developed and are presented below.

4.7.4.1 TSN Time-Aware Shaper

In this section, we present a local analysis for TSN's time-aware shaper (TAS) [48, 87]. In contrast to IEEE802.1Q, TSN/TAS is a time driven link arbitration scheduler. In order to achieve this, a gate is added to each traffic class in the output port, which - if open - allows frames to pass through, or - if closed - blocks them. The programmable gate schedule defines the time intervals during which frames from specific traffic classes are allowed to pass through. On top of the gate schedule, link access arbitration is performed according to the class priorities. *Guard bands* are used to ensure that frames are only transmitted as long as the corresponding gates are open. These bands define a time period during which frames are not allowed to start transmitting, but frames already being transmitted are allowed to finish. TSN/TAS suggest to protect critical traffic classes by only opening their respective gates during special (scheduled) time intervals, gates from other classes are closed during these periods [48]. We further assume, that these intervals are seperated, i.e. not overlapping, so that link access without interference from higher or lower-priority traffic can be guaranteed. However, there can still be interference from same-priority traffic. Outside of these scheduled intervals, the other gates are opened so that (non-critical) traffic classes are allowed to compete for the link access according to their priorities.
We define $\mathcal{T}$ as the set of all Ethernet traffic classes, which receive isolated link access by the time-aware shaper gate schedule. Further, we define $\mathcal{I} \in \mathcal{T}$ as *TAS classes*, their corresponding traffic streams *TAS streams* and frames *TAS frames*. Each TAS class is assigned to a *TAS interval* t_I^{TAS} which is scheduled with a period of t_I^{CYC}, as shown in Fig. 4.20. While the gate schedule will eventually repeat itself, the TAS intervals do not necessarily have to be periodic or of equal length. This approach is, however, nevertheless chosen, as it naturally fits many automotive use cases. We present two analyes for TSN/TAS to derive worst-case transmission latency bounds for *A* streams of a TAS class and *B* streams of non-TAS classes.

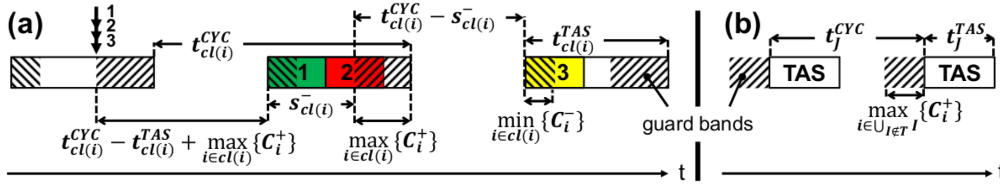


Figure 4.20: TSN/TAS: (a) Common notations. Frame 3 experiences same-priority blocking from Frames 1 and 2. (b) Maximum blocking by a single TAS class.

Analysis of Time-Aware Traffic Streams

Outside of their dedicated TAS interval the frames of all TAS streams i of the TAS class $\mathcal{I}$ are blocked and waiting for their upcoming interval. Within their interval, these streams have exclusive access to the link and only experience same-priority blocking from other streams $j \in cl(i)$, where the function $cl(i)$ maps a traffic stream i to its corresponding traffic class. Frames concurrently arriving, are assumed to be queued in the worst-case order, as in the priotiy blocking in IEEE802.1Q. Thus, to compute the worst-case queueing delay $W_i(q, a_i^q)$ (see Definition 1 in section 4.7.3) same-priority and what we will call closed-gate blocking has to be considered.

Same-priority blocking is independent of the number of TAS intervals, hence it can be computed following eq. 4.38.

Closed-gate blocking: The blocking due to closed gates describes the time the TAS class is blocked as its gate are not open including the time during which no frame preemption is allowed to start transmitting due to the guard band. The amount of closed-gate blocking can be derived from the number of TAS intervals required to serve the accumulated workload requested by TAS class $\mathcal{I}$. Due to the non-preemptive nature of frames this is a combinatorially complex problem. The worst-case combination of arriving TAS frames has to be found so that the maximum blocking is found. Each interval is to be utilised so that no additional TAS frame can start its transmission. This discrete problem is transformed into a continious one:

Theorem 1. The minimum workload, which can be processed in a TAS interval of TAS class $cl(i) = \mathcal{I} \in \mathcal{T}$ is given by:

$$s^-_{cl(i)} = \max\left\{t^{TAS}_{cl(i)} - \max_{i \in cl(i)}\{C^+_i\},\ \min_{i \in cl(i)}\{C^-_i\}\right\} \tag{4.45}$$

Proof: see [87].

We define the accumulated workload which is requested by TAS class $cl(i)$ as Δw. The maximum number of required TAS intervals can then be computed by dividing Δw by $s^-_{cl(i)}$ from eq. 4.45. The longest time interval between any two consecutive TAS intervals is $t^{CYC}_{cl(i)} - s_{cl(i)}$, i.e. the time interval during which no frames of $cl(i)$ are transmitted. The maximum closed-gate blocking for any frame of stream i is then given by:

$$I_i^{CGB(\Delta w} = \left(\left\lceil \frac{\Delta w}{s^-_{cl(i)}} \right\rceil - 1\right)\left(t^{CYC}_{cl(i)} - s^-_{cl(i)}\right) + t^{CYC}_{cl(i)} - t^{TAS}_{cl(i)} + \max_{i \in cl(i)}\{C^+_i\} \tag{4.46}$$

The last three terms represent an upper bound on the round-trip penalty, i.e. the maximum time a TAS frame has to wait for its first TAS interval to transfer. It assumes, that the longest frame has just arrived after the guard band has become active, see Fig. 4.20. This can - for instance - occur if the gate schedules along a path are not synchronised. For perfectly synchronised networks, this term can be improved for tighter results [87].

With Eqs. 4.38 and 4.46, the worst-case queueing delay of the q-th TAS frame, arriving at a_i^q, can be computed:

$$w_i(q, a_i^q) = I_i^{SPB}(q, a_i^q) + I_i^{CGB}\left(I_i^{SPB}(q, a_i^q) + C_i^+\right) \tag{4.47}$$

For the calculation of $I_i^{CGB}(\Delta w)$ in eq. 4.46 the accumulated workload requested by TAS class $cl(i)$ is required as its argument and $I_i^{SPB}(q, a_i^q)$ only considers $(q-1)C_i^+$. Therefore, we must add one transmission C_i^+. The maximum frame transmission latency can be computed similar to IEEE 802.1Q.

Analysis of Non Time-Aware Traffic Streams

While TAS streams are isolated from other classes by closing their respective gates, frames of non-TAS streams potentially interfere with all other non-TAS classes. Therefore, non-TAS traffic can be modelled like IEEE 802.1Q and TAS intervals from all classes $\mathcal{J} \in \mathcal{T}$ can be consider as additional periodic blocking terms, when deriving the worst- case queueing delay.

The blocking by **lower-, same- and higher priority blocking** can be modelled as described in Section 4.7.3 for IEEE802.1Q. Note, that for the calculation of these values, only non-TAS traffic classes must be considered (TAS classes are excluded due to closed gates).
Blocking by TAS classes: We start by bounding the maximum interference a single TAS interval can cause.
Theorem 2. *The maximum blocking caused by a single TAS interval t^{TAS} on a non-TAS class $\mathcal{I} \notin \mathcal{T}$ is*

$$\tilde{t}^{TAS}_{I,J} = \max_{i \in \cup_{I \notin \mathcal{T}} \mathcal{I}} \{C^{+}_{i}\} + t^{TAS}_{J} \tag{4.48}$$

Proof: see [87].
The maximum number of times a TAS class J can interfere with a non-TAS stream i can be computed by dividing Δt by t^{CYC}_{J}. The interference of all TAS classes $J \in \mathcal{T}$ can be calculated with eq. 4.48 on a non-TAS stream i as:

$$I^{TASB}_{i}(\Delta t) = \sum_{J \in \mathcal{T}} \left\lceil \frac{\Delta t}{t^{CYC}_{J}} \right\rceil \tilde{t}^{TAS}_{cl(i),J}. \tag{4.49}$$

This consercatively assumes, that the blocking by different TAS classes does not overlap, i.e. leading to the worst case interference. We can calculate the worst-case queueing delay of frame q arriving at a^{q}_{i} of a non-TAS stream with eqs. 4.36, 4.37, 4.38 and 4.49 as:

$$w_{i}(q, a^{q}_{i}) = I^{LPB}_{i} + I^{SPB}_{i}(q, a^{q}_{i}) + I^{HPB}_{i}(w_{i}(q, a^{q}_{i})) + I^{TASB}_{i}(w_{i}(q, a^{q}_{i})) \tag{4.50}$$

The maximum frame transmission latency can be computed equivalently to eq. 4.41 and 4.42 for IEEE802.1Q by substituting the respective queueing delays with eq. 4.50 or 4.47.

4.7.4.2 TSN Peristaltic Shaper

This section presents a local analysis for the TSN peristaltic shaper (PS) [87]. For the TSN/PS shaper, continous time is divided into two alternating time intervals of equal length: *even* and *odd*. Frames arriving at a switch are - additionally to their priority - classified via their arrival time, i.e. whether they arrive in an even or odd interval. Frames received in an even interval are to be transmitted in the next odd interval and vice versa. Hence, like AVB or TSN/TAS, TSN/PS is not work conserving. If an interval is too short to transmit all frames scheduled for it, the current interval overlaps into the following ones. Frames scheduled for intervals which are affected by this transient overloads are delayed until the overload has been subsided. This way the intra-class frame order is preserved (even) under transient overload.
The motivation for TSN/PS is to provide an easier means to compute latency bounds for the residence time of frames within a switch. This, however, only works if no interfering traffic exists. In reality, a thorough timing analysis is required nonetheless.
We define $\mathcal{P}$ as the set of PS traffic classes, i.e. the ones being scheduled according to their arrival intervals. All other streams - or non-PS classes - are scheduled according to IEEE802.1Q. Traffic classes $\mathcal{I} \in \mathcal{P}$ PS classes and, correspondingly, their traffic streams are called PS streams and frames of these streams PS frames. We further define the length of a PS interval for a $\mathcal{I} \in \mathcal{P}$ PS class as $t^{PS}_{\mathcal{I}}$.
PS frames are scheduled for the following interval, that means, that a frame cannot be sent during its arrival interval, even if the port is not busy. The first arriving frame (and its waiting time) marks the beginning of the busy period and define the PS interval pattern for it. We define this waiting for $\mathcal{I} \in \mathcal{P}$ with ϕ^{PS}_{I}, see Fig. 4.21.
In the following paragraphs, we present the analyses to derive the worst-case transmission latency bounds for PS frames and for non-PS frames.

4.7.4.2.1 Analysis of Non Peristaltic Traffic Streams

Non-PS traffic classes are scheduled according to IEEE802.1Q. Therefore, the calculation of the blocking from **same-priority** streams is equivialent to IEEE802.1Q. The same is true for **lower-priority** traffic, for which both PS and non-PS traffic has to be considered. For blocking by **higher-priority** streams, we have to consider two different cases:
(a) blocking by higher-priority non-PS classes j, $j \in hp(i) \wedge cl(j) \in \mathcal{P}$, and *(b)* the higher-priority blocking is by frames of PS streams j, $j \in hp(i) \wedge cl(j) \in \mathcal{P}$.
For (a), the computation is identical to the computation for higher-priority blocking in IEEE802.1Q:

$$I^{HPB,nPS}_{i}(\Delta T) = \sum_{j \in \{k | k \in hp(i) \wedge cl(k) \notin \mathcal{P}\}} \eta^{+}_{j}(\Delta t) C^{+}_{j} \tag{4.51}$$

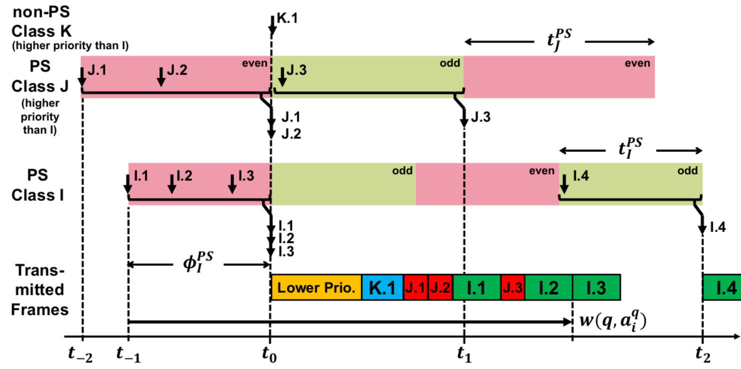


Figure 4.21: TSN/PS: Frames of PS class I experience interference from higher priority classes K (non-PS) and J (PS) and one lower-priority frame. It takes more than one PS interval to transmit the first three frames of PS stream I.

In the case of PS streams (b), we have to take into account that frames of PS streams are delayed until the next interval before they are scheduled. These frames could be delayed for the length of their PS interval $t_{\mathcal{J}}^{\mathcal{PS}}$ in the worst-case, before they can interfere with the frames of stream i. Hence, interference from PS streams occurs at the end of their PS intervals. In Fig. 4.21 frames $J.1$ and $J.2$ from PS class $\mathcal{J}$ are delayed during their arrival intervals and released at the same time at the end of this interval.
The maximum number of PS intervals of class $\mathcal{J}$ in the time interval Δt can be calculated with $\lceil \Delta t / t_{\mathcal{J}}^{PS} \rceil$ In order to get the accumulated interference from PS stream $\mathcal{J}$, we multiply this number with the length of the interval: $\lceil \Delta t / t_{\mathcal{J}}^{PS} \rceil t_{\mathcal{J}}^{PS}$. With this, we can calculate the higher-priority blocking for each PS stream j of class $\mathcal{J}$ for stream i:

$$I_i^{HPB,PS}(\Delta t) = \sum_{j \in \{k | k \in hp(i) \wedge cl(k) \in \mathcal{P}\}} \eta_j^+ \left(\left\lceil \frac{\Delta t}{t_{cl(j)}^{PS}} \right\rceil t_{cl(j)}^{PS} \right) \tag{4.52}$$

Analysis of Peristaltic Traffic Streams

The analysis of a PS traffic stream i is similar to the analysis of non-PS traffic, with the exception of an inital offset $\phi_{cl(i)}^{PS}$ which only affects streams of the analysed PS class $cl(i)$. Streams of other traffic classes are generally allowed to transmit during this time. Therefore, the maximum interference is generated if the interferering traffic classes arrive just before the end of i's offset, directly after $\phi_{cl(i)}^{PS}$ at the beginnnig of the busy period (e.g. at t 0 in Fig. 4.21). Non-PS interferers start interfering with PS stream i directly after $\phi_{cl(i)}^{PS}$ and the end of the first PS intervals of PS interferers is aligned to the end of$\phi_{cl(i)}^{PS}$ (e.g. t_0 in Fig. 4.21) in the worst-case.
The calculation of **lower-priority** blocking is identical to the one for non-PS classes. This blocking starts right after $\phi_{cl(i)}^{PS}$, but is independant of its value.
Same-priority blocking is independent of $\phi_{cl(i)}^{PS}$ as well, as the amount of same-priority blocking only depends on the $q-1$ preceding frames of stream i itself and additional frames of class $cl(i)$ arrived before the arrival time a_i^q of candidate q (see Eq. 4.38).
For blocking caused by **higher-priority** streams, the reasoning from the non-PS analysis is also valid for the PS analysis and Eqs. 4.51 and 4.52 can be used. However, frames of interfering streams released before $\phi_{cl(i)}^{PS}$ cannot interfere during $\phi_{cl(i)}^{PS}$, as argued above. In order to create the worst-case blocking, we align interfering traffic from higher priorty classes so, that their transmission begins directly after $\phi_{cl(i)}^{PS}$, relative to the beginning of the busy period, as shown in Fig. 4.21
Lastly, we have to take the initial offset $\phi_{cl(i)}^{PS}$ as an additional blocking term for stream i into account to compute the worst-case queueuing delay:

$$w_i(q, a_i^q) = \phi_{cl(i)}^{PS} + I_i^{LPB} + I_i^{HPB,nPS}\left(w_i(q, a_i^q) - \phi_{cl(i)}^{PS}\right) + I_i^{HPB,PS}\left(w_i(q, a_i^q) - \phi_{cl(i)}^{PS}\right) + I_i^{SPB}(q, a_i^q) \tag{4.53}$$

The additive term $\phi_{cl(i)}^{PS}$ is used to model the initial waiting time that PS stream i and streams of $cl(i)$ experience. However, interference from other streams affects these streams only after $\phi_{cl(i)}^{PS}$ (relative to the beginning of $w_i(q, a_i^q)$), not during $\phi_{cl(i)}^{PS}$. Therefore, $\phi_{cl(i)}^{PS}$ must be subtracted from the time intervals which determine the

blocking by higher-priority streams. This is also shown in Fig. 4.21, where I.4 is not part of the busy period, even though it arrives within it, as it has to wait for the next interval to be scheduled. The analysis presented here, accounts for this when computing the set of relevant arrival candidate set A_i^q in Eq. 4.39 and the largest value for $\hat{q}$, derived from Eq. 4.43 in Section 4.7.3
Note that, if all interval length $t_{\mathcal{I}}^{PS}$ approach 0, TSN/PS becomes IEEE 802.1Q.

4.7.4.3 TSN Burst Limiting Shaper

The TSN Burst Limiting Shaper (TSN/BLS)[83] is similar to the Ethernet AVB, as both are utilising a credit-based shaper to control bandwith consumption. In contrast to AVB which only allows bursts of critical traffic after previous chargin up credit, TSN/BLS allows bursts of limited size to be transmitted without previous blocking. The rationale behind this is to allow TSN/BLS to react faster to transient overloads of critical traffic.
TSN/BLS limits the workload, which can be released by a traffic-shaped class in a given time interval, by a credit- based traffic shaper [40]. We define $\mathcal{B}$ as the set of all traffic classes which are shaped by a burst-limiting shaper. Accordingly, we refer to these classes as *BLS classes*, streams within these classes as *BLS classes* and frames within these streams as *BLS frames*. Each shaper of a BLS class $\mathcal{I}$ is utilising a credit counter, either allowing the shaper to consider frames of this class $\mathcal{I}$ for the arbitration process, or blocking them. (Other proposals in the case of a blocked traffic stream include frame dropping or reducing the priority of a stream instead of blocking it; these suggestions are not considered here.) Whenever a frame in class $\mathcal{I}$ of size C^+ is sent, ths credit of $\mathcal{I}$ is increased following a predefined sendslope $s_{\mathcal{I}}^S > 0$, i.e. the value $S_{\mathcal{I}}^S C^+$. If the class $\mathcal{I}$ is blocked by frames from other classes, its credit is reduced at the rate $s_{\mathcal{I}}^I > 0$. A class is allowed to send until its credit reaches a certain limit $H_{\mathcal{I}}$ at which point it becomes blocked. The credit is capped at $H_{\mathcal{I}}$, i.e. if the limit is reached during the transmission of a frame, this frame is allowed to finish transmitting, but the credit does not increase beyond $H_{\mathcal{I}}$. While being in the blocked state, the credit of $\mathcal{I}$ also increases with $s_{\mathcal{I}}^I$. Once the credit has been reduced to L_I, class I is allowed to send again. The credit cannot reach values below 0. Figure 4.22 a) shows an example of credit being consumed and recharging. It also shows in b), how the credit of the green class is reduced by higher-priority interference during its enable interval.

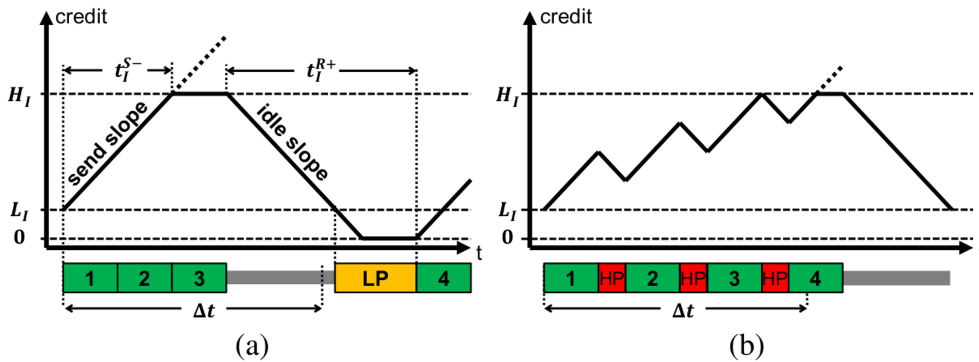


Figure 4.22: TSN/BLS: (a) Frames of class I are sent at their maximum rate until H I is reached and traffic from I is blocked for credit replenishment. (b) Frames of class I are interleaved with frames of other classes and class I's credit is decremented while these frames are being transmitted. Notice how in (b) more workload is released in the marked interval t.

In the following we present the local analyis to compute the worst-case transmission latencies of a frame at a specific switch. We focus again on the delay in the output port. Similarly to IEEE802.1Q and the TSN shaper presented above, we calculate the worst-case queueing delay of frames of I within their corresponding output ports. In order to determine $w_(q, a_i^q)$, we need to consider different blocking effects.
Shaper blocking: If the credit counter of a BLS class $I \in \mathcal{B}$ has reached H_I, it is blocked and has to recharge down to L_I before resuming its transmission. The time to recharge is dependant on these limits and the idle slope s_I^I. If more data has to be sent than fits into one sending interval, this recharging can possibly have to happen multiple times:
Theorem 1. The maximum credit replenishment interval of a BLS class $I \in \mathcal{B}$ is:

$$t_I^{R+} = \left\lceil \frac{H_I - L - I}{-s_I^I} \right\rceil + \max_{j \in lp(I)} \{C_j^+\} \tag{4.54}$$

with $lp(I)$ being the set of streams of all traffic classes with a priority lower than that of class I. Proof: see [83].

The credit recharging described above is required whenever the upper limit H_I is reached for BLS class I. Here, we present the shortest time interval between any two replenishment cycles, i.e. the shortest time for I's credit to go from L_I to H_I. The shortest time occurs, when frames are sent back-to-back, as shown in Fig. 4.22 a).
Theorem 2. The shortest service interval of a BLS class I during a busy period is:

$$t_I^{S-} = \max\left\{\left\lfloor\frac{H_I - L_I}{s_I^S}\right\rfloor, \min_{i\in I}\{C_i^+\}\right\} \tag{4.55}$$

We again define $cl(i)$ as a function that maps a traffic stream i to its corresponding traffic class and Δw as the accumulated workload of BLS class $cl(i)$. We then can calculate the number of replenishment cycles required to transmit the workload Δw as a quotient of $\lceil \Delta w / t_{cl(i)}^{S-} \rceil$. From this, the blocking caused by the shaper $I_i^{SB}(\Delta w)$ for Δw can be bound by:

$$I_i^{SB}(\Delta w) = \left\lceil \frac{\Delta w}{t_{cl(i)}^{S-}} \right\rceil t_{cl(i)}^{R+} \tag{4.56}$$

Since non-BLS streams i are not shaped by credit shapers, they do not experience any shaper blocking, hence $I_i^{SB}(\Delta w) = 0$ if $cl(i) \notin \mathcal{B}$.
Lower-priority blocking, occurs if a lower-priority frame has just started transmitting before analysed frame became ready to send. For BLS streams i, lower-priority blocking, is already covered by the shaper blocking via Eq. 4.54. Since non-BLS streams do not experience shaper blocking and we have to explicitly include lower-priority blocking, hence I_i^{LPB} for $cl(i) \in \mathcal{B}$. This is identical to the calculation of the lower-priority blocking in IEEE802.1Q, i.e. can be found in eq. 4.36.
Higher-priority blocking: We define $hp(i)$ as a function which gives us the set of all traffic streams with a priority higher than that of stream i. We have to distinguish between two different types: a) higher priority non-BLS streams $j \in hp(i) \wedge cl(j) \notin \mathcal{B}$ and b) higher-priority BLS streams $j \in hp(i) \wedge cl(j) \in \mathcal{B}$.
For non-BLS streams, the calculation is equivalent to the calcuation of IEEE802.1Q higher-priority blocking, with the restriction, that only the non-BLS classes are included:

$$I_i^{HPB,nBLS}(\Delta t) = \sum_{j\in\{k|k\in hp(i)\wedge cl(k)\notin\mathcal{B}\}} \eta_j^+(\Delta t) C_j^+ \tag{4.57}$$

For BLS higher-priority streams j, the interference is bounded by J's burst-limiting shaper. However, if frames from j are interleaved with other high-priority frames, J could cause more interference on stream i (see Fig. 4.22 b)) than without these interleaved frames (see Figure 4.22 a)). Therefore, a back-to-back activation of frames from J might not represent the worst-case activation and not lead to the critical instant. Instead, we also have to consider BLS class J to arrive interleaved with frames from other traffic classes. To solve this, we form an integer linear program (ILP) to derive an interleaving pattern that maximizes the interference from BLS class J on stream i for any given time interval Δt:

$$\tilde{I}_{i,j}^{HPB,BLS}(\Delta t) = \text{maximize} \sum_{j\in J} x_j^J C_j^+ \tag{4.58}$$

with $x_j^J \in \mathbb{N}$ representing how many frames of each stream $j \in J$ contribute to J's interference.
Firstly, we look at which interfering traffic streams of other priorities (classes) can be interleaved with class J. As the analysis is considering the stream i, the higher-priotiy BLS class J can be interleaved with frame of any streams of a traffic class of higher of priority to class $cl(i)$. We define $D_J^i = hep(i) \backslash J$ as the set of interleaving streams, with $hep(i)$ returning the set of all traffic streams with a higher or equal priority to that of stream i, including i itself. Lower priority frames are considered separately.
At any point, the credit of BLS class J must be positive and smaller than H_J, leading to the following constraint:

$$0 \leq \sum_{j\in J} x_j^J s_J^S C_j^+ + \sum_{j\in D_i^J} x_j^D s_J^I C_j^+ + x^R s_J^I t_J^{R+} \leq H_J + s_J^S \max_{j\in J}\{C_j^+\} \tag{4.59}$$

with x_j^D representing the number of frames for each interleaved traffic stream $j \in D_i^J$. The equation also considers, that the credit of J can be reduced by inserting $x^R \in \mathbb{N}$ credit replenishment intervals t_J^{R+}. Each of these recharging periods decreases the shaper credit of J by $s_J^I t_J^{R+}$. Since frames of J are non-preemptive, if a frame of J just started transmitting before J reaches its upper credit limit H_J, the upper bound is extended by $s_J^S \max_{j\in J}\{C_j^+\}$.

The workload a port scheduler can release in a given time interval Δt is limited, i.e. Δt plus a frame which started transmitting just before the end of Δt. [6]. This holds true for BLS class J and interleaving frames from D_i^J. The workload is composed of both J and D_i^J parts. Since for J, the accumulated workload amounts to at most Δt plus J's longest frame, we can formulate the workload constraint:

$$0 \leq \sum_{j \in J} x_j^J C_j^+ + \sum_{j \in D_i^J} x_j^D C_j^+ + x^R t_J^{R-} \leq \Delta t + \max_{j \in J}\{C_j^+\} \quad (4.60)$$

We conservatively model the replenishment intervals with their shortest delays (see Theorem 1).

$$t_J^{R-} = \left\lfloor \frac{H_J - L_J}{-s_J^I} \right\rfloor \quad (4.61)$$

The activation event models of J and their interleaving streams bound the number of frames they can issue with $0 \leq x_j^J \leq \eta_j^+(\Delta t), \forall j \in J$ and $0 \leq x_j^D \leq \eta_j^+(\Delta t), \forall j \in D_i^J$. The number of replenishments intervals can be bounded by $0 \leq x^R \leq \eta_J^{R+}(\Delta t)$.
To cover the worst-case, η_J^{R+} models that the shortest service interval of BLS class J t_J^{S-}, is always followed by the shortest credit replenishment interval of J:

$$\eta_J^{R+}(\Delta t) = \left\lceil \frac{\Delta t}{t_J^{S-} + t_J^{R-}} \right\rceil \quad (4.62)$$

We conservatively bound the higher-priority blocking of BLS classes of a traffic stream i by assuming, that the interference from different BLS classes J are independent and accumulate them:nterference over all these J.

$$I_i^{HPB,BLS}(\Delta t) = \sum_{J \in \{cl(j) | j \in hp(i)\} \cap \mathcal{B}} \tilde{I}_{i,j}^{HPB,BLS}(\Delta t) \quad (4.63)$$

Same-priority blocking: Frames can experience blocking from frame with the same priority. For the TSN/BLS, this blocking is identical to IEEE802.1Q and can be found in eq. 4.38.
We, again, have to perform a candidate search in order to determine the proper worst-case of same-priority blocking. The set of relevant candicates is identical to the one for IEEE802.1Q and can be found in eq. 4.39
With the individual terms, we can compose the worst-case queueing delay for the q-th frame of stream i, arriving at a_i^q:

$$w_i(q, a_i^q) = I_i^{SB}(I_i^{SPB}(q, a_i^q) + C_i^+) + I_i^{LPB} + I_i^{SPB}(q, a_i^q) + I_i^{HPB,nBLS} w_i(q, a_i^q) + I_i^{HPB,BLS} w_i(q, a_i^q) \quad (4.64)$$

Note that the shaper blocking I_i^{SB} takes the accumulated workload of BLS classes $cl(i)$ as its argument and the inner function I_i^{SPB} only considers $q-1$ frames of stream i. Therefore, we need to add an additional C_i^+. As a reminder: $I_i^{SB}(\Delta w) = 0$ if $cl(i) \notin \mathcal{B}$ and $I_i^{LPB} = 0$ if $cl(i) \in \mathcal{B}$. Since $w_i(q, a_i^q)$ occurs on both sides of the equation, eq. 4.64 cannot be solved directly, but represents a fixed-point problem. It has to be solved by iteration, which can be aided by using an appropriate starting point, for example $w_i(q, a_i^q) = (q-1)C_i^+$
Having the worst-case queueing delay obtained, we can calculate the transmission delay with eq. 4.41. The worst-case frame transmission latency for a frame of stream i can be calculated equivalently to IEEE802.1Q in eq. 4.42. The calculation of $\hat{q}$ is equivalent to the calculation for AVB in [6].

4.7.5 TSN Frame Preemption

Previously, frame transmission - independently of the arbitration mechanism - has been non-preemptively, i.e. any frame which has started to transmit will finish this transmission [84]. For an Ethernet link working at 100Mbit/s, the longest Ethernet frame can block the link for about 120 μs. This means, that even high-priority time-critical traffic can be fully blocked by a frame of lower priority at each switch for 120 μs, which might be problematic for time-critical applications. The upcoming IEEE 802.3br standard for Ethernet addresses this problem by introducing frame preemption to Ethernet. The frame preemption allows to reduce the aforementioned blocking of 120 μs in a 100 Mbit/s network to be reduced to about 12 μs, at the price of a preemption overhead.
For IEEE802.1Q the frame preemption is illustrated in Fig. 4.23, where two frames, *LP* with a low priority and *HP* with a high priority, are transmitted. It shows the comparision between non-preemptive and preemptive transmission, as well as sketches the preemption overhead.

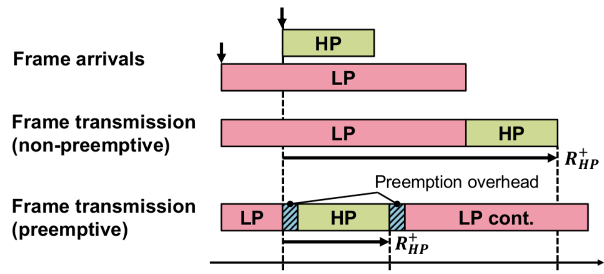


Figure 4.23: Example of non-preemptive and preemptive frame transmission.

The frame preemption is introduced for Ethernet in IEEE802.3br [46] and for Ethernet TSN in IEEE802.1Qbu [47] Here, we focus on how the preemptions affect the timing behaviour for these two.
In IEEE802.3br two MAC interfaces are defined: the *express* and *preemtable* MAC interface. Exactly one level of preemption is supported and only frames from express classes can preempt preemptable classes. This means, that preemptable frames cannot be preempted by other preemptable frames and express frames cannot preempt other express frames. The preemption is defined on link-level in IEEE802.3br, i.e. frames are split into fragments and are reassembled at the MAC interfaces, so that switches (internally) only process complete frames to make the preemption transparent to the phycial layer of Ethernet. Therefore, each fragment must appear to the PHY as a valid Ethernet frame and multiple MAC frame formats are defined, shown in Fig. 4.24.

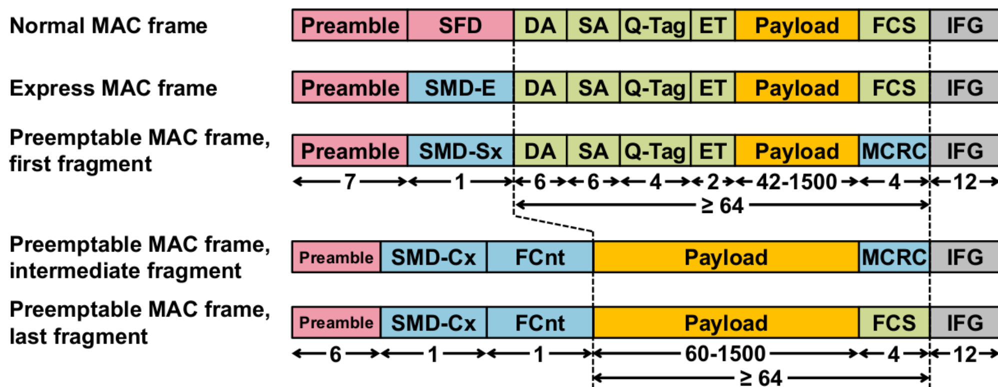


Figure 4.24: Frame preemption introduces new frames, which all have to look like valid Ethernet frames to the PHY.

In order to represent valid frames, this also means, that even fragmented frames have to satisfy the minimum Ethernet frame size, being 84 bytes if the inter-frame gap (IFG) is included. If the payload were too small, the frame is padded. However, preemption is not allowed to add any additional padding to frames, which leads to a restriction on the preemption and its granularity. A) A frame from the preemptable class can only be preempted after the transmitted payload is sufficient to represent the minimum frame size, B) a frame from the preemptable class can only be preempted if both halfs are sufficiently large to satisfy the minimum frame sizes of both frames. These restrictions limit to the minimum frame size of preemptable frames, leading to the longest non-preemptable frame size of *143 byte*. From the standard, we get that the preemption overhead for each additional frame fragment is *24 byte*.
In the following, we present the local analysis of the effect of frame preemption on IEEE802.1Q and TSN (IEEE802.1Qbv).

4.7.5.1 Frame preemption in IEEE802.1Q

In this section we present the local analysis of the timing effects of frame preemption in IEEE802.1Q. We consider both express and preemptable frames and compute their frame transmission latencies. Again, we only consider the queueing delay in the output ports (see Section 4.7.3) and utilise the level-i busy period approach. Even though we consider *frame preemption* fragments themselves are non-preemptive. Hence, we have to adapt the definition of the queueing delay from above in Section 4.7.4.1.
Definition 1. The worst-case queueing delay $w_i(q, a_i^q)$ is the time interval from the beginning of the level-i busy period until the last non-preemptable part of said q-th frame can be transmitted for stream i.

For frames from the express class, this part is the frame itself. For frames from the preemptable class, this is a non-preemptable, minimum-sized fragment.
We consider the two sets of traffic classes: $\mathcal{P}$ the set of *preemptable* traffic classes and $\mathcal{E}$ the set of *express* traffic classes. We again define the function $cl(i)$ mapping a stream i to its class, $lp(i)$ to return the set of all traffic streams with a lower priority than i, $hp(i)$ the set of streams with a higher priority and $sp(i)$ all other streams with the same priority, respectively. We assume, that all express classes have a higher priority than the preemptable ones. In order to compute the worst-case queueing delay, we have to take into account several blocking effects.
Lower-priority blocking: Identical to IEEE802.1Q, for $\mathcal{P}$ classes, the worst-case lower priority blocking occurs if the longest frame of a lower priority stream started sending right before the first frame of stream i gets ready for transmission. Then, the lower priority blocking for preemptable streams can be computed with:

$$I_i^{LPB,P} = \max_{j \in lpl(i)} \{C_j^+\} \tag{4.65}$$

Before computing the lower-priority blocking of express classes, we define the two functions: $lp^P(i)$ which yields the set of preemptable lower-priority classes and $lp^E(i)$ yielding the set of express traffic classes with a lower priority. With these functions, we can calculate the lower-priority blocking for express traffic with:

$$I_i^{LPB,E} = \max\left\{\max_{j \in lp^E(i)} \{C_j^+\}, min\left\{\max\{C_j^+\}, \frac{143\text{byte}}{r^{TX}}\right\}\right\} \tag{4.66}$$

Same-priority blocking: A frame of traffic stream i can be blocked by frames from traffic streams of equal priority [43]. We assume the worst-case ordering for concurrently arriving frames of the same priority. For frames of the express class, this is identical to the same-priority blocking in IEEE802.1Q, hence can be found in Eq. 4.38.
For preemptable traffic, we need consider different fragments. As argued above, only the last (non-preemptable) fragment frame is not part of the queueing delay. Therefore, the last fragment of q has to wait for its preceding fragments to finish, in addition to frames from streams of its own priority (including its own $q-1$ frames). Assumting the worst-case fragmentation, the last fragment is left with the mimimum size of 84 bytes, leaving the rest of q's size for preemption ($C_i^+ - {}^{84byte}/_{r_{TX}}$)

$$I_i^{SPB,P}(q, a_i^q) = (q-1)C_i^+ + C_i^+ - \frac{84\text{byte}}{r_{TX}} + \sum_{j \in sp(i)} \eta_j^{+]}(a_i^q)C_j^+ \tag{4.67}$$

Frames are generally processed in FIFO order within the same priority in Ethernet. Therefore, as argued in Section 4.7.3 we need to perform a candidate search to determine the worst-case same priority blocking. For IEEE802.1Q with frame preemption, this is identical to IEEE802.1Q without preemption, hence the set of relevant candidates can be calculated with eq. 4.39
Higher-priority blocking: Frames of stream i can be blocked by traffic streams with higher priority than i. This is identical to the calculation of the higher-priority blocking in IEEE802.1Q without frame preemption. The calculation can be found in eq. 4.37. If, however, this blocking occurs due to preemption, we need to consider additional overhead (see below).
Preemption overhead: In IEEE 802.3br, preemptable frames suffer an overhead when preempted. This overhead can be modelled as an additional blocking term, if the number of preemptions is known. We bound the number of preemptions of a single frame of stream i with:

$$F_i^+ = \left\lfloor \frac{p_i^+ - 42\text{byte}}{60\text{byte}} \right\rfloor \tag{4.68}$$

with p_i^+ being the maximum payload of i. The maximum number of frame preemptions within the queueing delay can be calculated by multiplying the number of preemptable frames in this period by their respective F_i^+. A preemptable frame fo i can only be blocked once by a lower-priority frame of j, however, this lower-priority frame might get preempted, leading to additinal overhead. This overhead depends on the number of preemptions this frame of j can experience. This number is given with:

$$N_i^{LP} = \max_{j \in lp^P(i)} \{F_j^+\} \tag{4.69}$$

Similarly, we have to consider the preemption overhead of frames of preemptable traffic with the same priority as stream i. The total number of preemptions of a traffic class $cl(i)$ is given by i's own preemption plus the preemptions of other classes:

$$N_i^{SP}(q, a_i^q) = qF_i^+ - 1 + \sum_{j \in sp(i)} \eta^{+]}(a_i^q)F_j^+ \tag{4.70}$$

We again, silimiar to eq. 4.67, the last fragment is not taken into account.
We also need to consider the preemption overhead of higher-priority preemptable frames. The maximium number frames of a higher priority than i can be preempted is given with:

$$N_i^{HP}(\Delta t) = \sum_{j \in hp^P(i)} \eta_j^+(\Delta t)F_j^+ \tag{4.71}$$

So far, eqs. 4.69, 4.70 and 4.71 have bounded the number of preemptions based on the worst-case way each frame can theoretically be fragmented. We can, however, also bound this number by the worst-case preemption pattern of express streams with a higher priority than stream i, which we group in $hp^E(i)$. The preemption overhead in a given time interval Δt with q frames of stream i can be computed for the arrival time a_i^q with:

$$I_i^{PO}(\Delta t, q, a_i^q) = \frac{24\text{byte}}{r^{TX}} \min \left\{ \sum_{j \in hp^E(i)} \eta_j^+(\Delta t), N_i^{LP} + N_i^{SP}(q, a_i) + N_i^{HP}(\Delta t) \right\} \tag{4.72}$$

The worst-case queueing delay of express traffic streams of the q-frame of stream i arriving at a_i^q is:

$$w_i^E(q, a_i^q) = I_i^{LPB,E} + I_i^{SPB,E}(q, a_i^q) + I_i^{HPB}(w_i^E(q, a_i^q)) \tag{4.73}$$

This equation represents a fixed-point problem as $w_i^E(q, a_i^q)$ occurs on both sides and has to solved iteratively. As for previous similar fixed-point problems, $w_i^E(q, a_i^q) = (q-1)C_i^+$ could be used as a starting point.
For traffic i of preemptable streams i, we have to consider the additional preemption overhead for the computation of the worst-case queueing delay:

$$w_i^P(q, a_i^q) = I_i^{LPB,P} + I_i^{SPB,P}(q, a_i^q) + I_i^{HPB}(w_i^P(q, a_i^q)) + I_i^{PO}(w_i^P(q, a_i^q), q, a_i^q) \tag{4.74}$$

This, again, is a fixed point problem, to be solved iteratively, with a starting point suggestion of $w_i^E(q, a_i^q) = (q-1)C_i^+$.
We can derive the worst-case transmission delay $R_i(q)$ for the q-th frame of stream i from the queueing delay. For express streams, this is the queueing delay plus its transmission time, relative to the actication time a_i^q. For preemptable streams, we have to take into account, that we calculated the queueing delay for the last fragment of a preempted frame. Therefore, instead of adding the transmission delay of the whole frame, we only consider the transmission delay of the last fragment:

$$R_i(q) = \begin{cases} \max\limits_{a_i^q \in A_i^q} \left\{ w_i^E(q, a_i^q) + C_i^+ - a_i^q \right\} & \text{if } cl(i) \in \mathcal{E} \\ \max\limits_{a_i^q \in A_i^q} \left\{ w_i^P(q, a_i^q) + \frac{84\text{byte}}{r_{TX}} - a_i^q \right\} & \text{if } cl(i) \in \mathcal{P} \end{cases} \tag{4.75}$$

With the transmission delay for each frame, we can derive the worst-case delay of a traffic class $cl(i)$, which is done identical to IEEE802.1Q without frame preemption, see eq. 4.42.
Now we present how to compute the longest level-i busy period from which $\hat{q}$ can be derived for eq. 4.42. As mentioned in the description of eq. 4.42, we do not need to distinguish between individual frames. Hence, the same priority part $\hat{I}_i^{SPB}$ can be computed following eq. 4.44.
Similarly, we can calculate the level-i busy period preemption overhead via $I_i^{PO}(\Delta t, q, a_i^q)$ from eq.4.80 as $\hat{I}_i^{PO}(\Delta t, q, a_i^q)$ which substitutes $N_i^{SP}(q, a_i^q)$ with:

$$\hat{N}_i^{SP}(\Delta t) = \sum_{j \in sp(i) \cup \{i\}} \eta_j^+(\Delta t)C_j^+ \tag{4.76}$$

With this update, the level-i busy periods can be computed for express ($\hat{w}_i^E$) and preemptable ($\hat{w}_i^P = I_i^{LPB,P}$) traffic classes with:

$$\hat{w}_i^E = I_i^{LPB,E} + \hat{I}_i^{SPB}(\hat{w}_i^E) + I_i^{HPB}(\hat{w}_i^E) \tag{4.77}$$

and

$$\hat{w}_i^P = I_i^{LPB,P} + \hat{I}_i^{SPB}(\hat{w}_i^P) + I_i^{HPB}(\hat{w}_i^P) + \hat{I}_i^{PIO}(\hat{w}_i^P) \tag{4.78}$$

Again, eqs. 4.77 and 4.78 are fixed-point problems, to be solved iteratively, with a starting point suggestion of $\hat{w}_i^P = C_i^+$. The values for $\hat{q}$ can then be derived with $\hat{q}_i = \eta_i^+(\hat{w}_i^P)$ and $\hat{q}_i = \eta_i^+(\hat{w}_i^E)$, respectively.

4.7.5.2 Frame preemption in TSN (IEEE802.1Qbv)

In this section we present the effects of frame preemption onto the timing behaviour of the time-aware shaper (see section 4.7.4.1). In TSN/TAS, traffic is divided into critical and non-critical traffic, which are scheduled in separate time intervals. These time intervals are protected against overlapping transmission by guard bands, whose length are dictated by the longest possible frame of its corresponding time interval. This can result in low link utilization, due to long guard bands. The motivation of frame preemption is to improve the latency of critical traffic. Therefore, we assume that time-triggered traffic in TSN/TAS is defined of express traffic, hence cannot be preempted. Since TSN/TAS already guarantees that critical traffic is transmitted without interference from other traffic classes, the critical traffic can not profit from adding frame transmission to the network. However, setting the less critical traffic as preemptable allows to reduce guard bands within their respective segments, as the longest possible blocker can be significantly shortened when preempted, improving the end-to-end latency for non-critical traffic.

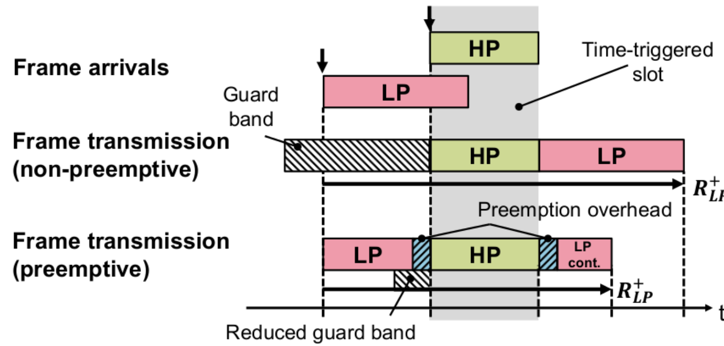


Figure 4.25: Example of non-preemptive and preemptive frame transmission in IEEE 802.1Qbv

This is illustrated in Fig. 4.25. With non-preemptive scheduling, the frame *LP* would arrive during its guard band period and not be allowed to start its transmission until after *HP*'s time slot. If frame preemption is allowed, the guard band can be shortened allowing a partial transmission of *LP* before *HP*'s slot reducing its overall transmission latency. The worst-case blocking a non-TAS class stream $\mathcal{I}$ can suffer by the time-triggered interval with length t_J^{TAS} from express classes $\mathcal{E}$, can be computed similar to eq. 4.48, with the reduced guard band[5]:

$$\tilde{t}_{I,J}^{TAS} = \min\left\{ \max_{i \in \bigcup_{I \in \mathcal{P}} I} \{C_i^+\}, \frac{143\text{byte}}{r_{TX}} \right\} + t_J^{TAS} \tag{4.79}$$

Traffic classes of the preemptable MAC interface cannot preempt each other, hence, frame preemption does not affect the lower-, same- or higher-priority blocking. Their calculation is identical to eqs. 4.36, 4.38 and 4.37 in section 4.7.3.
We can compute the number I_i^{TASB} of interrupting time intervals[6] by traffic classes $\mathcal{E}$ identical to eq. 4.49 by using $\tilde{t}_{I,J}^{TAS}$ from 4.79.
Preemption overhead: Less-critical traffic streams cannot preempt frames from the same MAC interface. Therefore, only one frame preemption can occur per time-triggered slot. Multiplying the number of possible preemptions with the frame overhead yields the total overhead per time interval:

$$I^{PO}(\Delta t) = \frac{24\text{byte}}{r_{TX}} \sum_{J \in \mathcal{E}} \left\lceil \frac{\Delta t}{t_J^{CYC}} \right\rceil \tag{4.80}$$

[5] Note that in [84] the term $\tilde{t}_{I,J}^{TT}$ was used instead of $\tilde{t}_{I,J}^{TAS}$
[6] Note that in [84] the term I_i^{CTB} was used instead of I_i^{TASB}

The worst-case queuing delay $w_i^P(q, a_i^q)$ for frame q of stream i arriving at time a_i^q under frame preemption can then be calculated with:

$$w_i^P(q, a_i^q) = I_i^{LPB,P} + I_i^{SPB,P}(q, a_i^q) + I_i^{HPB}(w_i^P(q, a_i^q)) + I_i^{CTB}(w_i^P(q, a_i^q)) + I^{PO}(w_i^P(q, a_i^q)) \quad (4.81)$$

With the worst-case queuing delay, we can calculate the frame transmission delay $R_i(q)$ via eq. 4.41 and the worst-case transmission delay R_i^+ of stream i with eq. 4.42, similar to IEEE802.1Q.
Then, the maximum frame transmission latency of the q-th frame of traffic stream i under IEEE 802.1Qbv with preemption can be computed very similarly to that under IEEE 802.1Q (Eqs. (15) and (16)). As argued before, the worst-case timing guarantees of critical traffic in IEEE 802.1Qbv are not affected by frame preemption and can be computed as presented in [8].

4.7.6 Software Defined Networking

In this section we want to perform an evaluation of the general suitablility of the concept of SDN for real-time networks [82]. We present a simplified version of SDN/OpenFlow, as it was not designed with real-time constraints in mind. The simplified approach differs from Openflow in two major points: a) instead of TCP, we use UDP as the transport protocol and b) we use shorter Ethernet messages in comparision to the standard OpenFlow messages.

SDN-based Network Configuration

SDN network traffic is stream (or "flow") based, which means that a traffic stream is defined as a sequence of frames from the same source to the same destination are routed equally. In accordance with OpenFlow, we assume that frames are routed by the data plane within the switches represented by flow tables, with a set of rules, determining the action to be applied to the respective frames. Arriving frames are matched against the rule fields, e.g. by their MAC and/or IP addresses, VLAN ID, or TCP/UDP ports. The rules, which can be applied to the frames, are forwarding the frame to a specific switch port, drop the frame, and request further instructions from the SDN controller on how to handle the frame. SDN networks are configured by the process of creating and distributing flow table entries, which is performed by the SDN controller. We will consider two different means of this configuration for the example of admission control.
Admission control manages the access of streams to the network. Either flows are allowed to enter the network (forward action), are blocked (drop action), or require an explicit configuration from the SDN controller (request action). For this section, we evaluate the worst-case timing behaviour of such a request action.
When the first frame of a newly incoming flow arrives at a switch (Switch 1 in Fig. 4.26), this switch checks its flowtable for a matching rule to decide whether to forward or block the stream, or - in this example - generate a request (*req*) and send it towards the controller. If the controller decides to deny access for this stream, it sends this configuration to the requesting switch at once (this scenario is not shown in Fig. 4.26). However, if the controller decides to allow the stream to access the network, multiple switches have to be reconfigured, potentially. The configuration messages (*conf*) sent to the switches can contain multiple updates for their respective flow tables, even modifying entries for other flows. After having received the *req* message, each switch confirms the update to the SDN controller (*ack*). Once each switch has confirmed the update, the controller sends a confirmation to Switch 1 (*en*) to enable the flow table entries, allowing the requesting flow to enter the network.
Each step of this protocol introduces a certain delay. The red boxes in Fig. 4.26 represent the execution times of the CPUs on the SDN agents and controller, including the interference from other SDN related requestes/configurations. The communication delay due to the Ethernet communication is indicated by the blue arrows, including the interference from all traffic stream in the network. These different delays compose the overall delay R_{SDN}^+ from the initial request until the admission is granted.
Alternatively to the explicit flow control, the configuration can be shifted towards the initial network configuaration. This means, that switches are preconfigured (which is typically done in automotive systems) but requires that the network changes are known beforehand. The admission then can be granted or denied by simply toggling the flow tables of the flow's entry switches, as shown in Fig. 4.26 b). This can significantly reduce the time for the admission control to a simple request/enable handshake.

Modeling SDN in Compositional Performance Analysis

In this section we present the modelling for the above described admission control scheme in CPA. We use the simple example system shown in Fig. 4.29.

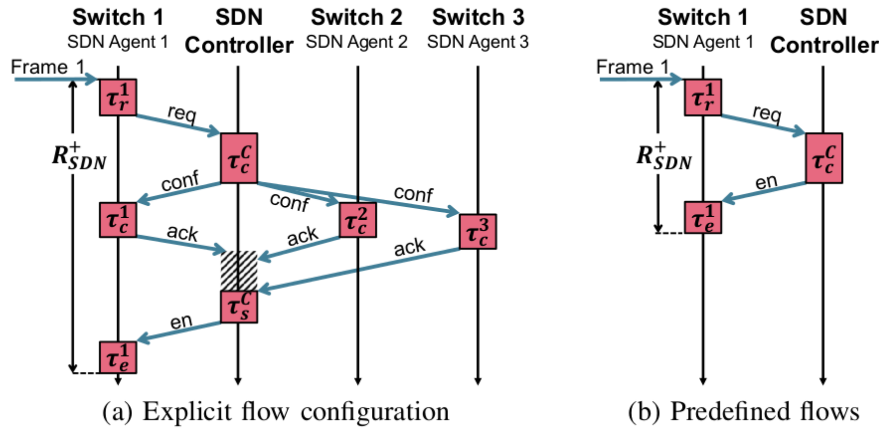


Figure 4.26: SDN network (re)configuration protocols

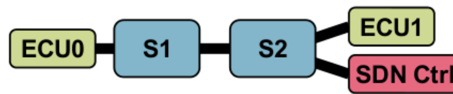


Figure 4.27: Example network for the SDN admission control

The example system consists of two switches, each with their local SDN agent, and the SDN controller, connected to switch S2. Note, that there is non-SDN traffic sent from ECU0 to ECU1. For this example, we assume that only S1 sends a request to the controller
The CPA model of the explicit admission control scheme is shown in Fig. 4.29. We model the processing resources (CPUs) of the SDN agents in the switches and of the SDN controller as CPA resources. The processing of SDN messages is modelled as tasks on their respective SDN resources, e.g. τ_r^1 represents the request message generation on performed on switch S1. The Ethernet communication flow is identical to the one described in the previous sections and above, here with IEEE802.1Q as the chosen transmission scheme: A frame transmission is modelled by mapping the transmission task onto the output port resource, with the task exeuction time corresponding to the frame size. The synchronization mechanism at the SDN controller (hatched time interval in Figure 2a) is modeled by an AND-junction of the corresponding traffic streams in the CPA model, i.e. the controller waits for all frames to arrive before starting to work. Since the SDN and non-SDN traffic partially share the same path, interference between these two classes will occur, to be seen at Port1.2 on switch S1. The total worst-case latency R_{SDN}^+ for the admission request can be upper bounded by first deriving the worst-case end-to-end latencies from task τ_r^1 at the requesting switch S1 to the admission task τ_e^1, also at S1. Note, that we have a fork in distribution of the *conf* messages, which is rejoined at the SDN controller at the AND junction before τ_s^C. In order to provide an upper bound, we need to consider the maximum path latency over all forked paths from τ_c^C to τ_e^1.
The implicit flow control scheme is presented in Fig. 4.29, showing only the handshake as discussed above. As expected, this model is less complex.

4.8 Vulnerability Detection for Networks

A timing analysis approach for networks has been presented in D3.1, which can be used to detect timing vulnerabilities (usually at design time) of the network. the description in D3.1 was already of the final state of the algorithms, which since then have been implemented. Hence, no update is given in this deliverable.
In deliverable D3.3, a methodology for applying this approach is presented.

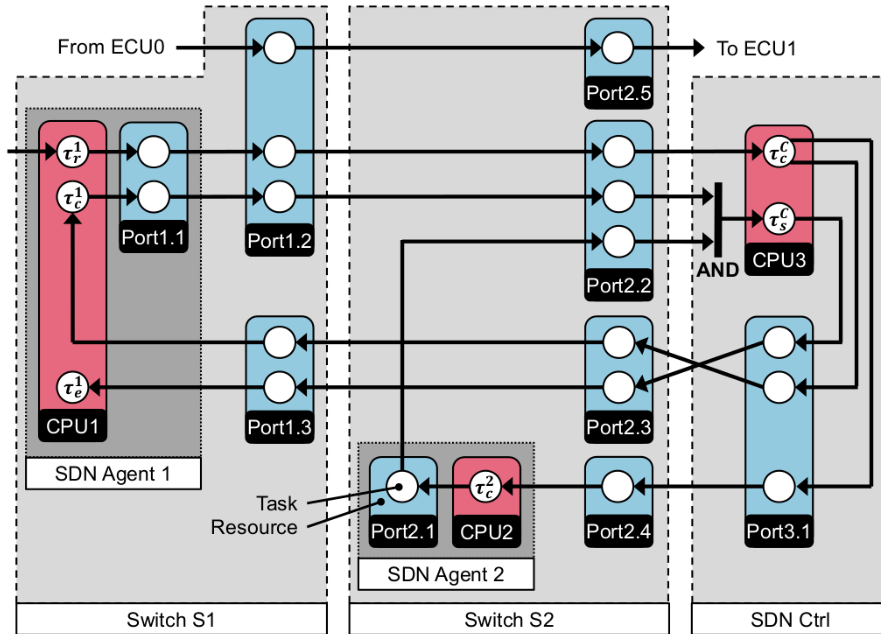


Figure 4.28: CPA model for the explicit flow configuration protocol from Figure 4.26 a)

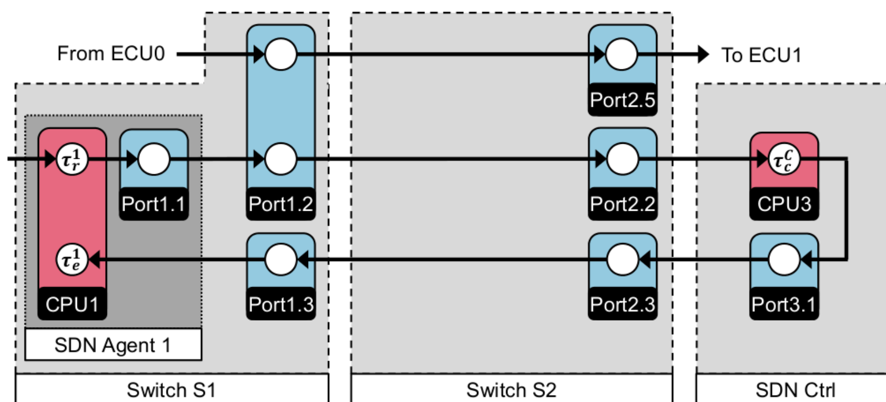


Figure 4.29: CPA model for the predefined flow protocol from Figure 4.26 b)

Chapter 5 Conclusion

This document overviews the different integrity aspects considered in SAFURE and their respective integrity algorithms. This document presents the final analysis of integrity algorithms. We consider three different aspects of integrity:

5.1 Temperature Integrity

For temperature integrity, we detail a server based thermal protection scheme (TI Server). This scheme can be used to provide thermal protection in a mixed-critical context. Details on how this scheme can be applied in a hardware platform running a dual-critical application are given in D3.2.
We also detail how temperature sensors and systems DVFS mechanisms can be exploited to compromose system security. A quantification of this security threat and possible mitigation actions are dovered in D3.3. Lastly, we present a preliminary study on thermal side-channel attacks.

5.2 Data Integrity

For data integrity, we have implemented several state-of-the-art algorithms and some quite new algorithms (EdDSA, Poly1305 and KMAC). We have investigated their performance regarding speed, memory (ROM and RAM) consumption, and overall suitability for usage in embedded systems.

5.3 Timing and Resource Sharing Integrity

Regarding timing and resource sharing integrity, we have shown how to analyze and detect vulnerabilities. In deliverable D3.3, we give indications about how to prevent timing integrity violations.

5.4 Integration Plan

Table 5.1 presents the integration plan for technologies presented in this deliverable. *Telecom* denotes the telecommunication use-case based on Sony Xperia platform, *Automotive* denotes the automotive use-case based on Infineon Aurix platform, and *Juno* denotes the WP4 use-case based on ARM Juno Board platform. Table 5.1 has been composed and added to this deliverable in response to reviewer's recommendation. However, please note that this table presents a tentative plan. Final integration details will be presented in SAFURE Deliverable D6.7.

Technology	Section	Use-case	Comments on integration
TI Server	2.1	*Juno*, *Telecom*	Feasibility and extent of integration on *Telecom* will be assessed in September 2017.
Security implications of temperature	2.2, 2.3, 2.4	*Telecom*	Architectural characteristics of *Telecom* are similar to the platforms evaluated in sections 2.2-2.4. Therefore the security implications hold for *Telecom*. Further integration steps are not planned.
Data integrity algorithms	3.1, 3.2, 3.3, 3.4	*Automotive*, *Telecom*	Extent of integration will be assessed in September 2017.
Budget-Based RunTime Engine	4.2	*Juno*	BB-RTE is currently evaluated on the *WP4 prototype*
		Telecom	Feasibility and extent of integration on *Telecom* will be assessed in September 2017. BB-RTE requires PikeOS with privileged driver and MUXA-over-ethernet
Timing integrity in overload conditions	4.3 and 4.4	*Automotive*	The algorithms have been validated on a fuel injection application provided by the automotive partner MM (as reported in the paper). The tool integration is currently not planned.
Function placement optimization	4.5	*Automotive*	The placement optimization algorithm will be applied to applications examples and architectures defined for the automotive case studies. The tool integration is currently not planned.
Multicores vulnerability due to contention	4.6	*Juno*	Currently being integrated on the *WP4 prototype*
		Automotive	Currently being integrated on the *Automotive* multicore use case
Real-time Ethernet	4.7	*Automotive*	Different Ethernet transmission schemes will be evaluated in the virtual demonstrator
Vulnerability Detection for Networks	4.8	*Automotive*	Algorithms presented in D3.1 with methodology in D3.3. Implementation in the model-based SymTA/S timing analysis tool to be applied to automotive scenarios.

Table 5.1: Integration plan for technologies presented in deliverable D3.2

Bibliography

[1] ARM® CoreSight©ip, . URL https://www.arm.com/products/system-ip/coresight-debug-trace.

[2] Nexus 5001 forum, . URL http://www.nexus5001.org.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/.

[4] ARINC Industry Activities. *Avionics Full DupleX Switched Ethernet (AFDX)*. 2002.

[5] R. Ahmed, P. Huang, M. Millen, and L. Thiele. On the design and application of thermal isolation servers. In *ACM International Conference on Embedded Software (EMSOFT). Accepted for publication*, 2017.

[6] Philip Axer, Daniel Thiele, Rolf Ernst, and Jonas Diemer. Exploiting shaper context to improve performance bounds of ethernet avb networks. In *Proc. of DAC*, San Francisco, USA, June 2014.

[7] Elaine Barker. Recommendation for key management Part 1: General (Revision 4). *NIST special publication*, 800(57):1–147, 2016.

[8] Davide B. Bartolini, Philipp Miedl, and Lothar Thiele. On the Capacity of Thermal Covert Channels in Multicores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 24:1–24:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901322. URL http://doi.acm.org/10.1145/2901318.2901322.

[9] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. In *Advances in Cryptology-CRYPTO 2006*, pages 602–619. Springer, 2006.

[10] Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *Computers, IEEE Transactions on*, 50(4):308–321, 2001.

[11] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[12] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core COTS architectures. *Embedded Real Time Software and Systems conference*, Feb 2014.

[13] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39(1-3):5–30, 2008.

[14] Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. Response-time analysis for real-time tasks in engine control applications. In *Proceedings of the 6th International Conference on Cyber-Physical Systems (ICCPS 2015)*, Seattle, Washington, USA, April 14-16, 2015.

[15] Alessandro Biondi, Marco Di Natale, Youcheng Sun, and Stefania Botta. Moving from single-core to multicore: initial findings on a fuel injection case study, 2016.

[16] B. Brandenburg. The FMLP+: An Asymptotically Optimal Real-Time Locking Protocol for Suspension-Aware Analysis. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 61–71, July 2014.

[17] B. Brandenburg and J. Anderson. Optimality Results for Multiprocessor Real-Time Locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 49–60, December 2010.

[18] Reinder J Bril, Johan J Lukkien, and Rudolf H Mak. Best-case response times and jitter analysis of real-time tasks with arbitrary deadlines. In *Proceedings of the 21st International conference on Real-Time*

Networks and Systems, pages 193–202. ACM, 2013.

[19] Giorgio Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[20] Darren Buttle. Real-time in the prime-time. In *Keynote speech at the 24th Euromicro Conference on Real-Time Systems*, Pisa, Italy, July 12, 2012.

[21] Thidapat Chantem, Robert P. Dick, and X. Sharon Hu. Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs. In *DATE*, pages 288–293, 2008.

[22] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13 (4s):124:1–124:29, April 2014. ISSN 1539-9087. doi: 10.1145/2584654. URL http://doi.acm.org/10.1145/2584654.

[23] Franois Chollet. Keras, 2015. URL https://github.com/fchollet/keras.

[24] Jin Cui and D.L. Maskell. A Fast High-Level Event-Driven Thermal Estimator for Dynamic Thermal Aware Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6): 904–917, 2012.

[25] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89 Proceedings, Lecture Notes in Computer Science*, volume 435, pages 416–427. Springer-Verlag, 1989.

[26] Robert I Davis, Ken W Tindell, and Alan Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 222–231. IEEE, 1993.

[27] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007. ISSN 1573-1383. doi: 10.1007/s11241-007-9012-7. URL http://dx.doi.org/10.1007/s11241-007-9012-7.

[28] Robert I. Davis, Timo Feld, Victor Pollex, and Frank Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *Proc. 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, Berlin, Germany, April 2014.

[29] S.K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operation Research*, pages 127–140, 1978.

[30] M. Di Natale and Haibo Zeng. Efficient Implementation of AUTOSAR Components with Minimal Memory Usage. june 2012.

[31] Jonas Diemer, Daniel Thiele, and Rolf Ernst. Formal worst-case timing analysis of ethernet topologies with strict-priority and avb switching. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES12)*, jun 2012. URL http://dx.doi.org/10.1109/SIES.2012.6356564. Invited Paper.

[32] Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015.

[33] Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

[34] D Eastlake and Tony Hansen. Us secure hash algorithms (sha and hmac-sha). Technical report, RFC 4634, July, 2006.

[35] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[36] Hamid Reza Faragardi, Björn Lisper, and Thomas Nolte. Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores. In *Emerging Technologies and Factory Automation (ETFA)*, pages 1–5, 2013.

[37] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380389. URL http://doi.acm.org/10.1145/2380356.2380389.

[38] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip. In *Proc. 22nd IEEE Real-Time Systems Symposium*,

2001.

[39] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for hard real-time systems using multi-core COTS. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[40] Franz-Josef Götz. Alternative shaper for scheduled traffic in time sensitive networks. In *IEEE 802.1 TSN TG Meeting*, 2013.

[41] J.J. Gutiérrez, J.C. Palencia, and M. González Harbour. Response Time Analysis in AFDX Networks. In *XIV Jornadas de Tiempo Real*, February 2011.

[42] Zain AH Hammadeh, Sophie Quinton, and Rolf Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *Proceedings of the 14th International Conference on Embedded Software*, page 10. ACM, 2014.

[43] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. *System Level Performance Analysis - the SymTA/S Approach*, volume System-on-Chip: Next Generation Electronics, chapter 2, pages 29–72. The Institution of Electrical Engineers, London, United Kingdom, 2006.

[44] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 1997.

[45] IBM. Ibm ilog cplex 12.4, 2011. http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/.

[46] IEEE Time-Sensitive Networking Task Group. 802.1Qbr - Enhancements for Scheduled Traffic. http://www.ieee802.org/1/pages/802.1.html, .

[47] IEEE Time-Sensitive Networking Task Group. 802.1Qbu - Enhancements for Scheduled Traffic. http://www.ieee802.org/1/pages/802.1.html, .

[48] IEEE Time-Sensitive Networking Task Group. IEEE Time-Sensitive Networking Task Group. http://www.ieee802.org/1/pages/tsn.html, .

[49] J. Imtiaz, J. Jasperneite, and L. Han. A performance study of Ethernet Audio Video Bridging (AVB) for Industrial real-time communication. In *IEEE Conference on Emerging Technologies Factory Automation*, 2009.

[50] John Kelsey, Shu jen Chang, and Ray Perlner. SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. Technical report, National Institute of Standards and Technology (NIST), December 2016.

[51] Jung-Eun Kim, Tarek F. Abdelzaher, and Lui Sha. Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 221–231, 2015.

[52] Pranaw Kumar and Lothar Thiele. Quantifying the effect of rare timing events with settling-time and overshoot. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 149–160, 2012.

[53] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, volume 90, pages 201–209, 1990.

[54] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. In *IEEE Transactions on Computers*, page 44(12):1429–1442, 1995.

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

[56] Yongpan Liu et al. Accurate Temperature-Dependent Integrated Circuit Leakage Power Estimation is Easy. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1526–1531, 2007.

[57] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 12–21, 1999. doi: 10.1109/REAL.1999.818824.

[58] M. M Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. In *IEEE Transactions on Computers*, 1995.

[59] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[60] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, Washington, D.C., August 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti.

[61] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2), June 1970.

[62] R.C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford, 1979.

[63] R.C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89 Proceedings, Lecture Notes in Computer Science*, volume 435, pages 218–238. Springer-Verlag, 1989.

[64] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource software on multicore automotive ecus - combining runnable sequencing with task scheduling. In *IEEE Transactions on Industrial Electronics*, page 59(10):3934–3942, 2012.

[65] Syeda Iffat Naqvi and Adeel Akram. Pseudo-random key generation for secure hmac-md5. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 573–577. IEEE, 2011.

[66] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. doi: 10.1109/EDCC.2012.27.

[67] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. In *Real-Time Systems*, page 9(3):207–239, 1995.

[68] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 741–746, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL http://dl.acm.org/citation.cfm?id=1870926.1871105.

[69] Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 515–520. EDA Consortium, 2012.

[70] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086713. URL http://doi.acm.org/10.1145/2086696.2086713.

[71] RapiTime Systems Ltd. *www.rapitasystems.com*.

[72] Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 165–172. IEEE, 2002.

[73] Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Worst-case temperature guarantees for real-time applications on multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 87–96. IEEE, 2012.

[74] L Sha, R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real Time Cache Management Framework for Multi-core Architectures. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[75] L Sha, R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. In *Tech Report, University of Illinois at Urbana Champaign, available at http://hdl.handle.net/2142/55672*, 2013.

[76] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[77] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[78] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, pages 2–, 2003.

[79] Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):30:1–30:39, 2008.

[80] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Trans. Architec. Code Optim.*, 1(1):94–125, 2004. ISSN 1544-3566.

[81] The AUTOSAR consortium. The AUTOSAR standard, specification version 4.3, 2017.

http://www.autosar.org.

[82] Daniel Thiele and Rolf Ernst. Formal analysis based evaluation of software defined networking for time-sensitive ethernet. In *Design Automation and Test in Europe (DATE)*, Dresden, Germany, March 2016.

[83] Daniel Thiele and Rolf Ernst. Formal worst-case timing analysis of ethernet tsn's burst-limiting shaper. In *Design Automation and Test in Europe (DATE)*, Dresden, Germany, March 2016.

[84] Daniel Thiele and Rolf Ernst. Formal worst-case performance analysis of time-sensitive ethernet with frame preemption. In *Proceedings of Emerging Technologies and Factory Automation (ETFA)*, page 9, Berlin, Germany, September 2016. Best Paper Award.

[85] Daniel Thiele, Philip Axer, Rolf Ernst, and Jan R. Seyler. Improving formal timing analysis of switched ethernet by exploiting traffic stream correlations. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New Delhi, India, October 2014. URL `http://dx.doi.org/10.1145/2656075.2656090`.

[86] Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In *Design Automation Conference (DAC)*, San Francisco, CA, USA, June 2015.

[87] Daniel Thiele, Rolf Ernst, and Jonas Diemer. Formal worst-case timing analysis of ethernet tsn's time-aware and peristaltic shapers. In *Vehicular Networking Conference (VNC)*, Kyoto, Japan, December 2015.

[88] A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS 2013)*, pages 45–56, December 2013.

[89] A. Wieder and B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, pages 49–58, June 2013.

[90] Yuan Xie and Wei-lun Hung. Temperature-Aware Task Allocation and Scheduling for Embedded Multi-processor Systems-on-Chip (MPSoC) Design. *The Journal of VLSI Signal Processing*, 45(3):177–189, 2006.

[91] Wenbo Xu, Zain AH Hammadeh, Alexander Kroller, Rolf Ernst, and Sophie Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 247–256. IEEE, 2015.

[92] Q. Zhu, H. Zeng, W. Zheng, M. Di Natale, and A Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. In *ACM Transactions on Embedded Computing Systems (TECS)*, volume 11 (4), 85, 2012.