Resource Management for Real-Time and Mixed-Critical Systems

Gautam Jayantilal Gala

Dissertation 2021

Resource Management for
Real-Time and Mixed-Critical Systems

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Gautam Jayantilal Gala
geboren in Gandhidham, Indien.

D 386

Eingereicht am:	13.10.2021
Tag der mündlichen Prüfung:	22.11.2021
Dekan des Fachbereichs:	Prof. Dr. rer. nat Marco Rahm
Promotionskomission	
Vorsitzender:	Prof. Dr.-ing. Wolfgang Kunz
Berichterstattende:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
	Prof. Dr. Johan Eker

Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

(Kaiserslautern, 13.10.2021) (Gautam Jayantilal Gala)

Abstract

Multicore processors and Multi-Processor System on Chips (MPSoCs) have become essential in Real-Time Systems (RTS) and Mixed-Critical Systems (MCS) because of their additional computing capabilities that help reduce Size, Weight and Power (SWaP), required wiring, and associated costs. In distributed systems, a single shared multicore or MPSoC node executes several applications, possibly of different criticality levels. However, there is interference between applications due to contention in shared resources such as CPU core, cache, memory, and network. Existing allocation and scheduling methods for RTS and MCS often rely on implicit assumptions of the constant availability of individual resources, especially the CPU, to provide guaranteed progress of tasks. Most existing approaches aim to resolve contention in only a specific shared resource or a set of specific shared resources. Moreover, they handle a limited number of events such as task arrivals and task completions. In distributed RTS and MCS with several nodes, each having multiple resources, if the applications, resource availability, or system configurations change, obtaining assumptions about resources becomes complicated. Thus, it is challenging to meet end-to-end constraints by considering each node, resource, or application individually.

Such RTS and MCS need global resource management to coordinate and dynamically adapt system-wide allocation of resources. In addition, the resource management can dynamically adapt applications to changing availability of resources and maintains a system-wide (global) view of resources and applications. The overall aim of global resource management is twofold. Firstly, it must ensure real-time applications meet their end-to-end deadlines even in the presence of faults and changing environmental conditions. Secondly, it must provide efficient resource utilization to improve the Quality of Service (QoS) of co-executing Best-Effort (BE) (or non-critical) applications.

A single fault in global resource management can render it useless. In the worst case, the resource management can make faulty decisions leading to a deadline miss in real-time applications. With the advent of Industry 4.0, cloud computing, and Internet of Things (IoT), it has become essential to combine stringent real-time constraints and reliability requirements with the need for an open-world assumption and ensure that the global resource management does not become an inviting target for attackers.

In this dissertation, we propose a domain-independent global resource management framework for distributed MCS and RTS consisting of heterogeneous nodes based on multicore processors or MPSoCs. We initially developed the framework with the French

Aerospace Lab – ONERA and Thales Research & Technology during the DREAMS project and later extended it during SECREDAS and other internal projects. Unlike previous resource management frameworks MCS and RTS, we consider both safety and security for the framework itself. To enable real-time industries to use cloud computing and enter a new market segment – real-time operation as a cloud-based service, we propose a Real-Time Cloud (RT-Cloud) based on global resource management for hosting RTS and MCS.

Finally, we present a mixed-criticality avionics use case for evaluating the capabilities of the global resource management framework in handling permanent core failures and temporal overload condition, and a railway use case to motivate the use of RT-Clouds with global resource management.

"Two roads diverged in a wood, and I–
I took the one less traveled by,
And that has made all the difference."

\- Robert Frost

Acknowledgments

Reading acknowledgments in countless books and dissertations, I never apprehended how difficult a task it would be for finding the right words to say "thank you". There have been so many people without whose help, support, and guidance I would have never completed this dissertation. I would have never been able to accomplish this on my own. It took a large and disparate community consisting of family, friends, academics, and industrialists throughout the world to help me through this journey. I only fear that I may commit the error of under-representation or omission.

First and foremost, I would like to thank my parents (Manisha and Jayantilal Gala), aunt (Harsha Vora-Naghashian), uncle (Turadj Naghashian), sister (Aarti Gala), and cousin (Adib Naghashian). Without their support, I would have never even come close to reaching this stage of my life. I also want to acknowledge Karishma Shah, who stood by me through my travails in the past three years. I am deeply thankful for her support, patience, and encouragement whenever my morale was down.

The person I appreciate the most for this dissertation is my research supervisor, Prof. Dipl.-Ing. Dr. Gerhard Fohler. I am grateful for the opportunity to pursue a Ph.D. Without his guidance, inspiration, and support, I would have never completed this dissertation. I also want to thank him for his understanding over these past years and begin a friend and mentor at appropriate moments.

I want to thank my committee members, Prof. Dr. Johan Eker and Prof. Dr.-ing. Wolfgang Kunz (TU Kaiserslautern). Unfortunately, I have not had the opportunity to work with Prof. Eker directly; however, the impact of his work on my research is evident in this dissertation. I took Prof. Kunz's lectures during my master's at TU University. His teaching style and enthusiasm for the topic made a strong impression on me. I have always carried positive memories of his lectures.

Prof. Fohler gave me the opportunity to participate in European projects[1]. As apparent throughout this dissertation, these projects had a strong positive influence on my research and dissertation. I express my sincere thanks to the European Union and BMBF for funding these projects. The projects facilitated me to collaborate and interact with academics and industrialists throughout Europe. I always look forward to conferences and project meetings as I can meet many interesting people and visit new

[1] The work presented in this dissertation has partly received funding from the European FP7-ICT project DREAMS under reference n°610640 and ECSEL-JU H2020-EU.2.1.1.7. project SECREDAS under reference n°783119.

places. There are countless people from these two projects that have assisted me in my research and shared interesting thoughts from different viewpoints. I would especially like to thank Dr. Daniel Gracia Pérez (Thales R & T), Dr. Claire Pagetti (ONERA), Prof. Dr. Roman Obermaisser (University of Siegen), Prof. Dr. Alfons Crespo (UPV), Dr. Peter Tummeltshammer (Thales Austria), Dr. Stefan Resch (Thales Austria), Dr. Thomas Koller, Dr. Hamidreza Ahmadian, Javier Coronel, Marine Kadar (Sysgo), Arjan Geven (TTTech), and Dr. Michael Meidlinger. A special appreciation goes to Prof. Roman Obermaisser for his efforts in coordinating the DREAMS project. I also want to acknowledge the support for XtratuM Hypervisor from Prof. Alfons Crespo and the team at fentISS.

The Chair of Real-Time Systems at TU Kaiserslautern was a second home for the past few years. Prof. Fohler and my colleagues provided a safe and dynamic place to study, discuss, and research. I express my sincere thanks to Stephanie "Steffi" Jung and Markus Müller. Steffi always helped me navigate the jungle of bureaucracy proactively and with a smile. Without her support, it would have been laborious and, at times, nearly impossible to solve expected and unexpected bureaucratic issues promptly. Markus is the IT wizard of the chair. Like a wizard, he is neither later nor early; he is precisely there when I need his help with my IT-related issues. I also want to thank him for maintaining the immaculate version control and backup system at the chair, which has saved my day more than once.

I want to thank Javier Castillo and Carlos Rodriguez for their collaboration over the last year. I also want to acknowledge my other colleagues, students and alumni from the Chair of Real-Time Systems: Florian Heilmann, Ibrahim Alkoudsi, Kristin Kruger, Luiz Maia Neto, Ali Syed, Rodrigo Coelho, Stefan Schorr, Gabriele Monaco, Giorgio Farina, and Isser Kadusale. Thank you all for the fruitful discussions, insights, and reviews.

Last but not least, I would like to acknowledge my friends and other family members: Gautam and Shruti Sawala, Rahul and Priyanka Dahiwal, Rohan Gugale, Nabajeet Barman, Pramod Murthy, Pratik and Neha Shah, Manjeet Singh, and Patrick Agostini. I thank them for exciting adventures that made this journey enjoyable. I also like to thank my flight instructors at Flugsportverein Kaiserslautern: Hermann and Michael Bolz, Otto, and Dieter Zimmerman. The flying lessons helped me immensely to relax my mind. Other family members, friends, colleagues, and students with whom I have interacted over the years are too numerous to list. But rest assured, you all have a place in my heart.

Gautam Jayantilal Gala
Thursday 9th December, 2021

Publications

I have co-authored the following publications:

Book Chapters

1. G. Gala, G. Fohler, D. G. Pérez, and C. Pagetti, “Resource management services,” in *Distributed Real-Time Architecture for Mixed-Criticality Systems* (H. Ahmadian, R. Obermaisser, and J. Perez, eds.), ch. 9, pp. 377–402, CRC Press, 2018

2. R. Obermaisser, M. Abuteir, H. Ahmadian, P. Balbastre, S. Barner, M. Coppola, J. Coronel, A. Crespo, G. Fohler, G. Gala, M. Grammatikakis, A. Larrucea Ortube, T. Koller, Z. Owda, and D. Weber, “Architectural style,” in *Distributed Real-Time Architecture for Mixed-Criticality Systems* (H. Ahmadian, R. Obermaisser, and J. Perez, eds.), ch. 2, pp. 7–78, CRC Press, 2018

Conference and refereed workshop papers

1. G. Gala, J. Castillo Rivera, and G. Fohler, “Work-in-progress: Cloud computing for time-triggered safety-critical systems,” in *IEEE Real-Time Systems Symposium (RTSS)*, IEEE TCRTS, 2021

2. G. Gala and G. Fohler, “Safe and secure global resource management for real-time and mixed-criticality systems,” in *2021 IEEE International Conference on Computing (ICOCO)*, IEEE, 2021. Best paper award

3. G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, “RT-cloud: Virtualization technologies and cloud computing for railway use-case,” in *24th IEEE International Symposium On Real-Time distributed Computing (IEEE ISORC)*, IEEE, 2021

4. G. Gala and G. Fohler, “Distributed decision-making for safe and secure global resource management via blockchain: Work-in-progress,” in *2020 International Conference on Embedded Software (EMSOFT)*, pp. 28–30, IEEE, 2020

5. G. Fohler, G. Gala, D. G. Pérez, and C. Pagetti, “Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator,” in *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), (Toulouse, France), Jan. 2018
6. G. Gala, T. Koller, D. G. Pérez, G. Fohler, and C. Ruland, “Timing analysis of secure communication between resource managers in dreams,” in *Workshop on Security and Dependability of Critical Embedded Real-Time Systems in conjunction with RTSS*, 2016
7. T. Koller, G. Gala, D. G. Pérez, C. Ruland, and G. Fohler, “Dreams: Secure communication between resource management components in networked multi-core systems,” in *2016 IEEE Conference on Open Systems (ICOS)*, 2016. Best paper award
8. G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, “DREAMS about reconfiguration and adaptation in avionics,” in *ERTS 2016*, (Toulouse, France), Jan. 2016

Technical Reports

- G. Gala (eds.), G. Fohler (eds.), S. Barner, A. Diewald, R. Obermaisser, T. Koller, D. G. Pérez, B. Nikolic, M. Coppola, M. Grammatikakis, A. Crespo, J. Coronel, K. Chappuis, G. Bouwer, G. Klaes, J. Perez, and A. Larrucea, “White paper on mixed-criticality research and innovation,” DREAMS project, Sept. 2017. Available: `https://cordis.europa.eu/docs/projects/cnect/0/610640/080/deliverables/001-DREAMSD932WhitepaperonmixedcriticalityresearchandinnovationR10.pdf`

Contents

Acknowledgments vii

Publications ix

I Introduction 1

I.1 Scope of the Dissertation . . . 3
I.2 Document Structure . . . 4

II Background, Related Work and Problem Statements 7

II.1 Distributed System . . . 7
II.1.1 Federated and Integrated Architectures . . . 7
II.2 Multicore processor . . . 9
II.2.1 Cache . . . 9
II.2.2 On-Chip (Shared) Interconnect . . . 10
II.2.3 Integrated Memory Controller (IMC) . . . 11
II.3 System on a Chip (SoC) . . . 11
II.4 Examples of Multicore processors and SoCs . . . 12
II.4.1 NXP QorIQ T4240 with e6500 processor . . . 12
II.4.2 Xilinx ZC706 with Zynq-7000 SoC . . . 13
II.4.3 Dell R640 Server with Intel Xeon Gold 5218 Processor . . . 14
II.5 Real-Time Systems (RTS) . . . 15
II.6 Task . . . 16
II.6.1 Task parameters . . . 16
II.7 Task models . . . 17
II.8 Mixed-criticality Systems (MCS) . . . 18
II.9 Scheduling . . . 19
II.10 Timebase . . . 20
II.11 TT and ET Task Schedulers . . . 21
II.12 Time Triggered Architecture (TTA) . . . 22
II.13 TT Ethernet (TTE) . . . 23
II.13.1 ARINC 653 . . . 24

II.14 Shared Resource Contention 27
II.15 Existing Approaches for Shared Resource Contention 29
II.16 Modes 32
II.17 Virtualization 34
II.17.1 Hypervisors 34
II.17.2 Virtualization techniques 34
II.17.3 Hypervisors for Real-time and Mixed-Criticality Systems 36
II.17.4 XtratuM Hypervisor 37
II.17.5 Hierarchical Scheduling in Hypervisors 39
II.18 Cloud Computing 39
II.18.1 Public vs. Private Clouds 40
II.19 Fault, Error, Failure 40
II.20 Fault-tolerance 40
II.20.1 N-Modular Redundancy (NMR) 41
II.20.2 Reconfiguration for Fault-Tolerance 41
II.20.3 Byzantine fault tolerance (BFT) 42
II.21 Blockchain 43
II.22 Relevant EU Projects 44
II.23 DREAMS Project 46
II.23.1 Avionics Demonstrator 48
II.24 SECREDAS Project 48
II.24.1 Common Technology Elements (CTEs) 49
II.24.2 Railway Use Case 49
II.25 Problem Statements 50
II.25.1 Resource Management in Distributed RTS and MCS 50
II.25.2 Exploitation of Multicore Processors for Fault-Tolerance 51
II.25.3 Simultaneous Execution of Mixed-Criticality Applications 52
II.25.4 Safety-Critical Operation as a Cloud-Based Service 53
II.25.5 Open-world Assumptions: Safety- and Security-Aware Approaches 54
II.26 Relevant Resource Management Frameworks 55
II.26.1 Matrix Resource Management Framework 55
II.26.2 ACTORS Project Resource Management Framework 55
II.26.3 Real-Time Adaptive Resource Management (RTARM) 56
II.26.4 Resource Allocation and Control Engine (RACE) Framework . . 57
II.26.5 ACROSS Project Resource Management Framework 57
II.26.6 HiDRA 58
II.26.7 RT-CORBA 58
II.26.8 Game-Theoretic Resource Management Framework 59
II.26.9 Resource Manager for Guaranteeing End-to-End Deadlines 60
II.26.10 Miscellaneous 60
II.27 Contributions 61

II.27.1 Global Resource Management Framework for Real-Time and Mixed-Criticality Distributed System 61
II.27.2 Exploitation of Multicore Processors for Fault-Tolerance (in Avionics) 62
II.27.3 Simultaneous Execution of Mixed-Criticality Applications (in Avionics) 62
II.27.4 Safety-Critical Operation as a Cloud-Based Service 62
II.27.5 Railway Operation as a Cloud-Based Service 63
II.27.6 Safety- and Security-Aware Global Resource Management 63

III Resource Management Framework 65

III.1 System Structure 65
III.2 Resource Management Framework 65
III.3 Low Overhead Resource Management 66
III.4 Adapting to Fluctuations 67
III.4.1 Adapting to Failures in System Resources 68
III.5 Modular Monitoring and Scheduling of Resources 68
III.6 Tackling Heterogeneous Nodes, Complex System Structures, and Scalability 70
III.6.1 Effective System reconfiguration 70
III.7 Dynamic addition or modification of applications 74
III.8 Safety and Security for Resource Management 75
III.9 Configuring Platform-Specific Components 75

IV Local Resource Management 79

IV.1 Monitoring Services 79
IV.2 Monitoring opportunities 81
IV.2.1 Hardware Performance Monitoring Counters 81
IV.2.2 Intel CMT and MBM Monitoring Technologies 84
IV.2.3 XtratuM Hypervisor 84
IV.2.4 Linux/Kernel Virtual Machine (KVM) 84
IV.3 Existing Monitoring Approaches for RTS and MCS 85
IV.4 New Monitors (MONs) for RTS and MCS 86
IV.4.1 Hardware MONs 86
IV.4.2 Hardware MONs for Intel CMT and MBM 88
IV.4.3 Core Failure MON 88
IV.4.4 Deadline Overrun Local Resource Monitor (MON) 89
IV.4.5 Health MONs (XtratuM) 91
IV.5 Local Resource Scheduler Services 92
IV.6 Scheduling Opportunities 92
IV.6.1 XtratuM Hypervisor 92
IV.6.2 Linux/KVM 93

IV.6.3 Intel CAT and MBA 93
IV.6.4 Time-Triggered (TT)-Network-on-Chip (NoC) Scheduler 93
IV.6.5 Existing Approaches for Scheduling RTS and MCS 94
IV.7 Newly Implemented Local Resource Schedulers (LRSs) 95
IV.7.1 LRS for Critical Partitions (XtratuM) 95
IV.7.2 LRS for Online Reconfiguration (XtratuM) 96
IV.7.3 LRS for Intel CAT and MBA 96
IV.7.4 TT LRS for Linux/KVM 97
IV.8 Local Resource Management Services 99
IV.9 Local Resource Management Policies 103
IV.9.1 Core Failure Management 103
IV.9.2 Temporal Overload Management 106
IV.10 Chapter Summary 110

V Global Resource Management 113

V.1 Global Resource Manager (GRM) Implementation Options 115
V.2 Global Resource Management Communication 116
V.2.1 Update Channel 117
V.2.2 Order Channel 118
V.2.3 Membership Channel 118
V.3 Core failure Scenario Example 118
V.4 Resource Management Security 121
V.4.1 Security Services 122
V.4.2 Security Levels 123
V.4.3 Security Levels for Resource Management Channels 124
V.4.4 Implementation of Security Services 125
V.5 Using Existing Protocols For Resource Management Communication . . 127
V.5.1 MQTT and OPC UA 127
V.6 Distributed Decision-Making 131
V.6.1 Distributed Global Resource Manager (DGRM) 132
V.7 Blockchain-Based Global Resource Management 135
V.7.1 Resource Manager (RM) Transaction Family 135
V.7.2 RM Transaction 139
V.8 Chapter Summary 140

**VI Avionics Use Case:
Resource Management for Distributed Mixed-Critical System (MCS) 143**

VI.1 Avionics Use Case 144
VI.2 Use Case Demonstrator 146
VI.3 Resource Management Implementation 146
VI.3.1 LRSs 147
VI.3.2 MONs 147

VI.3.3 Local Resource Managers (LRMs) . . . 149
VI.3.4 GRM . . . 150
VI.3.5 Resource Management Communication . . . 151
VI.4 Reconfiguration Strategy for Core Failure . . . 154
VI.4.1 Application Model . . . 154
VI.4.2 Local Configuration . . . 155
VI.4.3 Global Configuration . . . 156
VI.4.4 Reconfiguration Requirements . . . 156
VI.4.5 Local Reconfiguration Graphs . . . 157
VI.4.6 Global Reconfiguration Graph . . . 157
VI.4.7 Example of Reconfiguration Graphs . . . 158
VI.4.8 Network reconfiguration . . . 162
VI.5 Resource Management Evaluation . . . 164
VI.5.1 Evaluation of Local and Global Core Failure Management . . . 164
VI.5.2 Evaluation of Local Temporal Overload Management . . . 168
VI.5.3 Evaluation of Secure Resource Management Communication . . . 170

VII Railway Use Case: Real-Time Cloud via Resource Management 175

VII.1 Railway Use Case . . . 175
VII.1.1 TAS Control Platform . . . 178
VII.1.2 TAS Control Platform in Cloud . . . 179
VII.1.3 Safety Requirements for TAS Virtual Machines (VMs) . . . 180
VII.2 RT-Cloud Component Selection – Hypervisor . . . 181
VII.2.1 Qualitative Analysis . . . 182
VII.3 Quantitative analysis . . . 184
VII.4 Demonstration . . . 192
VII.5 Resource Management Layer for RT-Cloud . . . 193
VII.6 TT Scheduling on Cloud Nodes . . . 197
VII.6.1 TT-LRS Evaluation . . . 198
VII.7 Intel Memory Bandwidth Allocation (MBA) Evaluation . . . 203
VII.8 DGRM Evaluation . . . 205
VII.8.1 Discussion . . . 209

VIII Conclusion 211

VIII.1 Overview of Contributions . . . 212
VIII.1.1 Resource Management Framework for MCS and Real-Time System (RTS) . . . 212
VIII.1.2 Local Resource Monitor (MON) . . . 213
VIII.1.3 Local Resource Scheduler (LRS) . . . 213
VIII.1.4 Resource Management Policies . . . 214
VIII.1.5 Resource Management Communication . . . 216
VIII.1.6 Blockchain-Based Distributed Global Resource Management . . . 217

VIII.1.7 Safety-Critical Railway Operation as a Cloud-Based Service . . 218
VIII.1.8 Time-Triggered (TT) Scheduling in Cloud Nodes 219
VIII.2 Ongoing and Future Work 220
VIII.2.1 Adding Flexibility to TT Scheduling on RT-Cloud Nodes 220
VIII.2.2 Evaluation of Global Resource Management in RT-Cloud . . . 220
VIII.2.3 Energy-aware TT-ET Joint Scheduling in RT-Cloud Nodes . . . 221
VIII.2.4 Memory bandwidth regulation on Intel Xeon Processors 221
VIII.2.5 Simultaneously Execution of Critical Applications 221
VIII.2.6 Application State and Fault-Tolerance 222
VIII.2.7 Peripherals in Mixed-Critical System (MCS) 222
VIII.2.8 Blockchain-Based DGRM with Industrial Use Case 222
VIII.2.9 Resource Management Communication with MQTT and OPC UA 222

A Configuration File Examples 223

B Examples of Useful PMU events 231

C Pseudo-Code for MONs, LRSs, LRM, and GRM 237
C.1 MONitors (MONs) 237
C.1.1 Hardware MONs 237
C.1.2 Core Failure MON 239
C.1.3 Deadline Overrun MON 240
C.1.4 Health MONs (XtratuM) 241
C.2 Local Resource Schedulers (LRSs) 242
C.2.1 LRS for Critical Partitions (XtratuM) 242
C.2.2 LRS for Online Reconfiguration (XtratuM) 244
C.2.3 LRS for Intel CAT and MBA 244
C.2.4 TT LRS for Linux/KVM 245
C.3 Local Resource Manager (LRM) 248
C.4 Global Resource Manager (GRM) 249

D Avionics Use Case-Specific Pseudo-Code of GRM and LRM 251

E Time-Triggered (TT) Scheduling for Cloud Nodes 255
E.1 Heuristic for Generation of a TT Scheduling Table 255

Bibliography 261

Acronyms 280

Summary 281

Zusammenfassung 287

Curriculum Vitae 297

List of Figures

II.F1 Example of a distributed system 8
II.F2 Examples of Federated and Integrated Architectures 8
II.F3 Example of a Multicore Processor 9
II.F4 NXP QorIQ T4240 [12] 13
II.F5 Xilinx Zynq-7000 [13] 14
II.F6 Intel Xeon Gold 5218 Processor 15
II.F7 Example of TT-Ethernet (TTE) Schedules 24
II.F8 ARINC 653 Minor Frame (MiF) and Major Frame (MaF) 25
II.F9 Example of Modes 33
II.F10 Type-1 (Bare metal) Hypervisor 35
II.F11 Hierarchical Scheduling in Avionics 39
II.F12 DREAMS Architecture [2] 47

III.F1 System Structure 66
III.F2 Basic Architecture of the Resource Management Framework 67
III.F3 Modular LRM design 69
III.F4 Examples of Flat and Hierarchical Architectures 71
III.F5 Resource Management Domains 73
III.F6 Configuration of Resource Management Components 77

IV.F1 Classification of Performance Monitor Units (PMUs) 81
IV.F2 Hardware MON for XtratuM Partitions 87
IV.F3 Core failure MON 89
IV.F4 Deadline Overrun MON Location 90
IV.F5 Deadline Overrun MON 91
IV.F6 Intel CAT and MBA 94
IV.F7 LRS for Critical Partitions 96
IV.F8 LRS for Intel CAT and MBA 97
IV.F9 TT LRS for Linux/KVM 98
IV.F10 Implementation of LRMs in Different Domains 100
IV.F11 LRM services 101
IV.F12 Example of Resource Abstraction by LRM 102
IV.F13 Core Failure Management by LRM 105
IV.F14 Deadline Overrun Management by LRM 108

IV.F15 QoS Management by LRM 109

V.F1 Global Resource Manager (GRM) 115
V.F2 Implementation Options for the GRM 116
V.F3 Conceptual Resource Management Communication Channels 117
V.F4 Global Resource Management - Core failure Scenario 120
V.F5 Implementation of Resource Management Security Services 125
V.F6 Resource Management Communication using Existing Communication Protocols 130
V.F7 Distributed Global Resource Manager (DGRM) 133
V.F8 RM Transaction Family for Sawtooth 137

VI.F1 Example of a Flight Display Panel 144
VI.F2 Applications of the Avionics Use Case 145
VI.F3 Avionics Use Case Demonstrator 147
VI.F4 Resource Management Components in the Avionics Demonstrator . . 148
VI.F5 Deadline Overrun MON in Avionics 149
VI.F6 LRM for Core Failure Management in Avionics 149
VI.F7 Potential Deadline Overrun Management in Avionics 150
VI.F8 Resource Management Communication Message Formats 152
VI.F9 Secure Message Frame 153
VI.F10 Example of Local Configuration Graph (Expanded) of Node N_1 (4 cores) 159
VI.F11 Examples of Local Reconfiguration Graphs (Condensed) 160
VI.F12 Example of the Global Reconfiguration Graph 161
VI.F13 Configuration of Resource Management Components in Avionics Use Case 163
VI.F14 Global Reconfiguration Delay 165
VI.F15 Resource Management Communication Overheads – DREAMS Harmonized platform (DHP) and T4240 173

VII.F1 Layered railway domain [5] 176
VII.F2 Layered Architecture of the *TAS Control Platform* [5] 179
VII.F3 TAS Control Platform with Virtualization 180
VII.F4 Cyclictest Bechmark 189
VII.F5 Sysbench CPU Benchmark 190
VII.F6 Sysbench Memory Read and Write Benchmark 191
VII.F7 iPerf3 Benchmark 192
VII.F8 Test Setup for the Radio Block Center (RBC) Application 193
VII.F9 Resource Management in Cloud 196
VII.F10 LRM and TT LRS 199
VII.F11 TT Dispatcher Overheads 199
VII.F12 Overheads for TT-LRS with No Migration Allowed 201
VII.F13 Overheads for TT-LRS with Migration Allowed 202

VII.F14 Impact of Memory Bandwidth Allocation (MBA) Delay Values on a Benchmark 205
VII.F15 Use Case Specific Resource Management Logic of the RM transaction family 208
VII.F16 Global Reconfiguration Delay 209

List of Tables

II.T1 Example of TTE Virtual Links (VLs) 24

IV.T1 Performance Monitor Counters (PMCs) per Core PMU 83
IV.T2 Memory Bandwidth Monitoring (MBM) and Cache Allocation Technology (CAT) properties Intel Xeon Gold 5218 94

V.T1 Resource Management Security Levels 123
V.T2 Default Security Levels for Resource Management Communication Channels 125
V.T3 Existing Communication Protocols for Resource Management 129

VI.T1 Avionics Use Case Application Parameters 145
VI.T2 Resource Management Component Implementation Summary 151
VI.T3 Resource Management Communication Implementation 152
VI.T4 Resource Management Communication Message Sizes 154
VI.T5 Global Reconfiguration Delay 164
VI.T6 Example of Local and Global Reconfiguration in Avionics Use Case . . 167
VI.T7 Resource Management Overhead 167
VI.T8 Evaluation of Local Temporal Overload Management 169
VI.T9 Performance of Non-Critical Applications in Deadline Overrun Scenario 170
VI.T10 DHP – Maximum Observed Time for Resource Management Communication 171
VI.T11 T4240 – Maximum Observed Time for Resource Management Communication 171

VII.T1 Comparison of Virtualization Technologies for Cloud Computing 183
VII.T2 Comparison of Existing Resource Management features in Virtualization Technologies 185
VII.T3 KVM vs. Xen - Summary of the Three Scenarios 187
VII.T4 KVM vs. Xen - Three Scenarios 187
VII.T5 Summary of Slot 197
VII.T6 Maximum Observed TT LRS Overhead (μs) 200

VIII.T1 Distributed vs. Central Global Resource Management 218

B.T1 Core PMU event examples of e6500 cores in T4240 232
B.T2 Core PMU events of Cortex-A9 (Zynq 7000 Multi-Processor System on a Chip (MPSoC)) 232
B.T3 APM (uncore PMU) events (Zynq 7000 MPSoC) 233
B.T4 Core PMU events of Intel Xeon scalable processors ($2^n d$ Gen.) 234
B.T5 Uncore PMU events of Intel Xeon scalable processors ($2^n d$ Gen.) 235

E.T1 Memory Bandwidth and Worst-case Execution Time (WCET) pairs for VM τ_n 257

I

Introduction

> *"A good problem statement often includes what is known, what is unknown, and what is sought."*
>
> – Edward Hodnett

Real-Time Systems (RTS), i.e., systems that must deliver the expected logical results within stringent timing constraints, are found in various domains such as multimedia, aerospace, railway, automotive, nuclear power plants, and healthcare. A substantial number of RTS are embedded systems where it is not instantly evident that a computer is involved. An illustration of an embedded RTS encountered in our daily life is a digital media player, where failing to meet deadlines can cause an undesirable lag. We refer to such RTS where failure to comply with real-time constraints causes loss of functionality or performance without catastrophic consequences as Non-Critical Real-Time Systems (NCRTS). Often, we put our lives in the hands of RTS without even realizing it and rely on their proper functioning; for example, failure to meet deadlines in an airplane's RTS can have catastrophic outcomes. We refer to RTS where failure to meet real-time constraints can lead to death or severe injury, loss or damage to property/equipment, or cause environmental harm as Safety-Critical real-time Systems (SCS). SCS must be certified by Certification Authorities (CAs) according to the appropriate industrial standards, which ensure their safe operation by reducing risks to appropriate assurance levels against failures.

Traditionally, many RTS have used federated architectures to host each real-time application on a dedicated hardware platform (called a node). The nodes exchange only control and sensor data among each other. The federated architectures ensure fault containment, limit errors in a node from propagating to other nodes, and avoid unwanted interactions by design. Thus each application can be certified in isolation from the other. However, the recent increase in the implemented applications in these domains has dramatically increased the number of nodes in the system, leading to an increase in Size, Weight and Power (SWaP), required wiring, and the associated costs. These considerations have prompted industries to move away from federated architectures and move to a new age of integrated architectures.

Integrated architectures implement several real-time applications on a single shared node. Integrated architectures have attained popularity as they counter the drawbacks

of federated architectures. Integrated architectures often use Commercial-Off-The-Shelf (COTS) multicore processors and Multi-Processor System on Chips (MPSoCs). Multicore processors and MPSoCs have multiple CPU cores on a single die, allowing to increase the performance and integration of more applications without the physical limitations of uniprocessors, thus contributing to a further reduction in SWaP, wiring, costs, and environmental footprint. The industry is shifting to multicores not only for their advantages but also because they expect mass-market obsolescence for single-core processors soon[14]. Another notable trend in integrated architectures is Mixed-Critical Systems (MCS). In MCS, applications of different criticality levels can execute simultaneously on a node and share the node's resources, e.g., the safety-criticality Flight Management System (FMS) and the non-critical In-Flight Entertainment (IFE) for the passengers can run in parallel on a single shared node.

The advantages of integrated architecture offer industries a compelling reason to use them. However, there are some drawbacks. It is difficult to achieve the required isolation, especially in multicore platforms and MPSoCs due to contention in the shared resources such as CPU, shared-bus, memory (controller), and network. These shared resources can cause unpredictable delays leading to deadline misses in real-time applications. Moreover, the boundaries for fault isolation and error contamination are not as sharply defined as in the federated architecture. As a result, ensuring that the RTS meet their deadline becomes challenging, especially in the absence of suitable resources management techniques for guaranteeing isolation and predictable shared resources access delays. Certifying Real-time safety-critical applications without these guarantees is cumbersome. There are also contrary goals for safety-critical and non-critical/best-effort applications that exacerbate the preexisting resource management problem in integrated architectures. The pessimistic Worst-case Execution Time (WCET) estimations of safety-critical applications under-utilize the resources significantly in the average case. Simultaneously, the non-critical/best-effort applications require efficient resource utilization to provide the best possible Quality of Service (QoS).

Many allocation and scheduling methods exist for RTS and MCS. These methods rely on implicit assumptions of constant availability of individual resources, especially the Central Processing Unit (CPU). Classical scheduling algorithms like Round-Robin (RR) or Earliest Deadline First (EDF) assume control over the entire CPU or a single core. Methods such as XtrautuM hypervisor [15] or PikeOS [16] schedule to fixed proportions and allow hierarchical scheduling. However, in these existing approaches availability of a constant amount of processing is assumed to provide guaranteed progress of tasks. Such pre-planned assumptions may be possible on single nodes; however, they become less meaningful in distributed systems with several nodes, each having multiple resources. If the applications, availability of resources, or system configurations change, obtaining assumptions about resources becomes complicated. Moreover, it is challenging to meet end-to-end constraints by considering each resource or node individually. A system with dynamically changing availability and requirement of resources requires ***global resource management*** to maintain a global (system-wide) view of resources and applications and coordinate and adapt system-wide resource allocations. In addition, the global resource management must adapt applications to changing resource availability.

Most existing resource management frameworks, such as [17, 18, 19, 20] focus on non-RTS. Some examples of existing approaches for resource management frameworks in RTS are the Matrix framework [21], the ACTORS framework [22], the ACROSS framework [23] and Real-Time Adaptive Resource Management (RTARM) [24]. The Matrix resource management framework presented a method to manage NCTRS using home networks and single-core CPUs as system resources. The resource management framework in ACTORS project provided temporal isolation in RTS at CPU-level through resource reservation in a single multicore platform. In Project ACROSS [23], a Trusted Resource Manager provided the possibility to reschedule communication on Network-on-Chip (NoC). RTARM is a adaptive resource management framework for end-to-end resource allocations on heterogeneous COTS nodes to provide guaranteed QoS to applications. However, these existing works have not considered an architecture consisting of distributed MCS and RTS consisting of heterogeneous nodes based on multicore processors or MPSoCs.

A single fault in the global resource management can render it useless. In the worst case, it can make faulty resource management decisions leading to a deadline miss in real-time applications. Thus, it is essential to ensure the safety of the global resource management itself by providing fault-tolerance in its components. Nevertheless, there can be no safety without security. With the advent of Industry 4.0, cloud computing, and Internet of Things (IoT), it has become essential to combine stringent real-time constraints and reliability requirements with the need for an open-world assumption. As a result, the global resource management for these systems becomes an inviting target for passive and active attackers as it can actively decide on the system's resource management. For instance, the attackers can masquerade as a resource management component and produce incorrect resource management decisions or obtain sensitive system information from resource management communication. None of the existing frameworks consider both, safety and security, together for the global resource management in RTS or MCS.

I.1 Scope of the Dissertation

This dissertation proposes a domain-independent global resource management framework for distributed MCS and RTS consisting of heterogeneous nodes based on multicore processors or MPSoCs. The global resource management framework can ensure efficient resource utilization while providing the required resource isolation and predictable resource access behavior to guarantee that all real-time applications meet their deadlines (and safety-critical applications adhere to their safety assurance levels). The framework also provides fault-tolerance or recovery for real-time applications upon changes in operational conditions or availability of resources. Simultaneously, the resource management can allocate resources to non-critical/best-effort applications to improve QoS. The global resource management framework presented in this dissertation considers both safety and security for the framework itself. To enable real-time industries to use cloud computing and enter a new market segment, e.g., *safety-critical operation as a cloud based service*, this dissertation extends the global resource management framework to develop a Real-Time Cloud (RT-Cloud) for hosting RTS and MCS. Finally, this dissertation presents a mixed-criticality avionics use case for evaluating the capabilities of the global resource

management framework in handling permanent core failures and temporal overload condition, and a railway use case to motivate the use of RT-Clouds with global resource management.

I.2 Document Structure

The rest of this dissertation is structured into seven chapters. The structure is as follows:

Chapter II describes the terms and concepts used throughout this dissertation. The chapter also provides the related work, relevant (EU) projects, and the state-of-the-art resource management frameworks for RTS and MCS. In addition, the chapter explains the problem statement of this dissertation and summarizes our contribution.

Chapter III presents our resource management framework for distributed MCS and RTS. The chapter explains the requirements and challenges in designing the resource management framework and our proposed solution to meet them.

Chapter IV explains the Local Resource Manager (LRM), a component that manages a subsystem and provides adaptability within the subsystem upon changes in availability and demands of resources. The chapter also explains in-depth the two main modular sub-components of the Local Resource Manager (LRM): Local Resource Monitor (MON) and Local Resource Scheduler (LRS). The chapter also introduces the newly implemented Local Resource Monitor (MONs) and Local Resource Schedulers (LRSs). Finally, this chapter explains the local resource management policies to manage permanent core failures on a node and potential deadline overrun in critical applications while improving the QoS of best-effort applications and overall resource utilization of a multicore node.

Chapter V explains the Global Resource Manager (GRM), a component that manages and provides adaptability within the entire system with the help of the LRMs that it controls and supervises. In addition, this chapter explains the introduced security measures for the communication among a single central Global Resource Manager (GRM) and the LRMs. Furthermore, this chapter extends this central GRM to make distributed global resource management decisions instead of centralized ones to increase fault tolerance. Finally, the chapter proposes to implement GRM for distributed decision-making using a blockchain to achieve Byzantine fault-tolerance for the global resource management decisions and security for the resource management communication.

Chapter VI presents the avionics use case from the DREAMS project. The chapter explains how we use and evaluate our resource management framework for system-wide adaptability upon core failures in an avionics system. The chapter also presents an evaluation of how resource management improves the QoS of best-effort applications and overall resource utilization of a multicore node while ensuring safety for the critical avionics applications. Lastly, the chapter provides an evaluation of the secure resource management communication.

Chapter VII explores virtualization technologies and cloud computing for migrating an existing real-time safety-critical railway use case from dedicated hardware solutions. The chapter examines existing virtualization technologies for deploying a (private) RT-Cloud on COTS server hardware to run an existing railway use-case while meeting stringent safety and security requirements. Based on the examination, the chapter

provides insight into using existing virtualization technologies the resource management architecture to safely and securely execute the railway use case applications. The chapter also presents the evaluation of the distributed global resource management in a cloud-based scenario. Finally, the chapter provides the evaluation of the Time-Triggered (TT)-LRS with a cloud node running Kernel Virtual Machine (KVM) hypervisor.

Chapter VIII provides the conclusion and the future work.

II

Background, Related Work and Problem Statements

"If I have seen further, it is by standing on the shoulders of Giants"
– Issac Newton

This chapter describes the terms and concepts used throughout this dissertation and provides an overview of the relevant works and projects.

II.1 Distributed System

There is no single definition for a distributed system. In general, a distributed system is a system where the software components run on multiple networked computer systems called *nodes* to achieve common objectives. Alternatively, each node can run software with individual goals, and the distributed system facilitates coordination for the use of shared resources and provides services for different software to communicate. Each node has its memory. They can be physically near each other and connected via a Local Area Network (LAN) or located at a distance and connected via a Wide Area Network (WAN). The software components can run on nodes with heterogeneous processors from various vendors, and they communicate and coordinate by passing messages between each other via the network. Figure II.F1 shows an example of a distributed system with five nodes.

II.1.1 Federated and Integrated Architectures

Federated architectures host each functionality on a dedicated node. The nodes exchange only control and sensor data among each other. The federated architectures ensure fault containment, limit errors in a node from propagating to other nodes, and avoid unwanted interactions by design.

However, the recent increase in the implemented functionalities in various domains has dramatically increased the number of nodes in the system, leading to an increase in Size, Weight and Power (SWaP), required wiring, and the associated costs. These

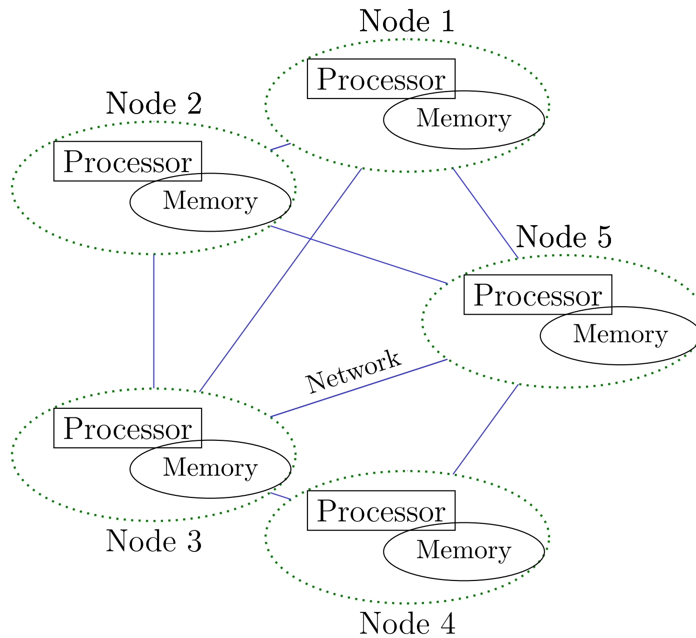
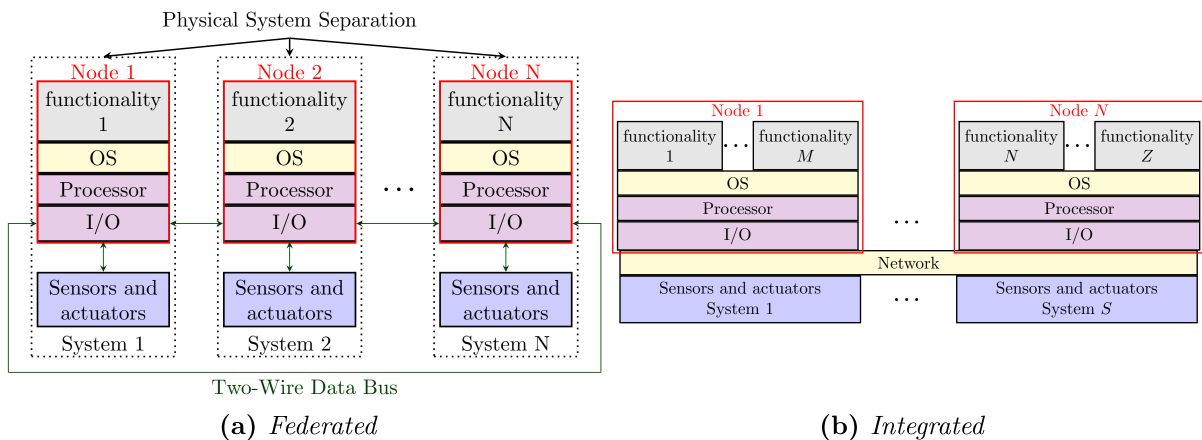


Figure II.F1: *Example of a distributed system*

considerations have prompted industries to move away from federated architectures to integrated architectures which implement several functionalities on a single shared node and counter the drawbacks of federated architectures. For example, Boeing and Airbus have reported a significant reduction in weight and the number of required processors due to the use of integrated architecture in the form of Integrated Modular Avionics (IMA) [25]. Figure II.F2 shows examples of federated and integrated architectures.

Integrated architectures often use Commercial-Off-The-Shelf (COTS) multicore processors or Multi-Processor System on Chips (MPSoCs) as they allow to increase the performance and integration of more functionalities without the physical limitations



(a) *Federated* **(b)** *Integrated*

Figure II.F2: *Examples of Federated and Integrated Architectures*

of uniprocessors, thus contributing to a further reduction in SWaP, wiring, costs, and environmental footprint. Furthermore, the industry is shifting to these processors not only for their advantages but also because they expect mass-market obsolescence for single-core processors soon[14]. Thus, in this dissertation, we consider integrated architectures consisting of nodes with multicore processors or MPSoCs. However, many of the ideas discussed in this dissertation are relevant to single-core processors as well.

II.2 Multicore processor

A multicore processor is a single integrated circuit consisting of two or more processing units called *cores* on the same die. The number of cores in a processor is usually to a power of two. For example, a quad-core processor has four cores. A homogenous multicore processor has cores with the same hardware architecture, while a heterogeneous multicore processor has cores with different hardware architecture. The processor can execute instructions parallelly on all its cores. In addition to cores, a multicore processor has other shared resources such as caches, on-chip interconnect(s), Integrated memory controller(s), and various peripheral I/O controllers. Figure II.F3 presents an example of a multicore processor.

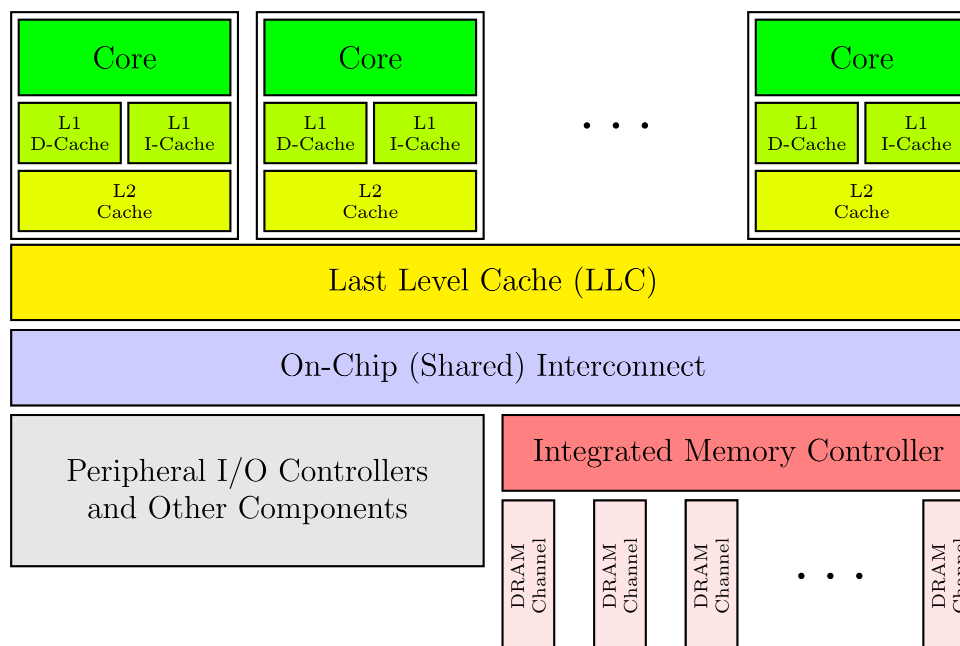


Figure II.F3: *Example of a Multicore Processor*

II.2.1 Cache

Caches are fast data storage components located close to the core and store copies of data from a slower data storage device: the main memory. They help to reduce the

average time needed to read data from the main memory. The caches are usually much smaller in size than the main memory to be cost-effective. Multicore processors often have a hierarchy of cache levels starting from *Level 1 (L1)* and usually going up to *L2* or *L3*. We refer to the cache at the last level in the hierarchy as *Last Level Cache (LLC)*. Caches generally increase in size and decrease in speed as we move along the hierarchy. L1 caches are the smallest and the closest to the cores. Many processors have separate instruction-specific and data-specific caches at Level 1, referred to as I-cache and D-cache, respectively. L1 caches are usually *private caches*, i.e., accessible from only one core, while LLC caches are usually *shared caches* and accessible by multiple cores.

There are two possible scenarios when a core accesses a cache:

1. A *cache hit* occurs if the core finds the requested data in a cache. Upon cache hit, the core directly uses the data from the cache and performs the required operation.

2. A *cache miss* occurs if the core does not find the requested data in the cache. Upon cache miss, the processor creates a new cache entry by copying the requested data from a higher level cache or the main memory to the current cache in a fixed-size block called the *cache line*. As a result, the core can now access the requested data from this new cache entry. When there is no more space in the cache to make new entries to serve cache misses, the cache evicts existing entries based on a *cache replacement policy* such as the Least Recently Used (LRU) or First In First Out (FIFO).

II.2.2 On-Chip (Shared) Interconnect

An on-chip interconnect consists of communication wires connecting the processor cores, caches, and memory controller. Depending on the hardware architecture, it may also connect additional components such as the peripheral I/O controllers and other buses such as the Peripheral Component Interconnect. In the past, many on-chip interconnects were shared bus-based architectures that connect multiple masters to multiple slaves. The use of hierarchical buses was a step forward in the on-chip interconnect architecture that facilitated complex multicore processor designs. Advanced Microcontroller Bus Architecture (AMBA) [26] is an example of a bus-based on-chip interconnect. It supports multiple buses such as the Advanced High-performance Bus (AHB) and Advanced eXtensible Interface (AXI) for high-performance and high clock frequency systems, and the Advanced Peripheral Bus (APB) for connecting low-bandwidth peripherals.

Crossbar is a non-blocking on-chip interconnect that eliminates serialization to a considerable extent. A crossbar has N inputs ports and M outputs ports and can simultaneously transport signals from the N input ports to the M output ports as far as the signals do not have identical source or destination ports. An example of crossbar architecture is the CoreNet Coherency Fabric (CCF) used in NXP T4240 [12].

A Point-to-Point (P2P) interconnect is another on-chip interconnect architecture that uses dedicated wires to connect the components. Intel QuickPath Interconnect (QPI) [27] is an example of a P2P interconnect.

Network-on-Chip (NoC) is an alternative interconnect architecture that provides better scalability than the other options while still supporting high bandwidth and is thus

suitable for designing multicore core processors with many cores. Routers are the basic building blocks of NoCs. Various network topologies, such as Mesh and Ring topologies, are possible for connecting them with each other and the network endpoints. [28] presents and compares some well-known NoC topologies. Spidergon STNoC [29] is an example of an NoC.

II.2.3 Integrated Memory Controller (IMC)

The Integrated Memory Controller (IMC) is an integral part of the multiprocessor and acts as an interface between the cores and the main memory. Main memory usually consists of Dynamic Random-Access Memory (DRAM). DRAMs store data as a presence or absence of charge on a capacitor. Since the capacitors slowly leak electric charge, DRAMs need an external circuit to perform a memory refresh periodically and ensure data preservation. Hence, the IMC contains the logic to perform memory refresh as well.

The DRAM chip contains multiple banks. Each bank is a two-dimensional array organized in rows (often referred to as Pages) and columns. The memory controller manages the DRAM using the following essential signals:

- *Activate (ACT)*: ACT opens a row of a DRAM bank for reading or writing.
- *Row Access Strobe (RAS) and Column Access Strobe (CAS)*: The CPU uses RAS to specify the row where the data exists, followed by using CAS a short time later to specify the column.
- *Precharge (PRE)*: PRE deactivates open rows in one or all banks. A row remains open till it is deactivated using PRE.

A row is often referred to as a page. If data access occurs to an open row (*page-hit*), then the memory controller must only issue CAS to read/write the data. However, if all banks are idle, i.e., no rows are open (*page-empty*), then the memory controller issues ACT (and RAS) followed by CAS after a short delay. Finally, if the access occurs to a different row than the one currently open (*page-miss*), then the memory controller must issue all the commands sequentially with a slight delay between them: PRE, ACT (with RAS), and CAS.

In this dissertation, we refer to memory access time as the time between the start and finish of a memory request. Memory cycle time is the minimum delay between successive memory operations. Memory bandwidth indicates the number of bits or bytes transferred from the memory to the CPU cores per second. Memory latency refers to the amount of time (in clock cycles) taken to retrieve a byte (or a word) by a CPU core after initiating a memory request for the byte. Memory latency is lower when a page hits occurs, while it is higher when a page misses occurs.

II.3 System on a Chip (SoC)

System on a Chip (SoC) integrates almost all computer system components on a single integrated circuit. There is no specification of components that an SoC must include;

nonetheless, most SoCs include (multicore) CPU, memory, input/output ports alongside other components such as Field Programmable Gate Array (FPGA), Graphics Processing Unit (GPU), radio modems, or analog and digital processing units. SoCs provide higher average performance and lower power consumption than traditional processors as they tightly couple various components of a computer system. As a result, a new trend of using SoCs is emerging across various industries.

An SoC must have at least one processor core. When an SoC integrates multiple processors, we refer to it as a MPSoC. MPSoCs usually contain heterogeneous processing elements such as microcontrollers, microprocessors, or digital signal processors geared towards specific requirements of the targeted domain.

Similar to processors, SoCs have a memory hierarchy consisting of multiple caches and main memory. Some SoCs feature On-Chip Memories (OCMs) that can store instruction as well as data. SoCs also feature Shared on-chip interconnects to connect various components. SoC manufacturers often use ARM's AMBA standard for interconnect design. A new trend towards using NoCs for SoC interconnects is emerging as they allow integrating more components.

II.4 Examples of Multicore processors and SoCs

In this dissertation, we consider three different hardware platforms containing either a multicore processor or a SoC. This section gives a high-level description of these three hardware platforms.

II.4.1 NXP QorIQ T4240 with e6500 processor

Figure II.F4 shows an overview of NXP QorIQ T4240 [12]. We used this platform for implementing the avionics demonstrator of Chapter VI. Our platform has $3 \times 8GB = 24GB$ DRAM. The QorIQ T4240 has an e6500 processor with the following relavent features:

- 12 dual-threaded cores, arranged in three clusters of four cores each
- 64-bit architecture
- Maximum operating frequency of $1.8GHz$
- $32KB$ I-L1 cache and $32KB$ D-L1 cache per core, $2MB$ L2 cache per cluster (shared by cores of a cluster), and a 1.5 MB L3 cache (LLC) shared by all cores
- Shared on-chip interconnect in the form of CCF
- DDR3 memory controller
- Integrated $1Gbps$ and $10Gpbs$ Ethernet

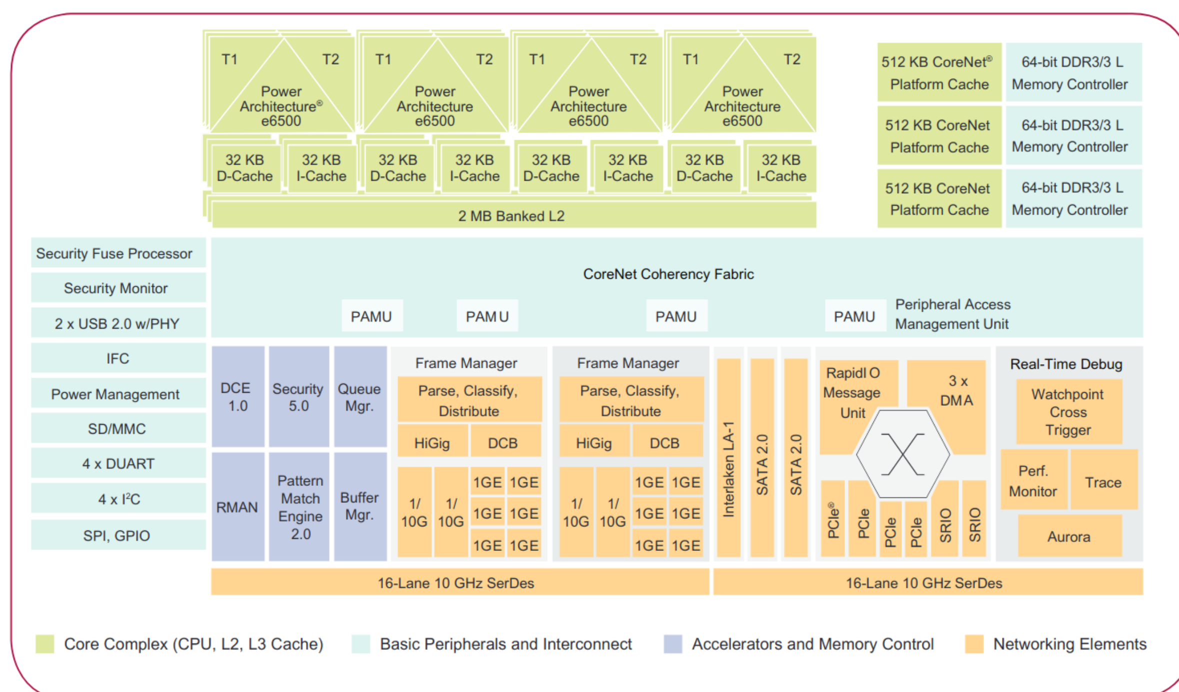


Figure II.F4: *NXP QorIQ T4240 [12]*

II.4.2 Xilinx ZC706 with Zynq-7000 SoC

The Xilinx ZC706 platform contains the Zynq-7000 SoC [13]. The platform has $1GB$ component memory on the Processing System (PS) and $1GB$ Small Outline Dual In-line Memory (SODIM) on the Programmable Logic (PL). We used this platform as the DREAMS Harmonized platform (DHP) for developing the avionics demonstrator of Chapter VI.

Figure II.F5 gives an overview of the Zynq-7000 SoC [13]. It has the following relavent features:

- Dual-core ARM Cortex A9 processor (ARMv7-A architecture) (a.k.a. PS)
- 32-bit architecture for the ARM processor
- Maximum $866MHz$ operating frequency for the ARM cores
- $32KB$ I-L1 cache and $32KB$ D-L1 cache per ARM core, $512KB$ L2 cache (LLC) shared by all ARM cores
- $256KB$ OCM (RAM)
- Xilinx Artix-7 FPGA (a.k.a. PL)
- ARM AMBA AXI [26] based shared on-chip interconnect
- DDR3 memory controller

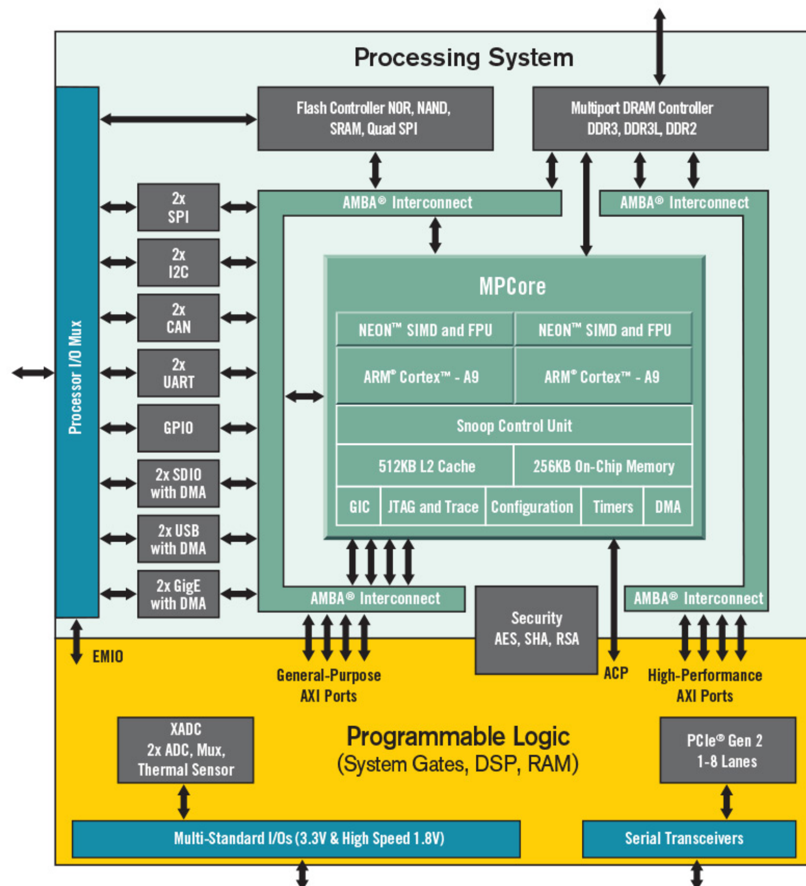


Figure II.F5: *Xilinx Zynq-7000 [13]*

- $10/100/1000 Gpbs$ tri-speed Ethernet

II.4.3 Dell R640 Server with Intel Xeon Gold 5218 Processor

Dell R640 is a dual-socket platform for use in data centers. Our Dell R640 platform contains a Intel Xeon Gold 5218 Processor [30, 31] and has $6 \times 16GB = 96GB$ DRAM. We used this platform in building a Real-Time Cloud (RT-Cloud) for the railway demonstrator from Chapter VII.

Figure II.F6 shows a simplified overview of the Xeon Gold 5218 processor. It has the following features:

- 16 dual-threaded processor cores (Intel Cascade Lake architecture)
- 64-bit architecture
- Base operating frequency of $2.3GHz$ (3.9 Ghz maximum Turbo frequency)

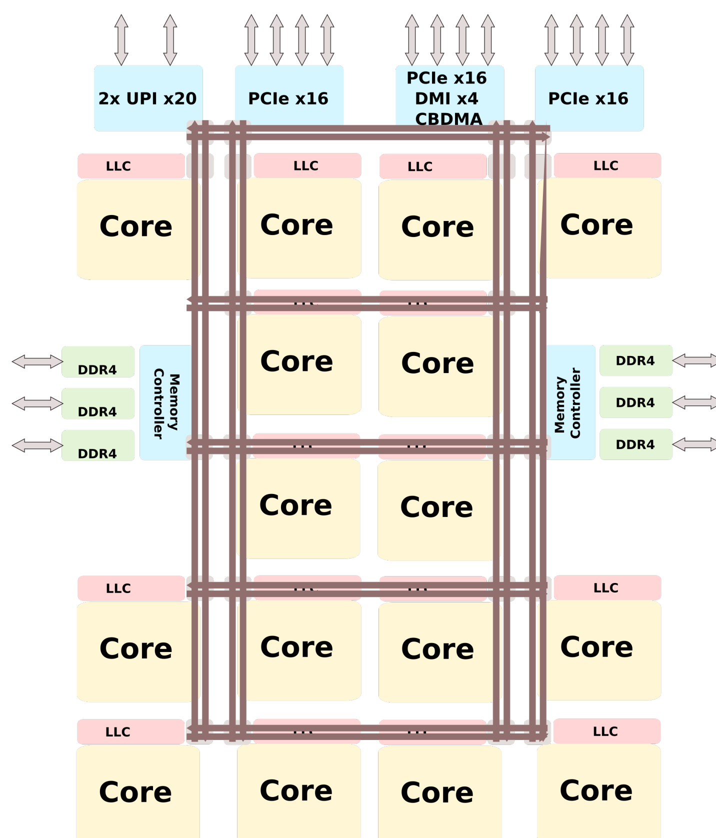


Figure II.F6: *Intel Xeon Gold 5218 Processor*

- $32KB$ I-L1 cache and $32KB$ D-L1 cache per core, $1MB$ L2 cache per core, and $22MB$ L3 Cache LLC shared by all cores
- Intel QPI [27] interconnect
- DDR4 memory controller

II.5 Real-Time Systems (RTS)

Real-Time Systems (RTS) are systems that must deliver the expected logical results within stringent timing constraints. We refer to RTS where failure to meet real-time constraints can lead to death or severe injury, loss or damage to property/equipment, or cause environmental harm as hard or *Safety-Critical real-time Systems (SCS)*. On the contrary, we refer to RTS where failure to comply with real-time constraints causes loss of functionality or performance without catastrophic consequences as *Non-Critical Real-Time Systems (NCRTS)*.

II.6 Task

A task, τ, is an elementary term used in real-time systems to describe a piece of software containing a sequence of instructions executed on a processor till completion. The collection of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ that execute on a computer system is referred to as a taskset, Γ. A task $\tau_i \in \Gamma$ can be activated multiple times with possible different input data. Each instant of a task activation with a specific input data is called a Job, J. In this dissertation, we consider an application, A, to be a group of tasks (possibly) communicating with each other and working together to achieve a common objective.

II.6.1 Task parameters

Each real-time task consists of specific properties dependent on the task implementation and the hardware architecture of the targeted processor. Moreover, real-time tasks consist of specific requirements put forth by the system designer. Each task has some essential task parameters that express these properties and requirements. The task parameters relevant to this dissertation are as follows:

- *Worst-case Execution Time (WCET)*, C_i, is the maximum length of time needed by a specific hardware platform to entirely execute a task, τ_i, without interruption over all possible input data. Thus, the WCET of a task is also the upper bound to the execution time of a task's jobs.

- *Release or activation time*, r_i is the time point when a job of a task, τ_i, becomes ready for execution.

- *Absolute deadline*, D_i, is a specific time point when a job of a task, τ_i, must finish executing. The *Relative deadline*, d_i, is a time point when a job of τ_i must finish executing relative to its own release time.

- *Priority*, $PRIO_i$, is a value that represents the relative importance of a task, τ_i, in a system with multiple tasks. In the context of this dissertation, we assign a task with more importance a higher priority value.

- *Preemption* refers to the possibility of interrupting a task in favor of another task. The intention is to resume the former task at a later point in time. Depending on the time point when preemption is possible, real-time tasks can be divide into three categories:
  1. A *fully preemptive* task is interruptible at any point in time.
  2. A *non-preemptive* task cannot be interrupted during its execution.
  3. A *partially preemptive* task is only interruptible at specific time points, or it cannot be interrupted during the execution of specific code sections.

- Real-time literature classifies tasks into hard, soft, and firm depending on the consequences of a deadline miss. However, the definitions of these classifications

vary. The IEEE Technical Committee on IEEE Technical Committee on RTS (TCRTS) defines them as:

1. A task is *hard* if a deadline miss by a job of the task jeopardizes the correct behavior of the system.
2. A task is *firm* if the result produced by a job of a task is entirely useless upon deadline miss by the job, but it does not jeopardize the system's correct behavior.
3. A task is *soft* if the utility of the result produced by a job of a task reduces upon deadline miss by the job, and it does not jeopardize the system's correct behavior.

- It is important to note that SCS tasks are not always hard, while NCRTS tasks are not always firm or soft. For example, deadline misses in a digital media player, a NCRTS, jeopardize the correct system behavior without causing catastrophic outcomes. In practical engineering contexts, for some SCS, the occasional loss of few deadlines can be tolerated due to the robustness of control algorithms and the resultant ability to react appropriately at the next invocation step without severe consequences.
- Criticality, l_i, is a designation for the level of assurance against failure needed in a task, τ_i. Safety standards such as the automotive domain's ISO26262 [32] and avionics domain's DO-178B [33] each define up to five criticality-levels called Automotive Safety and Integrity Levels (ASILs) (ASILs) and Design/Development Assurance Levels (DALs), respectively. As noted by [34] and others, different Mixed-Critical System (MCS) papers and standards assign criticality varying definitions.

II.7 Task models

In real-time systems, task models impose assumptions and focus on the essential task properties while abstracting unnecessary details. Liu and Layland [35] introduced the most predominantly used *periodic model*. In this task model, a task consists of an infinite sequence of jobs released periodically with a fixed interval between the release times of any two consecutive jobs. As a result, tasks in the periodic model have an additional parameter:

- *Period*, T_i, of a task, τ_i, is the fixed interval between the release times of two consecutive jobs of that task.

The periodic model assumes that the deadline of a job is implicitly equal to the task period ($d_i = T_i$), and the release time of a job is at the period start. Besides, this model considers that the tasks are independent of each other, have bounded execution time, and the jobs of a task cannot suspend themselves. A final assumption is that the scheduling overhead should be negligible. There exist various variations of this task model based on the relationship between the task parameters, such as $d_i < T_i$ or $d_i > T_i$.

In real-time scheduling theory, the term *hyperperiod* represents the duration after which the pattern of job release times starts to repeat for the whole taskset. The hyperperiod of a taskset, $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, is equal to $LCM(\tau_1, \tau_2, \ldots, \tau_n)$. The *utilization*, U_i, of a task expresses the periodic load generated by the task. For a task, $\tau_i \in \Gamma$, $U_i = C_i/T_i$. The *utilization*, U, of the taskset, Γ, is equal to $\sum_{i=1}^{n} U_i$.

Another commonly used task model in real-time systems is the sporadic model[36], which extends the periodic model by introducing a minimum time interval, called the inter-arrival time, T_{Si}, between two consecutive jobs of a task instead of a fixed time interval (Period). Apart from periodic and sporadic tasks, a system can also have aperiodic tasks [37]. Aperiodic tasks have a possibly infinite sequence of jobs without any restriction on the time interval between two consecutive job releases. Aperiodic tasks model tasks with infrequent jobs or jobs with irregular arrival patterns. There exist other task models, but many of them are application-specific and not directly relevant to this dissertation.

II.8 Mixed-criticality Systems (MCS)

Vestal [38] observed that it is not always possible to determine the WCET value of a task with complete certainty. The WCET, C_i, of a task, τ_i, is dependent on the criticality level, l_i, of τ_i. Since the verification process becomes more conservative with the increase in the criticality levels, the process considers more pessimistic WCET estimations to increase confidence and reduce uncertainties. Based on this observation, Vestal proposed a mixed-criticality task model extending the periodic model by introducing the following:

1. An order set of Criticality levels, $L = \{A, B, C, D\}$ with A being the highest criticality level in the system.
2. A WCET, C_{il}, for each task, τ_i, at every criticality level $l_i \in L$. Since the WCET increases with criticality of the task: $C_{iA} > C_{iB} > C_{iC} > C_{iD}$.

The goal of this model is to assure to level l_i that for all τ_i deadline is never missed. [39, 40] extend the vestal model by allowing a task to have multiple deadlines based on the criticality levels such that $L_i^1 > L_i^2$ than $D_i^1 < D_i^2$. Similarly, they also allows multiple periods based on the criticality levels. Some papers such as [41, 42] reduce the complexity of the MCS task model by considering just two criticality levels: Low (LO) and High (HI). In these dual-criticality models, a task, τ_i, has just two WCET values: $C_i(LO)$ and $C_i(HI)$.

Many research papers in MCS consider that system can execute in different *criticality modes*. The MCS starts executing in the lowest criticality mode, $\kappa = 1$, with criticality $L = 1$. The system stays in this mode as long as all tasks comply with this mode. If a job of a task executes longer ($C_i^{actual} > C_i^L$) or a task executes more often than permitted during this mode ($T_i^{actual} > T_i^L$), then a mode change occurs, and the MCS moves to a higher criticality mode, $\kappa + 1$ with criticality level $L + 1$. The mode change results in suspension of all tasks executing with criticality l_i less than equal to $\kappa + 1$. Some papers allow the criticality mode to only increase upwards. In contrast, other works allow the

criticality mode to revert to lower ones upon reaching a specific online state or if the system has slack.

However, Industrial safety standards such as ISO26262 [32] or DO-178C [43], do not recommend different WCET estimation techniques for tasks with different criticality levels. Mixed-criticality industries consider a task to have one single value on a specific hardware platform based on the assurance required by the task's actual criticality level. Higher the required assurance, more rigorous are the techniques used to demonstrate that the task meets the timing and the safety requirements. Industries consider a task to have one single WCET value on a specific hardware platform based on the assurance required by the task's criticality level. Industries assign criticality levels to a task based on the consequences of failure, the likelihood of failure occurrence, and the ability of the system to deal with faults in the task. Tasks deemed as non-critical have minor or non-catastrophic consequences of failure or negligible probability of occurrence. As explained in [44], when two tasks, τ_1 and τ_2, have the same severity class for failures, but if τ_1 has a higher probability of occurrence for failure than τ_2, then τ_1 may be assigned a higher criticality level than τ_2. Thus, the criticality levels assigned to tasks do not indicate the importance of the tasks. Consequently, favoring tasks with higher criticality levels over lower criticality tasks is unwarranted.

Industrial standards recommend isolation between tasks of different criticality levels even when they share the underlying system resources. Certification Authorities (CAs) require evidence, especially from non-critical to critical applications, to prove (1) isolation between tasks of different applications and (2) faults do not propagate between applications. If applications are not isolated, all non-isolated applications must have their tasks certified to the highest criticality non-isolated application to ensure meeting all safety requirements and prevent vulnerabilities from security attacks. However, certifying all tasks to the highest criticality level is extremely expensive and practically not viable. In this dissertation, we consider the industrial notion of mixed-criticality tasks.

ARINC 653 [45] and AUTOSAR [46] are two examples of industry software specifications developed to meet these isolation and independence requirements. In this dissertation, we consider ARINC 653 in the avionics use case (Chapter VI). Section II.13.1 gives a basic idea of this software specification.

II.9 Scheduling

Scheduling is a decision-making approach for allocating resources, such as cores, memory, and network bandwidth, to tasks of a taskset. A *scheduling algorithm* is a set of rules to perform scheduling. A *scheduler* applies a scheduling algorithm and determines the allocation of resources to jobs of tasks at the exact moments in time to produce a *schedule*. A *fully preemptible scheduler* can preempt the currently executing job of a task and apply a new scheduling decision. On the contrary, a *non-preemptible scheduler* does not interfere with a currently running job and only applies new scheduling decisions after the current job completes. A *dispatcher* is part of the scheduler that comes into play after the scheduler makes the resource allocation decision. The dispatcher gives the job(s) selected by the scheduler control over the desired resource(s).

The schedule is referred to as *valid* if it ensures that all jobs meet their timing constraints. Suppose there exists a valid schedule for a taskset in a system using a particular scheduling algorithm. In that case, the taskset is referred to as *schedulable* on this system using the specific scheduling algorithm.

Priority assignment to tasks forms an essential basis for the classification of real-time system scheduling. In *fixed-priority scheduling*, tasks have a fixed priority throughout the system's runtime, i.e., all jobs of the same task execute at the same priority. The Rate Monotonic Scheduling (RMS) algorithm is an example of fixed-priority scheduling. In RMS, a task with a smaller period gets a higher priority than a task with a longer period. [47] extended RMS to multiprocessor systems. The paper presented two different approaches: *partitioned* and *global*. In the partitioned approach, jobs of a task are bound to a processor and cannot migrate to another processor. Contrarily, in the global approach, the jobs of a task can execute on different processors, i.e., task migration is permitted.

In *dynamic-priority scheduling*, the priorities of a task can change at runtime, i.e., jobs of the same tasks may have different priorities over time. The Earliest Deadline First (EDF) algorithm is an example of dynamic-priority scheduling. In EDF, a task with the next upcoming deadline in the system has the highest priority. Liu and Layland [35] provided some initial analysis of fixed-priority and dynamic-priority schedulers on single-core processors.

[48] presents a survey covering many hard real-time multiprocessor scheduling algorithms. [49] gives a comprehensive overview of multicore scheduling algorithms.

II.10 Timebase

A safety-critical distributed system has multiple applications concurrently executing on different nodes. Moreover, replicated nodes may be present for fault tolerance. All nodes of the system have their own clocks. For Time-Triggered (TT) scheduling, the system must have synchronized clocks forming a *global timebase* for use by all applications on different nodes [50].

Depending on when the events from a *set of significant events* are allowed to occur on a directed *timeline* consisting of an infinite set of instances (ordered and dense set), there exist two kinds of timebase [50]:

1. In a system with a *dense timebase*, an event can occur at any instant of the timeline.

2. In a system with *sparse timebase*, an event can occur only at some fixed instances in the timeline. Thus, there are time intervals between the fixed instances when no events can occur. Fohler [51] referred to these minimum scheduling quantums as slots, while Steiner [52] referred to them as macroticks. The Time Triggered Architecture (TTA) uses a sparse-time base to ensure consistent ordering of events.

II.11 Time-Triggered (TT) and Event-Triggered (ET) Task Schedulers

There are two main scheduling approaches in real-time systems depending on the scheduler's activation: *TT* and *Event-Triggered (ET)*. In TT scheduling, the scheduler activation takes by the progression of global time, i.e., when the global time reaches predefined points on the global timebase, the scheduler is activated to schedule a task. On the contrary, scheduler activation in ET scheduling occurs due to significant events from the environment or by the progression of the previous task. Examples of such events are job releases, task job completion, and user actions.

The TT scheduler mainly consists of a dispatcher that assigns resource(s) to task(s) based on an offline (during design phase) computed schedule. The scheduler receives this schedule as a *scheduling table* consisting of all the necessary scheduling decisions and the point in time the dispatcher should implement those decisions. Since TT scheduling uses offline tables, it is referred to as offline scheduling as well.

To create the scheduling table for TT schedulers, the system designer must know all tasks in the system before runtime. The scheduler cannot handle a task that is specified offline. Since the exact activation times of sporadic or aperiodic tasks are unknown before runtime, they must be considered as periodic tasks when creating the scheduling table. Moreover, if the system designer needs to add a new task to an existing scheduling table, he/she must recompute the entire scheduling table. While creating the scheduling table, assignment of a resource to a task takes place based on the worst-case resource demand. Nevertheless, most tasks utilize only a fraction of the allocated resources at runtime in the average case.

In TT scheduling, since the creation and validation of the scheduling table takes place offline, a system designer can factor in complex constraints such as latency and precedence constraints, which would otherwise incur large overheads to handle directly at runtime. A TT scheduler also enforces strong temporal isolation between tasks. Since the scheduler is activated periodically, it can easily ensure that job of a task overrunning its WCET does not hamper other tasks' schedulability. Finally, a system using TT scheduling is highly predictable as events occur pre-planned at fixed points in time. Thus, testing and certifying the system is easier as there are only a few predictable scenarios to consider.

A scheduling table that contains all possible decisions for the complete lifetime of the system will be enormous and not easy to store on a node. Thus, there is a need to create a comparatively much smaller scheduling table, for example, for a hyperperiod of the taskset. The scheduler executes this smaller table on a cyclic basis for the system's entire lifetime. The smaller schedule stored in the scheduling table is referred to as a *cyclic schedule*.

ET schedulers, such as EDF, offer much more flexibility than TT schedulers, as they can handle events without pre-planning (before runtime). It is unnecessary to create an offline scheduling table as the scheduler allocates resources to tasks during runtime based on a scheduling algorithm and the actual demand. Once the tasks release the resources (often before the worst-case resource demand), the resources become available

for allocation to other tasks. The ET schedulers must handle all events quickly for them to ensure timely reaction to future events. As a result, the ET schedulers require simple scheduling algorithms to keep runtime overheads low (comparable to TT schedulers) and do not usually consider complex constraints.

ET schedulers can have much higher overheads despite a simple scheduling algorithm than TT schedulers, especially during peak load conditions [53]. Some scheduling algorithms used in ET schedulers do not ensure system schedulability when the job of a task overruns. Moreover, ET schedulers can produce widely different schedules for the same system when the sequence or timing of events changes leading to lower predictability. A system with an ET scheduler requires exhaustive testing using simulated loads considering even the rarest events. However, it is not straightforward to prove that the tests covered all possible scenarios that may occur at runtime. Kopetz [53] provides a detailed comparision between TT and ET scheduling.

An intriguing research aspect is the development of an architecture that combines the benefits of both the contrary schedulers. Fohler [54] presented the Slot-shifting algorithm of TT and ET tasks. Schorr [55] extended the Slot-shifting algorithm to multiprocessor systems. Syed [56] and Real et al. [57] presented a scheduler for admitting ET tasks in a hierarchical TT system. Other works such as [58, 59, 60, 61] also provide scheduling techniques that try to take advantage of both types of scheduling methods.

II.12 Time Triggered Architecture (TTA)

TTA provides a framework for the design and implementation of distributed SCS. In TTA, the progression of time initiates all activities of the computer system [62]. In most TTA implementations, interrupts caused by periodic overflows of a timer run the TT scheduler that schedules significant activities according to the offline schedule at pre-determined points in time. For consistent behavior in a distributed system using TTA, the system must have synchronized clocks forming a *global timebase* for use by all applications on different nodes. However, the finite precision of the global timebase makes it hard to consistently order events based on global timestamps in a distributed system. TTA ensures consistent ordering of events by using a sparse time base.

The TTA uses a two-phase design approach to support composability of applications and reuse of prevalidated components:

1. In the architecture design phase, the interfaces and the interactions among the distributed components in value and time domain are specified.

2. In the component implementation phase, the components are built, taking the interface specifications from the architecture design phase as constraints.

The DREAMS cross-domain architecture [2] used in the avionics demonstrator of Chapter VI is inspired by TTA and uses TT-Ethernet (TTE) for communication between the distributed system nodes. The next section explains TTE.

II.13 TT Ethernet (TTE)

Kopetz et al. [63] extended the standard ethernet (IEEE 802.3) with ideas from Time-Triggered Protocol (TTP)/C [64] to form TT-Ethernet (TTE) for use in safety-critical real-time and mixed-criticality systems. The main aim was providing services such as predictable message transmission, clock synchronization, membership, and redundancy management in distributed real-time systems. We use TTE for the mixed-criticality avionics demonstrator in Chapter VI.

TTE maintains compatibility with standard ethernet. It uses Ethernet frames to transfer data between nodes (referred to as *End System (ES)*) via special TTE switches. It meets the requirements of real-time and non-real-time (including multimedia) applications by providing several traffic classes in parallel on the same ethernet network:

1. *Time-Triggered (TT) class*: messages of the TT class get dispatched as per an offline-defined communication schedule. *Virtual Links (VLs)* provide temporal reservations for TT class messages on the physical links. They are defined by period, frame size, source ES, destination ESs, and a route. An offline-defined table stores time intervals (called *windows*) during which each VL gets access to the physical link. An offline planning tool ensures that window assignment to TT VLs is conflict-free, i.e., no two TT VLs will compete for the physical link. A dispatcher sends a VL's outgoing messages and receives its incoming messages during the respective time window. The dispatcher discards any frame received outside the time window. TTE provides a fault-tolerant clock synchronization protocol to ensure synchronization of all nodes for proper functioning TT class. The avionics demonstrator (Chapter VI) uses TT class messages for communication between safety-critical applications. Moreover, we use TT class messages for the global resource management communication in the avionics demonstrator.

2. *Rate-Constrained (RC) Class*: RC class messages on the same VL get dispatched with a minimum inter-frame duration between them called *Bandwidth Allocation Gap (BAG)*. Thus, the RC VLs have BAG instead of period in their definition. RC class does not require any form of synchronization. However, a system designer must use static analysis to check the latency and jitter of a message and ensure that they are within an acceptable range. The avionics demonstrator (Chapter VI) uses TT class messages for safety-critical applications. We use RC class messages for the non-critical frame-skipper application in the avionics demonstrator.

3. *Best-Effort (BE) class*: frames of the BE class get dispatched as per the best-effort paradigm, i.e., the dispatcher sends BE frames when it does not send TT or RC class frames.

Let us consider an example with three nodes: $ES1$, $ES2$, and $ES3$ communicating via a TTE switch. For simplicity, assume that all links are unidirection as shown in Figure II.F7. $ES1$ and $ES2$ must send safety-critical TT class messages every $3ms$ and $5ms$ respectively. Hence, a system designer must create a TT VL, A, with a $3ms$ period having $ES1$ as the source and $ES3$ as the destination as shown in Table II.T1 (the table

Virtual Link (VL)	Type	Source	Destination	Route
A	TT	ES 1	ES 3	ES 1->TTE SW->ES 3
B	RC	ES 1	ES 3	ES 1->TTE SW->ES 3
C	TT	ES 2	ES 3	ES 2->TTE SW->ES 3
D	RC	ES 2	ES 3	ES 2->TTE SW->ES 3

Table II.T1: *Example of TTE VLs*

doesn't show the periods and BAG for simplicity).. Similarly, the system designer must also create a TT VL, C, between $ES2$ and $ES3$ with a $5ms$ period. The system designer can schedule the TT VLs using an offline planning tool. Apart from TT VLs, the nodes $ES1$ and $ES2$ must send RC class messages to $ES3$. Thus, the system designer must also create RC VLs, B and D, with an appropriate inter-frame gap. Moreover, $ES1$ and $ES2$ send multiple BE class messages to $ES3$. Figure II.F7 shows a possible schedule for the three nodes. The TTE dispatcher repeats the pattern of TT class messages shown in the figure every $3ms \times 5ms = 15ms$ cycle.

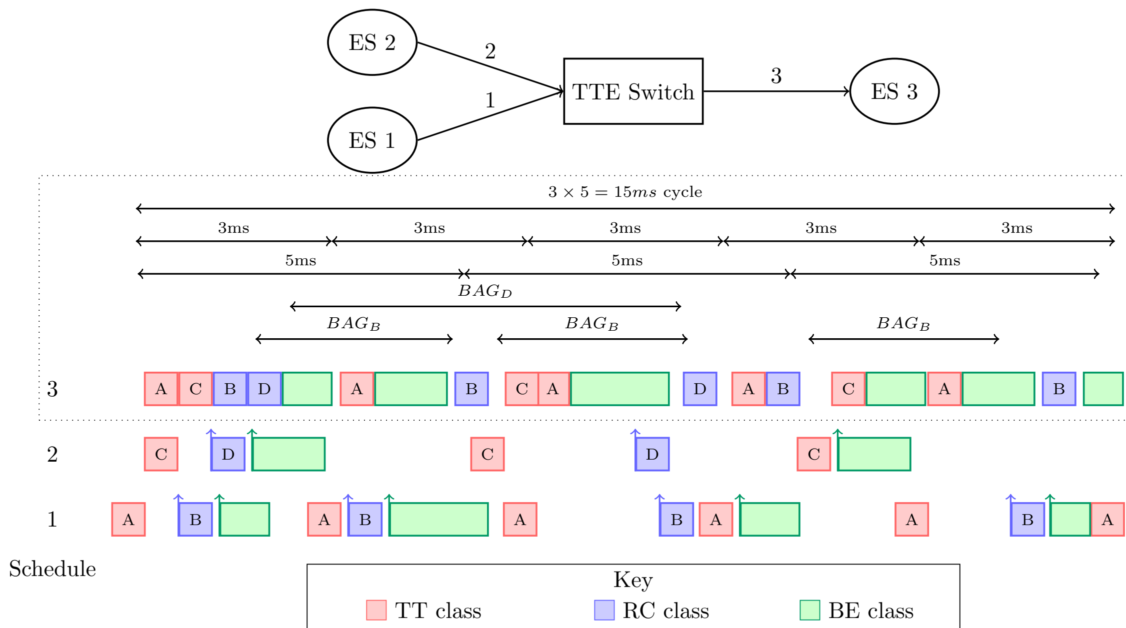


Figure II.F7: *Example of TTE Schedules*

II.13.1 ARINC 653

ARINC 653 [45] is a software specification to support the isolation of multiple safety-critical avionics applications, possibly with different criticality levels, hosted on a common hardware platform in the context of IMA. ARINC 653 defines the *APplication EXecutive interface (APEX API) (APEX API)* to decouple every application from the underlying

Operating System (OS) or hypervisor. The APEX API components belong to the following categories:

- **Partition management:** In ARINC 653 , the isolation is achieved through partitioning techniques to allow separation among applications and support validation and verification. Each application runs in an environment, called an *application partition*, containing all the necessary data, context, and configurations needed to execute the contained application. Henceforth, We shall refer to an application partition simply as a *partition*. Each partition provides spatial and temporal partitioning to the application running in it:

  1. *Temporal partitioning:* The schedule of each partition should be deterministic, with each partition executing for a fixed amount of CPU time. To acheive this, ARINC 653 extends the concept of TT scheduling to provide temporal partitioning. Each partition executes in a fixed time window called a *Minor Frame (MiF)*. A scheduling table-driven scheduler activates the MiFs on a fixed cyclic basis while ensuring that the partitions executing in the MiFs have uninterrupted access to the shared resources. Such a scheduler is called a *cyclic scheduler*. Each partition executes at least once in a time duration called the *Major Frame (MaF)*, as shown in Figure II.F8. MaF is often referred to as Major Cycle (MaC). The total time duration of a MaF equals the sum of all MiFs' duration belonging to the MaF.

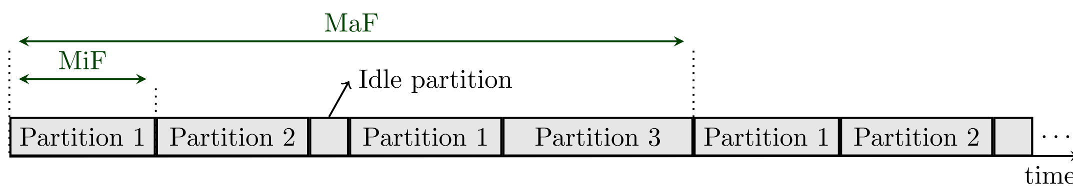


Figure II.F8: *ARINC 653 Minor Frame (MiF) and Major Frame (MaF)*

  When a MiF ends, the cyclic scheduler must preempt the currently executing partition and execute the subsequent partition of the MaF. All preempted partitions continue execution in the next MiF belonging to them.

  System partitions are an additional type of partition that can optionally exist in an ARINC 653 platform. These partitions can provide services not supported by APEX API, such as fault-tolerance or device drivers. System partitions can bypass the APEX API and communicate directly with the underlying OS (or hypervisor).

  2. *Spatial partitioning:* Each partition has a fixed assignment of memory allocation to it, and no partition can access memory regions outside those assigned to it.

- **Process management:** Process management refers to the management of tasks running in the partitions. There can be one or more periodic and sporadic tasks

belonging to an application running in a partition. Periodic tasks execute during each MiF of the partition, and they should be shorter than the duration of the MiF. Each task has a fixed priority assignment. Only tasks with higher priority can preempt tasks of lower priority. Tasks can only communicate with the underlying OS or hypervisor via the APEX API. All code in a partition, including tasks, execute in user mode only, i.e., they cannot perform privileged operations. Since all tasks of a partition belong to the same application, blocking or terminating a task in case of faults or overload situation is linked with the task's importance for the application rather than the task's criticality.

- **Intrapartition and interpartition communication:** Tasks in a partition can use buffers, blackboards, semaphores, and events for intrapartition communication and synchronization. For interpartition communication, tasks in a partition can send messages of finite length via offline-defined logical links called *channels*. Channels can exist between a source partition and one or more destination partitions. Partitions can access these channels via offline-defined access points called *ports*. We will refer to a port in a partition sending a message as a source port and the port in a partition receiving the message as a destination port. Two different modes exist to configure source and destination ports:

  1. *Sampling mode:* A port setup in sampling mode is suitable to transfer messages of the same size and structure but containing variable data. A message remains in the port until it is transmitted or overwritten by a new message. A message remains in a destination port until overwritten by a new message (non-consuming read). Thus, the destination partition always has access to the last received message.

     The system designer can define a time called the refresh period for which the message remains valid. The destination partition can read messages from the destination port after the refresh period, but it receives a flag indicating that the refresh period has expired.

  2. *Queuing mode:* A port setup in queuing mode is suitable to transfer messages with varying unique data. Source and destination queuing ports do not overwrite existing messages. A source port stores all messages written to it in a queue before transmitting them. Once a message is transmitted, the source port deletes the message from its queue. When a message is received, the destination port stores the message in a queue as well. The destination port deletes all messages read by the destination partition from its queue (consuming reads).

     The source and destination ports have buffers of predefined sizes. A task in a source partition can get blocked when it causes a buffer overflow in the source port. On the contrary, a task in a destination partition can get blocked when it reads from a destination port with an empty buffer.

- **Time management:** ARINC 653 provides time slices for scheduling, deadlines, periods, and timeouts of tasks in a partition. Process management and time

management together ensure the timely execution of tasks in a partition.

Avionics domain often uses specialized virtualization techniques (hypervisors) that implement the ARINC 653 APEX API. Section II.17 provides further information on virtualization techniques.

Avionics and other safety-critical domains are shifting to multicores and MPSoCs not only for their advantages but also because they expect mass-market obsolescence for single-core processors soon[14]. As explained in Chapter I, multicore platforms and MPSoCs suffer from contention in the shared resources, which can cause unpredictable delays leading to deadline misses in safety-critical applications. The following section explains the issues about contention in shared resources. The section further presents some related work on methods to analyze the impact of shared resources on the execution of tasks. Finally, it provides an overview of some existing solutions that deal with shared resource contention.

II.14 Shared Resource Contention

In the past, RTS and MCS used single-core processors and federated architectures that ensured deterministic response time and adequate resource isolation. Single-core processors execute tasks sequentially, i.e., only one task runs on the CPU at a time and accesses the node's resources exclusively. To ensure that all the tasks meet their resource demands, a resource allocation approach determines when a context switch must occur to replace the current task with another one. These approaches, for example EDF, primarily focus only on the CPU.

However, COTS multicore processors and MPSoCs are rapidly replacing single-core processors. Unlike single-core processors, multicore processors (or MPSoCs) execute multiple tasks concurrently on different processor cores. When these concurrently executing tasks compete for access to a shared resource with limited bandwidth, it results in unpredictable resource access delays. The main reason for such contention in shared resources is the arbitration delay. For example, assume a shared resource that can handle one resource access request at a time, but it receives two resource access requests simultaneously from two cores. The resource selects one of the two requests based on an arbitration algorithm and serves it. However, it can serve the remaining request only after serving the first selected one. Some examples of shared resources in multicore processors and MPSoCs leading to execution time variations for tasks are as follows:

- Parts of the memory hierarchy (shared caches and main memories, including memory controllers) are crucial points of contention since all tasks need to access the memory regularly for fetching instructions and reading or writing data. Despite multiple tasks accessing the memory, an IMC can only serve one memory request at a time. Thus, tasks have a significant impact on memory access latency as a result of memory contention. For example, Nowotsch et al. [65] demonstrated that the latency for a single memory store operation on a P4080 multicore platform

could increase by a factor of 25.82 when the number of active cores increases from 1 to 8. Radojković et al. [66] reported a maximum slowdown of 10% due to memory bandwidth contention and a 14.4× slowdown in different benchmarks due to contention in the L2 (shared) cache when executing benchmarks on an Intel Core2Quad processor

In most multicore and MPSoC architectures, it is tough to determine the actual memory latencies of tasks. Some factors that contribute towards increasing the complexity in determining the memory latencies are as follows:

- Concurrently executing tasks on different cores can access the main memory simultaneously and thus compete for memory bandwidth. The memory controller arbitrates memory access based on a hardware-implemented arbitration algorithm. The hardware designers optimize this algorithm to improve the system's average-case performance without regard for providing real-time guarantees. Often, the actual arbitration algorithm or some parameters thereof are unknown to the software designers. Similar concerns apply when concurrently executing tasks on different cores access a shared cache.
- Concurrently executing tasks τ_i and τ_j operating on different memory pages can cause a significant number of unexpected page-misses in the IMC. It is very tough to determine which exact memory access of τ_i will have a page-miss due to memory accesses of τ_j. As explained in Section II.2.3, a page-miss has the highest memory access overhead.
- A task τ_i can cause a shared cache to evict cache lines required by a concurrently executing task τ_j. As a result, the following access from τ_j to the data stored in the evicted cache line will take much longer due to a cache miss.
- Private caches of multiple cores can store a copy of the data from the same memory address. When a core writes to the address stored locally in its private cache, the private caches of other cores with data pointing to the same address must reflect the change. We refer to this problem of keeping data consistent between caches as the *cache coherency problem.* Cache coherency protocols solve this problem. However, the overheads involved in the protocols make it harder to analyze the worst-case behavior of tasks in multicore systems.

• The on-chip interconnect is another crucial juncture of contention since all cores access shared resources via the on-chip interconnect. Moreover, the DMA controller, IO/ devices, and cache coherence mechanism traffics are routed through the on-chip interconnect. For example, authors in [67] show that tasks can have a 46% variance in execution time due to interference from cache activities and I/O peripherals in the on-chip interconnect on an Intel Core2 CPU.

Bus-based architectures serialize access to the on-chip interconnect, while cross-bar architectures limit the total number of simultaneous accesses to the interconnect. In both architectures, a hardware-implemented arbitration algorithm controls access to the interconnect. NoC-based architectures also require some form of arbitration in the NoC routers. Similar to the arbitration algorithms in memory controllers,

interconnect arbitration algorithms aim to improve the system's average-case performance without regard for providing real-time guarantees. Often, the actual arbitration algorithm or some parameters thereof are unknown to the software designers.

- Contention can occur in other shared resources such as Direct Memory Access (DMA) controllers, interrupt controllers, and I/O devices in multicore processors and MPSoCs when tasks access them simultaneously. Moreover, MPSoCs have additional shared resources such as GPU where contention may occur.

II.15 Existing Approaches for Addressing Shared Resource Contention

There is a considerable amount of research on methodology to analyze the impact of shared resources on the WCET of tasks in COTS multicore processors. For example, Pellizzoni et al. [68] present an approach to calculate the upper bound for tasks delay due to memory contention. Yan and Zhang [69] present an approach to bound the WCET for threads on a multicore processor with shared instruction caches. The authors propose to statically analyze the worst-case cache interferences between different threads based on each thread's program control flow information. Chattopadhyay et al. [70] propose a unified WCET that considers shared caches, on-chip interconnect, and other micro-architectural components such as instruction pipeline and branch predictor. Sensfelder et al. [71] propose an approach based on timed automata to model and analyze the interference caused by cache coherence. In general, these methods suffer from enormous computational complexity as they need to search vast state spaces to consider all possible task interactions. Mancuso et al. [72] propose a Single Core Equivalence (SCE) framework by statically allocating $1/m$ of the shared resources such as DRAM banks, memory bandwidth, and shared caches to each of the m cores of the COTS platform. An essential aim of this framework was to provide WCET(m), a parametric WCET estimation for a task running on top of such a statically partitioned platform. Contrary to the static memory bandwidth allocation in SCE, Agrawal et al. [73] considered dynamic memory bandwidth allocation. They presented a method to analyze the worst-case response time of a task executing in a sequence of intervals, and each interval could have a different memory bandwidth allocation.

There have been many different methods proposed for resource allocation on COTS multicore processors. Some approaches propose developing completely new specialized real-time processor hardware that achieves a balance between performance and predictability. For example, Edwards and Lee [74] suggested specialized processor hardware called the Precision Timed (PRET) cores. Hardy et al. [75] proposed to reduce the worst-case interference in shared caches by performing static analysis to identify repeatedly used task blocks and only allow caching of these tasks blocks while forcing single-use task blocks to bypass the caches. The implementation of this approach requires adding an extra bit to the instructions set to control their cacheability. However, making changes to

existing hardware architectures or developing a new processor architecture is expensive and not applicable to COTS multicore processors and MPSoCs.

Some existing resource allocation techniques involve Time-Division Multiple Access (TDMA) arbitration mechanisms where an arbiter grants resource access to tasks in statically assigned time slices. For example, Rosen et al. [76] proposed storing a TDMA based bus schedule in the memory that the arbiter can access at runtime. In addition, Schranzhofer et al. [77] proposed a framework to analyze the worst-case response time of tasks considering TDMA arbitration for a shared resource. Nevertheless, as noted in [78], TDMA arbitration provides high predictability but yields poor resource utilization due to wastage of unused time slices.

Some existing approaches propose deterministic execution models to limit the contention in multicore processors. For example, in [79], authors proposed decomposing tasks into a fixed sequence of superblocks and dividing each superblock into three phases: acquisition, execution, and replication. This model allows tasks to only access shared resources in the acquisition and replication phases. In addition, the model prevents acquisition and replication phases of simultaneously executing tasks from overlapping to avoid resource contention. However, this model requires a tight static analysis approach to work effectively, and any minor design changes can significantly increase a task's worst-case response time. The PRedictable Execution Model (PREM) [80] was proposed to minimize contention from peripheral devices in COTS single-core processors. Houdek et al. [81] provided building blocks for implementing PREM on an ARM-based MPSoC. PREM splits task jobs into two non-preemptible intervals: a special predictable interval with no system calls and interrupts and a compatible interval with no special provisions. There are two phases in a predictable interval: a memory phase to prefetch all task data and instructions into private caches and an execution phase to perform the required task computation predictably without cache misses. The PREM model requires code instrumentation to mark task sections for execution in predictable intervals and a PREM-aware compiler for task compilation. Yao et al [82] introduced a memory-centric scheduler to improve performance in a TDMA based memory arbitration and considered a PREM model. The memory-centric scheduler increased the priority for all active jobs in the memory phase on a core over jobs in the execution phase on the same core when the TDMA scheduler granted a memory slot to that core. The sliced execution model [83] divides the CPU execution time into multiple slices of two types: execution and communication slices. In an execution slice, a core executes a task based on prefetched instruction and data. While in a communication slice, the core flushes all data from the previous execution slice to the DRAM and prefetches the instruction and data required for the next execution slice. The sliced execution model requires a particular toolset to generate the sliced architecture. All these models poorly utilize the resources [78]. Moreover, any requirement of code instrumentation makes it challenging to transition single-core legacy safety-critical applications to COTS multicore processors and increases the recertification costs. Biondi et al. [84] suggested taking advantage of a Logical Execution Time (LET) model [85] that decouples a task's CPU execution phase and the communication phase. They proposed a LET-based model that restricts the memory accesses of each task (prefetching and writing to memory) to precise time windows

(communication phase) located at the beginning of a task's period, thus avoiding memory contention by design. As noted by the authors, such a model can have priority inversion as the LET communication phase for a low-priority task can delay the execution of a high-priority task.

Some approaches take inspiration from aperiodic servers. For example, in [67], the authors proposed hardware servers that mask the unpredictable activity of I/O peripherals in COTS single-core processors, similar to how aperiodic servers mask the unpredictable aperiodic task arrival patterns. This idea can be extended to multicore processors as well to control contention from I/O peripherals. Bellosa [86] proposed using monitors built into the embedded hardware to count events occurring in the CPU, memory, and I/O subsystem. In this dissertation, we refer to these inbuilt event monitors as *Performance Monitor Counters (PMCs).* Bellosa also pointed out the possibility of using PMCs in reserving memory bandwidth for soft real-time tasks employing memory throttling mechanisms. The main idea of memory throttling is to limit the number of memory accesses, thus regulating the memory bandwidth (similar to how aperiodic servers work). Yun et al. [87] proposed an analytical method to calculate memory throttling parameters that can guarantee memory bandwidth to a safety-critical task executing on one core while limiting the impact on non-critical tasks executing on all the other cores. To determine the number of memory accesses by a core, the authors used the PMCs programmed to count the LLC misses. Yun et al. later extended these ideas to develop MemGuard [88], a memory reservation mechanism that statically partitions memory bandwidth between cores. Memguard divides the memory bandwidth between as guaranteed and best effort. Once all cores have exhausted their guaranteed bandwidth, they compete with other cores for the best effort bandwidth. Behnam et al. [89] proposed a multi-resource server-based approach on Freescale P4080 processor, where the tasks are guaranteed both CPU bandwidth and memory bandwidth.

Nowotsch et al. [65] proposed the concept of interference-sensitive WCET (isWCET), which extends the classical WCET to account for shared resource contention in COTS multicore processors. To determine the isWCET, the authors analyzed the WCET and resource usage of tasks in isolation and computed (offline) the shared resource interference delays resulting from co-executing tasks. Moreover, they complemented the offline resource usage analysis with an online mechanism based on monitoring to bound the maximum resource contention caused by a task (core). Based on the observation that the average case WCET and resource access of tasks are significantly lower, Nowotsch and Paulitsch [90] proposed an extension to the previous work to improve the utilization of the multicore platform significantly. For a slot-based TT system executing on a COTS multicore, Agrawal et al. [91] proposed two servers per core and use the isWCET concept. One server controls the processor time, and the other manages the memory bandwidth. Thus, both servers jointly control the contention between cores and the memory accesses per slot.

Newer MPSoCs contain some IP blocks, such as the interconnect and the IMC, with QoS mechanisms. The QoS mechanism aims to allow throttling the access of CPU cores, DMA, GPUs, and other I/O devices to the IP blocks. Serrano-Cases et al. [92] performed qualitative and quantitative analysis on various Quality of Service (QoS)-enabled IP

blocks on a Zynq UltraScale+ MPSoC [93]. Their analysis suggested that a system designer can exploit the QoS mechanisms in the IP blocks at design time to control contention in shared resources at runtime.

Furthermore, there are approaches based on task profiling; for example, Envelope-aWare Predictive model (E-WarP) [94] is a framework to profile tasks executing on CPU cores and accelerators. The framework bounds the execution time of tasks by monitoring DRAM activity via PMCs and enforcing memory-bandwidth regulation by a combination of Memguard [88] (for CPUs) and ARM QoS Support [95] (for accelerators). In DNA [96], the authors built an execution profile of tasks by analyzing their resource usage patterns and broadly dividing the them into phases based on these patterns. Then, a runtime mechanism dynamically allocates resources to tasks based on identification of tasks' phases. However, DNA only considers simple phase analysis and leaves considering complex task behavior in phase analysis to future work.

Approaches such as [97] aim to minimize cache evictions by providing mechanisms for cache partitioning. It is possible to implement these approaches in an Operating System (OS) for COTS multicore processors. [98] provides a comprehensive survey of cache partitioning techniques. Some OSs, such as FreeBSD [99], use page coloring [100] techniques to achieve cache partitioning and improve the performance of shared caches. However, Linux does not implement cache partitioning. As demonstrated in [101], page coloring techniques are only beneficial in multicore processors in the presence of extra hardware support. Furthermore, cache partition techniques often place additional constraints on memory space allocation and increase virtual memory allocation complexity and overheads. Recently, newer Intel Xeon processors [102] and AMD Zen2 processors [103] provide hardware-implement Cache Allocation Technology (CAT) to support Last Level Cache (LLC) partitioning and allocation to cores.

Many works propose techniques for DRAM bank partitioning to avoid bank sharing among cores for improving isolation on multicore processors. However, they require making changes to the existing hardware, and thus, they do not apply to COTS processors. PALLOC [104] is a technique to dynamically partition DRAM banks without requiring any hardware modifications. PALLOC extends Linux virtual memory system to allocate memory pages of tasks to specific DRAM banks. Specialized virtualization technologies (e.g., XtratuM hypervisor [15]) provide strong spatial isolation at the DRAM level by allowing a system designer to allocate memory regions to specific Virtual Machines (VMs) (in addition to temporal partitioning at the CPU level). In Chapter VI of this dissertation, the mixed-criticality avionics demonstrator uses the XtratuM hypervisor. Section II.17 explains virtualization technologies further.

II.16 Modes

Most RTS and MCS exhibit multiple operation phases. In each phase, applications may have different behavior and run different sets of tasks. For example, in an avionics system, a different set of tasks execute during take-off, landing, and cruising. RTS and MCS realize these operational phases as modes. Each mode contains a set of applications consisting of a set of tasks. Some examples of common modes observed in different RTS

and MCS are as follows:

- Initialization mode: Almost all systems require this mode to initialize different hardware and software components.
- Maintenance/Recovery mode: Systems require this mode to perform recovery or diagnostic operations or reinitialize faulty applications

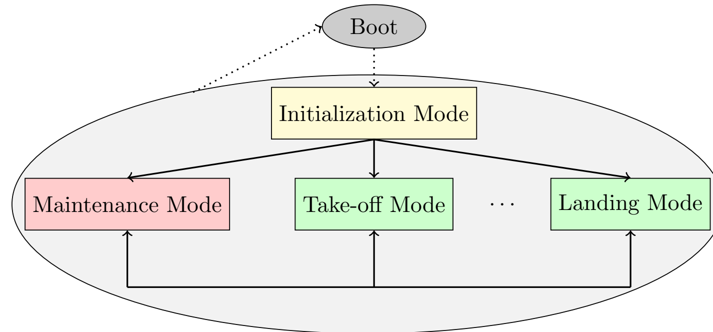


Figure II.F9: *Example of Modes*

Figure II.F9 shows a simple example of modes in an avionics system. Tasks in each mode can take a drastically different amount of execution time. As a result, a system designed without mode awareness can exhibit large latencies and jitter for critical applications, leading to system failure. RTS and MCS can provide better latency and jitter if they account for mode changes. For example, In a TT system, Fohler [105] captured mutually exclusive operational phases as a result of a change in the system or the environment via modes. A scheduling table represented each mode. Kopetz et al.[106] and Heilmann et al. [107] used modes to consider different operating phases in TT networks.

A system with multiple modes switches modes at runtime when it encounters a trigger, such as a change in environment or internal state. When the system encounters a trigger, it produces a mode change request. Upon seeing a mode change request, the system transitions from the source mode to the appropriate destination mode. Previous work [105, 106] refers to this process of transitioning between two modes as mode change. Jahanian [108] defined three variations of mode changes depending on the handling of currently executing tasks. The first possibility is to abort all ongoing tasks of the source mode, immediately perform a mode change and start running the tasks of the destination mode. Contrarily, the second possibility is to entirely execute all ongoing jobs of tasks from the source mode and then switch to the destination mode. Finally, the third option is to selectively finish some jobs and then switch to the next mode. Fohler [105] extended this concept by including transition modes that handle the termination of activities of the source mode during mode changes. Kopetz et al. [106] called a mode change as deferrable when the mode change can occur only at the hyperperiod, i.e., when all ongoing jobs of tasks from the source mode finish executing. They refer to a mode change as immediate when the mode change occurs soon after encountering the mode change trigger. Some specialized hypervisors for RTS and MCS, such as XtratuM [15],

support different operating modes and mode changes (see Section II.17.4). However, they refer to modes as scheduling plans and mode changes as plan changes.

II.17 Virtualization

Virtualization is the technique by which we can run multiple simulated computer system instances on a single (physical) computer system. To enable virtualization, we deploy a software virtualization layer on top of a physical computer system that provides multiple virtual environments called VMs. Each VM can use an instance of all or some of the physical resources of a computer system (e.g., CPU, memory, and network interface). We refer to the physical computer system as the host and the VMs running on the physical system as the Guests. It should be noted that in specialized avionics hypervisors, each ARINC 653 application partition runs as a VM of the real-time hypervisor. Hence, in the context of avionics, we use the term VM and partition interchangeably.

II.17.1 Hypervisors

The virtualization layer consists of a hypervisor that manages the physical resources and separates them from the VMs. The hypervisor is responsible for partitioning the physical resources and allocating the appropriate resources to VMs as per requirement. We refer to these resources as virtual resources. When a VM access a virtual resource, the hypervisor schedules the resource access on an appropriate physical resource. A hypervisor can easily reallocate resources between existing guests or allocate them to completely new guests. Moreover, Many hypervisors support running existing legacy applications without the need of modifying them. [109] provides a state-of-the-art survery on hypervisors.

A virtualization layer can use two different types of hypervisors:

1. Type-1 or bare-metal hypervisor executes directly on top of the physical hardware. A type-1 hypervisor directly allocates resources to a VM. Kernel Virtual Machine (KVM) [110] and Xen [111] are two popular type-1 hypervisors.

2. Type-2 or hosted hypervisor executes on top of an operating system. A Type-2 hypervisor must request the underlying OS to allocate resources to a VM. Oracle Virtual Box [112] is an example of type-2 hypervisor.

Enterprise data centers and other server-based environments often use type-1 hypervisors. In addition, type-1 hypervisors are also gaining popularity in real-time and mixed-criticality systems. In this dissertation, we only consider type-1 hypervisors as they suit the considered use cases better. Figure II.F10 shows a type-1 hypervisor.

II.17.2 Virtualization techniques

There are three main types of virtualization techniques: *full virtualization, hardware-assisted virtualization, and paravirtualization.*

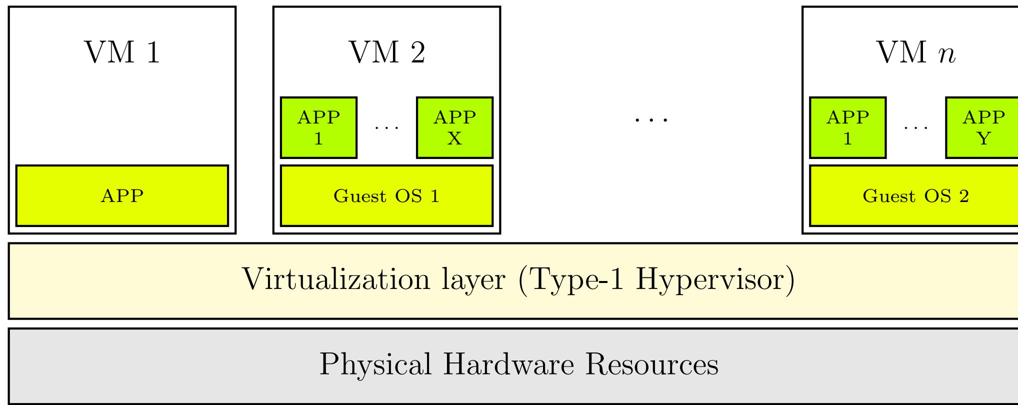


Figure II.F10: *Type-1 (Bare metal) Hypervisor*

In full virtualization, the application running in a guest VM is unaware that it runs on top virtualized resources. In this technique, the hypervisor uses a combination of binary translation and direct execution. Binary translation replaces certain privileged operations carried out by the guest application code (parts of kernel-level code in the case of a guest OS) at runtime to ensure the guest runs on the virtualized resources as intended. Binary translation provides complete decoupling between the guest VM and the hardware, and thus guest application needs no modifications to run it inside a VM. However, binary translation adds significant overheads. The remaining guest code (mostly user-level code in the case of a guest OS) runs directly on the underlying physical resource. Direct execution helps to improve the performance. Most commercially available generic hypervisors such as Xen and KVM support full virtualization.

Recently hardware vendors are introducing virtualization extensions at the hardware level. Intel VT-x technology [113] is an example of hardware-assisted virtualization extension. Such extensions aim to either eliminate or at least reduce the need for binary translation. Whenever a guest application performs a privileged operation, the hardware automatically traps it and invokes the hypervisor to handle the operation. However, this technology is still in its infancy. Some hardware virtualization extensions have lower overheads, while other extensions have higher overheads than binary translation [114]. The high overheads are a result of the guest-hypervisor transitions involved in the approach. Hypervisors depending purely on hardware-assisted virtualization can only execute on hardware platforms supporting the particular virtualization extension. We use Intel VT-x technology in the railway usecase (Chapter VII) for reducing some virtualization overheads.

Paravirtualization (also know as OS-assisted virtualization) requires communication between the guest application and the hypervisor. Thus, the guest must be aware that it is running on top of a virtualization layer. The guest application designer must replace privileged operations with *hypercalls* provided by the hypervisor. A hypercall is a request by a guest for the hypervisor to perform some service required by the guest. As a result, the hypervisor does not need to use the high overhead binary translation approach. The changes necessary to the guest applications for paravirtualization are often easy to make. Most commercially available hypervisors support paravirtualization for some parts of

the guest application. For example, KVM provides Virtio [115] network and disk drivers that a guest application designer can install in the Guest OS. However, many of them do not support paravirtualization solutions at the CPU level. Specialized hypervisors often support paravirtualization solutions to meet domain-specific requirements. For example, XtratuM [15] and PikeOS [16] provide hypercalls to support ARINC 653 style communication channels with sampling and queuing modes. We use such hypercalls for development of the avonics use case in Chapter VI.

Virtualization technologies form the backbone of cloud computing. The following section provides more information on cloud computing.

II.17.3 Hypervisors for Real-time and Mixed-Criticality Systems

At present, RTS and MCS research communities are directing considerable efforts towards providing robust partitioning on multicore platforms, for example, [116], [117], and [118]. Since hypervisors are a efficient way to build partitioned systems, RTS industries are starting to use hypervisors for deploying real-time and non-real-time applications as VMs on the same multicore platforms. Similarly, MCS industries consider hypervisors for deploying different criticality level applications as VMs on the same multicore platforms. Hypervisor help to achieve the the following main advantages:

- Hypervisors help increase resource utilization and reduce SWaP and wiring costs by hosting multiple VMs concurrently on the same multicore node (hardware consolidation). In turn, hardware consolidation results in a reduction in maintenance cost and fewer spare parts. Hypervisors provide mechanisms for improved resource sharing and help to allocate resources efficiently among various VMs.

- Hypervisors can partition the underlying hardware and ensure resource isolation among the VMs. Depending on resource isolation (and allocation) mechanisms implemented in a hypervisor, a hypervisor can limit the resource contention among VMs and ensure that specific faults in a VM do not impact other non-faulty VMs. Moreover, hypervisors can also assist in providing fault-tolerance to applications [119].

- Hypervisors abstract the underlying hardware from guest VMs. As a result, they allow easier reuse of legacy applications (and operating systems) and help overcome hardware obsolescence issues. Moreover, running legacy applications as VMs on a qualified hypervisor helps reduce recertification costs.

- Hypervisors also make it easier to monitor the VMs.

Generic hypervisors, i.e., hypervisors not specially designed for RTS or MCS already provide some resource allocation strategies beneficial to RTS. For example, KVM and Xen can provide temporal isolation to a certain extent at the CPU-level by using SCHED_DEADLINE (EDF with Constant Bandwidth Server (CBS)) [120] and RTDS [121], respectively. Vanderleest introduced an ARINC 653 CPU scheduler in Xen [122] to support the development of ARINC 653 applications. They also offer spatial isolation

to some extent by allowing to reserve memory regions for VMs. We present further details and in-depth comparison between some generic hypervisors in Chapter VII and present methods to use a generic hypervisor (KVM) for a safety-critical railway use case in Chapter VII.

The resource allocation mechanisms in generic hypervisors are not always sufficient to meet the stringent requirements of some RTS and MCS domains. We need to use specially developed type-1 hypervisors, such as XtratuM [15] and PikeOS[16], that provide strong temporal isolation at CPU level and spatial isolation at DRAM level. Moreover, these hypervisors adehere to industy standards such as ARINC 653 [45] or ISO 26262 [32]. [123] explains how a specialized hypervisor such as XtratuM can assist in safety certification for industrial mixed-criticality systems.

II.17.4 XtratuM Hypervisor

XtratuM [15] is a free and open-source hypervisor for use with multicore processors and MPSoCs. XtratuM aims at meeting the requirements of MCS. Moreover, XtratuM supports execution environments partly compliant with the avionics ARINC 653 standard. At the time of writing this dissertation, 220 satellites are flying using the XtratuM hypervisor. In Chapter VI, we developed the global resource management framework for the avionics demonstrator on top of the XtratuM hypervisor. Nevertheless, it is possible to extend the ideas discussed in the dissertation for implementation on top of other hypervisors.

XtratuM abstracts the underlying hardware and supports execution of multiple partitions with Time and Space Partitioning (TSP) as VMs[1]. XtratuM allows two kinds of partition corresponding to ARINC 653 system and application partitions: system partition and user partition.

XtratuM provides the following properties relevant for this dissertation:

- *Spatial isolation*: Based on input from the system designer, XtratuM allocates a uniques memory region (address space) to a partition for storing the partition's code, data, and stack. This memory region is not accessible by other partitions. However, XtratuM allows a system designed to define memory regions that partition may share.

- *Temporal isolation*: XtratuM schedules partition on a cyclic basis (ARINC 653 scheduling policy) using an offline defined table (in the form of a configuration file provided by the system designer). It ensures that a partition cannot run longer than its allocated time slot.

- *Fault isolation and management*: XtratuM implements a fault management model to detect and handle faults in partitions. It ensures that faults in a partition do not propagate to other partitions. Moreover, it provides facilities to partition to manage faults concerning the partition's execution. XtratuM has a Health Monitor

[1]In the context of XtratuM, the terms VM, guest, and partition are synonymous [15]

(HM) that helps in the early detection of faulty states or behaviors and carrying out recovery methods defined by the system designer.

- *Predictability*: XtratuM provides predictability concerning the operation of partitions and management of interrupts.
- *Security (and confidentiality)*: XtratuM ensures that all information of system and partitions cannot be modified (or read) by unauthorized partitions.
- *Inter-Partition Communication*: XtratuM provides encapsulated ARINC 653 style communication channels with sampling and queuing ports. A system designer must define channels and ports for every partition at design time.
- *Interrupt management*: XtratuM provides an interrupt model to partitions and adds 32 new sources of interrupts based on specific events.
- *Tracing*: XtratuM provides a mechanism to store traces generated by partitions and the hypervisor itself.
- *Hypercalls*: XtratuM provides hypercall APIs to enable paravirtualization.

Scheduling Plans

A single scheduling plan (table) may be too restrictive. For example, the initialization or maintenance of an application can need a different number of slots. However, the system designer typically defines a plan with the regular operation of the applications in mind. Thus, XtratuM allows the system designer to define multiple scheduling plans (via a configuration file) that allow reallocating CPU time to partitions in a predictable way. XtratuM uses mode changes to change scheduling plans. XtratuM reserves some scheduling plans as follows.

1. XtratuM reserves *Plan 0* for the initialization of applications in all partitions. The system starts in *Plan 0* and remains there until a (ARINC 653) system partition requests a plan change. The plan switch occurs after all remaining slots of the current plan finish execution. Henceforth, the system cannot go back to *Plan 0*.
2. XtratuM reserves *Plan 1* as a maintenance plan. XtratuM switches immediately to *Plan 1* when the HM encounters a condition defined at design time that requires switching to a maintenance plan or upon receiving a request from a system partition.

A system designer can define *Plan X* ($X > 1$). A system partition can request a switch to them at any time. When switching plans, the switch is not immediate. The plan switch occurs after all remaining slots of the current plan finish execution. Our global resource management framework for the avionics demonstrator (Chapter VI) exploits these scheduling plans to provide fault-tolerance in a MCS at runtime.

II.17.5 Hierarchical Scheduling in Hypervisors

Often there are two levels of scheduling hierarchy involved when using hypervisors. On the top level of the hierarchy, the VM scheduler of the hypervisor allocates physical resources to the VMs for a fixed amount of time or budget using a specific scheduling policy. Then, in the next level of the hierarchy, a *task scheduler* running in a VM assigns the resources to the tasks of that VM. So, naturally, a task scheduler can only run and assign the resources to a VM's tasks when the VM scheduler allocates resources to the particular VM.

In specialized avionics hypervisors, we refer to the VM scheduler of the hypervisor as a *partition scheduler*. The partition scheduler is a cyclic scheduler that divides the CPU execution time into well-defined time slots and assigns these slots to partition during their MiF(s) based on an offline scheduling table (the partition scheduler repeats the scheduling table every MaF). During each time slot assigned to a partition, the task-level scheduler running in a partition divides the allocated CPU time among the tasks of that partition. The task-level scheduler is often in the form of a cyclic executive or a part of a small real-time operating system running in the partition. Figure II.F11 shows an example of hierarchical scheduling in avionics. We use this technique for scheduling tasks of avionics applications in Chapter VI.

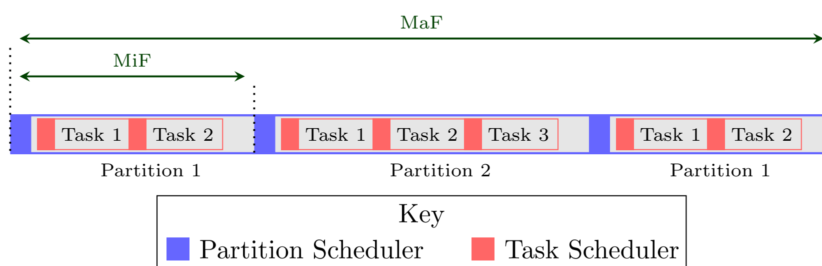


Figure II.F11: *Hierarchical Scheduling in Avionics*

II.18 Cloud Computing

Cloud computing is the on-demand delivery of computing resources over the internet. Hosting applications in the cloud supports ease of re-usability, reconfiguration, and scalability while providing higher availability, less need for maintenance, and reduced running costs. Moreover, cloud computing helps organizations reduce their carbon footprint by letting them reduce resource over-provisioning. Clouds use hypervisors to provide VMs to the users for hosting applications. The applications are neither aware that they are running in a virtualized environment nor aware of the actual underlying hardware resources. Thus, the reuse of existing applications is usually possible without modifications by directly executing them in a VM hosted on a cloud. In [124], the authors discuss various aspects of cloud computing technology, including definitions, features, and enabling technologies.

II.18.1 Public vs. Private Clouds

Public clouds are the most common form of cloud computing. A third-party service provider owns the computing resources of a public cloud and manages all hardware, software, and other infrastructure requirements. In the public cloud, a cloud user shares the cloud resources with other cloud users. VMs of different users can share a single node. The main advantages of public clouds are lower costs, no need for maintenance, and very high scalability. Some popular public cloud service providers are Amazon Web Services, Google Cloud, and Microsoft Azure.

A *private cloud* consists of cloud computing resources dedicated to a single organization. The organization can have the cloud computing resources on its premises or from a third-party provider. In any case, the organization has dedicated access and control over the hardware and software. Private clouds make it easy for organizations to meet specific IT requirements. The organization can do all maintenance over a private network. The main advantages of private clouds are more flexibility and control than public clouds while still offering scalability.

RTS and MCS industries, such as railways, want to benefit from the advantages of cloud computing. Public clouds do not ensure predictable execution of safety-critical or real-time VMs a third-party cloud service provider allocates resources among multiple users without taking into account the fine-grained requirements of the VMs. [125] presents some of the challenges in predictable cloud computing. However, in private clouds, an organization has dedicated access and complete control over the cloud resources. In Chapter VII of this dissertation, we propose a resource management layer to leverage a private cloud for supporting the execution of Triple Modular Redundancy (TMR) safety-critical railway VMs running the Thales TAS control platform. The next section explains fault-tolerance, including techniques such as Triple Modular Redundancy (TMR).

II.19 Fault, Error, Failure

A *fault* is a defect in a hardware or software component. Faults remain dormant until an event occurs that activates them. An *error* is an incorrect internal state that a system may encounter during its operation (unobserved) when an event occurrence triggers a fault. A *failure* is an incorrect behavior (observed) of the system (compared to the expected behavior). When errors propagate, they lead to failures in the system. Faults are classified as permanent, transient, and intermittent. *Permanent faults* persist indefinitely (or until repaired) after their occurrence (e.g., incorrectly implemented or damaged component). *Transient faults* persist for a short period and disappear (e.g., radiation-induced faults). *Intermittent faults* occur at irregular intervals in a system that otherwise functions correctly.

II.20 Fault-tolerance

Fault tolerance refers to a system's capability to stay operational without disruption despite faults in one or more components. The main goal is to prevent a single point

of failure and ensure high availability for safety-critical components. Unfortunately, it is not always economically viable to design a system that can tolerate any fault that may occur. Therefore, a system designer must precisely define a fault hypothesis specifying assumptions, such as types and number of faults, the fault containment regions, and error detection latency. Such a fault hypothesis helps ensure that the system implementation uses an appropriate system design and validation concepts [126] and supports the evaluation of assumption coverage [127]. A fault-tolerant system must ensure the integrity of the output data, detect errors caused by faults, evaluate the repercussions of the faults, and provide error recovery and fault isolation.

Fault tolerance is often based on some form of redundancy. Redundancy involves replicating components in a system to provide fault tolerance and increase the system's reliability. Wensley [128] originally introduced the concept of software redundancy. Redundancy often uses *Consensus* for the decision-making process in which the redundant components agree to support an acceptable output for specific input. Consensus requires some form of voting.

II.20.1 N-Modular Redundancy (NMR)

N-Modular Redundancy (NMR) is a technique to provide fault-tolerance where N identical software components run in the SCS. The identical software components receive the same input data. The SCS collects the output data produced by the replicas and uses majority-voting to mask any potential faults in one or more components. For the voting to be effective, the system requires N $>=$ 3.

Triple Modular Redundancy (TMR) is the most well-known version of N-Modular Redundancy (NMR) where $N = 3$. The main aim of TMR is to keep the system operational when a single fault, such as a single event upset or hardware wear-out, occurs in the reference frame of the fault hypothesis. Depending on the independence of the replicas (for example, if the replicas run on the same computer system or three different computer systems), the system can achieve different degrees of fault detection and fail-over. The Thales TAS control platform [129] used in the railway use case (Chapter VII) is an example of an industrial TMR-based system.

II.20.2 Reconfiguration for Fault-Tolerance

Classical redundancy-based solutions such as TMR require substantial hardware replication to provide the required degree of fault-tolerance and safeguard the systems against failures. Unfortunately, these approaches do not exploit the system's residual processing capability to achieve fault tolerance. However, it is possible to monitor the system to detect hardware component failures and use runtime reconfiguration for error recovery and maintenance of the desired system state. Dynamic (runtime) reconfiguration enables a system to dynamically adapt to resource failures. This section gives an overview of the reconfiguration strategy.

It is possible to view a system as being composed of two levels:

1. At the physical level, a distributed system consists of a set of resources such as the processing elements and the network connecting the processing elements.

2. At the logical level, the system consists of functionalities and the communication channels between them.

A system configuration is a mapping of physical elements to logical ones. For example, functionalities and communication channels can be mapped to the physical processors and the network elements. Reconfiguration involves mapping then the logical elements elsewhere in the distributed system when a component fails. For example, when a processor executing a functionality fails, a system reconfiguration takes place to allocate another available processor in the distributed system to the functionality and reallocate network elements to communication channels accordingly. Thus, reconfiguration not only allows to move the functionalities from fault components to non-faulty ones but also enables the possibility to maintain spare components among functionalities collectively. Overall, reconfiguration maximizes the use of system hardware while ensuring the system is operational.

Porcarelli et al. [130] proposed the Lira (Lightweight Infrastructure for Reconfiguring Applications) framework for fault-tolerance in distributed systems based on reconfiguration. The reconfiguration in safety-crtical systems should be predictable. Strunk et al. [131] proposed reconfiguration to improve the dependability of fail-stop safety-critical systems. Strunk and Knight [132] proposed a primary system design with low dependability and complex applications with the possibility to reconfigure to a simple but highly dependable system upon failures. Ellis [133] demonstrated the viability of reconfiguration to provide hardware fault tolerance in safety-critical systems. The SCARLETT [134] and DIANA [135] project reconfiguration strategies for IMA platforms based on single-core processors. SCARLETT project proposed a centralized reconfiguration approach, while DIANA project proposed a decentralized approach.

II.20.3 Byzantine fault tolerance (BFT)

A Byzantine fault [136] can occur in a system that requires consensus. A redundant component (or node) with a Byzantine fault presents different output values to different redundant components (or nodes) for the same input leading to disagreement during voting. A Byzantine failure occurs when a system stops operating correctly due to byzantine faults. Byzantine Fault Tolerance (BFT) aims at protecting systems against Byzantine failures. The fault hypothesis for BFT does not impose any restrictions or make assumptions about the behavior of nodes. Moreover, this fault is notoriously tough to tolerate as it is an asymmetrical fault.

Let us consider an system with a set of replicated components $R \in \{r_1, r_2, \ldots, r_n\}$. Assume that we provide all replicas belonging to set R with the same input, I. If a replica $r_r \in R$ broadcasts the value of the output O_r corresponding to input I to other replicas belonging to set R, then in Byzantine fault-tolerant system:

1. all honest replicas $\in R$ agree on the value O_r broadcasted by r_r for input I.

2. all honest replicas $\in R$ agree on the same value of the output O for input I.

BFT algorithm such as the Practical Byzantine Fault Tolerance (PBFT) algorithm [137] requires $n = 3f + 1$ total replicas in a system with f faulty replicas to guarantee *safety* and *liveness*. In the context of PBFT, safety means that the replicated components behave like a centralized component executing one operation at a time in an atomic manner, and liveness means that the client requesting the service from the replicated component eventually receives a reply to their request.

Sawtooth [138] blockchain platform can use the PBFT algorithm for Byzantine fault-tolerant consensus. In Chapter V, we use Sawtooth for implementing our distributed global resource management framework.

II.21 Blockchain

Blockchain is a type of Distributed Ledger Technology (DLT) with a growing list of blocks that allow recording transactions and the associated data. *Blocks* are basic data structures containing the timestamp, transaction data, and the previous block's cryptographic hash. A *cryptographic hash* value is mapping data to a fixed-size bit map (compactly stored array of bits) by an algorithm called a cryptographic hash function. For adding (*committing*) new blocks, the blockchain must reach a consensus among participating nodes using consensus mechanisms such as PBFT. The committed blocks form a chain as all blocks contain the information of all prior blocks, thus reinforcing the ones that came before them. Hence the data recorded in a single blockchain block cannot be modified without reconstructing all the subsequent blocks. This property makes the data stored on the blockchain immutable. Moreover, blockchain prevents a single point of failure by maintaining copies of the distributed ledger on multiple (or all) distributed system nodes. Blockchain has two types depending on who can participate in the blockchain network.

1. *Public (permissionless) blockchains* allow anyone node to join the blockchain network and access the blockchain data or request to commit new data on the blockchain. In addition, all nodes in the blockchain network can participate in the consensus as well.

2. *Private (permissioned) blockchains* set restrictions on which node can participate in the network. Moreover, they can restrict nodes from participating in certain transactions. Private blockchains are suited to application domains that require an additional level of security, privacy, and performance.

Smart Contracts

Smart contracts are self-executing simple programs stored on the blockchain. They execute automatically on nodes participating in the blockchain network consensus mechanism when some predetermined conditions occur. Once the smart contracts finish execution, the nodes update the result on the blockchain (via consensus).

Hyperledger Sawtooth

Sawtooth [138] is a private blockchain platform with support for smart contracts. An open-source collaborative effort called the Hyperledger project [139] hosted by the Linux foundation developed the Sawtooth blockchain platform. Sawtooth supports the PBFT algorithm for Byzantine fault-tolerant consensus. It provides a highly modular, scalable, and secure platform for implementing transaction-based updates to a shared ledger. Furthermore, Sawtooth separates the core functionality of the blockchain from the application level by allowing users to create Sawtooth transaction families (containing a smart contract, client, and data model) that can be deployed on top of the blockchain platform. In Chapter V, we present an implementation of a new transaction family for Sawtooth called the *DGRM transaction family* to realize components of our safe and secure distributed global resource management framework. We implemented the distributed global resource manager as a smart contract in this new transaction family.

II.22 Relevant EU Projects

This section gives an overview of completed and ongoing EU projects relevant to the topics considered in this dissertation.

- GENESYS (Generic Embedded System Platform) project [140] provided a cross-domain architecture with fixed core services and multiple optional services implemented as self-contained system components. The project targeted industrial challenges, such as integrated resource management, in automotive, avionics, industrial control, mobile, and consumer electronics domains. A follow-up project, INDEXYS (INDustrial EXploitation of the genesYS cross-domain architecture) [141], aimed to develop MPSoC based on GENESYS reference architecture.

- FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) aimed to develop a framework that integrates advanced flexible scheduling techniques directly into the embedded systems design methodology. It covered all the levels involved in the implementation, starting from the OS primitives, through the middleware, up to the application level.

- DIANA (Distributed equipment Independent environment for Advanced avioNic Applications) project [135] took one of the first steps towards developing a platform for execution of avionics applications as VMs. In addition, the project also incorporated a model-driven engineering approach.

- ACTORS (Adaptivity and ConTrol Of Resources in embedded System) project [22] addressed the challenging problem of efficient design of embedded systems for complex and demanding high-performance applications. The project approach raised the abstraction level of the specifications and computing models, including resource-constrained design exploration stages and real-time resource adaptation by developing the appropriate models and tools supporting the design from the specification down to the embedded implementation.

- CESAR (Cost-Efficient Methods and Processes for SAfety Relevant Embedded Systems) project [142] provided improved methods, tools, and demands for embedded systems development in Aerospace, Automotive, Automation, and Railways. CESAR addressed the entire system engineering life cycle by improving its disciplines and implementing fundamentals for interoperability in a reference technology platform as an integrated tool platform.

- MULCORS (The Use of Multicore Processors in Airborne Systems) project [14] provided guidance to EASA (European Union Aviation Safety Agency) on the architecture, advantages, and adverse effects of multicore processors in airborne systems. The project also provided recommendations on handling issues related to multicore processors.

- SCARLETT (Scalable and reconfigurable electronics platforms and tools) project [134] implemented the innovations in distributed modular electronics concept to improve scalability, portability, adaptability, fault tolerance, and reconfiguration capabilities in the current IMA.

- ACROSS (Artemis CROSS Domain architecture) project [23] aimed to realize a cross-domain multicore chip with the service architecture defined in the GENESYS project. It included an FPGA-based multi-processor system on a chip implementation of a TT NoC, tailored middleware components, and flexible embedded tools.

- MultiPARTES (Multi-cores Partitioning for Trusted Embedded Systems) project [117] aimed at developing tools and solutions for building trusted embedded systems with mixed-criticality components on multicore platforms.

- vIrtical (SW/HW extensions for virtualized heterogeneous multicore platforms) project [143] aimed the vertical and full development of the virtualization concept addressing the specific requirements for effective embedded virtualization. A virtualization-ready SoC platform and the associated programming models are developed, tackling all the system layers: applications, programming model, hypervisor, and hardware.

- ARAMiS (Automotive, Railway, and Avionics Multicore System) project [144] goal was to improve the operational safety of automobiles, trains, and airplanes. Timing, determinism, influence on safety, real-time applications, and certification were significant fields of research. A follow-up project, ARAMiS II, [145] started in 2016, aims at development processes, tools, and platforms for efficient use of multicores in the industry.

- CONTREX (Design of embedded mixed-criticality control systems under consideration of Extra-functional properties) project's [146] main challenge was to guarantee timing, power, temperature, and reliability requirements by controlling (shared) resource usage and access on the execution platform. It has also

considered extra-functional constraints right from the beginning, represented extra-functional properties in executable prototypes, and included them in local and global scheduling and control decisions.

- EMC^2 (Embedded Multicore systems for Mixed-Criticality applications in dynamic and changeable real-time environments) project [147] explored solutions for dynamic adaptability in open systems, provides handling of mixed-criticality applications under real-time conditions, scalability and utmost flexibility, full-scale deployment, and management of integrated toolchains, through the entire life cycle.

- SAFURE (Safety And Security By Design For Interconnected Mixed-Critical Cyber-Physical Systems) project [148] targeted the design of cyber-physical systems by implementing a methodology that ensures safety and security by construction.

- SAFEPOWER (Safe and secure mixed-criticality systems with low power requirements) project [149] aimed to combine power awareness with predictability, TSP, reliability, and security for use in cross-domain MCS. The project considered use cases from the avionics and railway domain to demonstrate the proof-of-concept implementation for low-power MCS.

- De-RISC (Dependable Real-time Infrastructure for Safety-critical Computer) project [150] aims to develop a platform based on the RISC-V multicore architecture and a safety-critical hypervisor for safety- and security-critical systems for use in the aerospace domain. This platform will primarily focus on mitigating interference in a RISC-V SoC.

The work presented in this dissertation contributed towards two EU projects: DREAMS and SECREDAS. The following two sections give a high-level overview of these projects.

II.23 DREAMS Project

DREAMS (Distributed REal-time Architecture for Mixed criticality Systems) project, funded by the European Commission, aimed to develop a cross-domain architecture and design tools for MCS executing in a distributed system with nodes consisting of multicore processors and MPSoCs. The four-year project had 16 partners from renowned academic institutes and industries across Europe.

The DREAMS cross-domain architecture was structured as a waistline inspired by the Internet. The internet protocol acts as a stable waist for supporting various communication technologies (e.g., Ethernet and wireless networks) below the waist and different protocols (e.g., User Datagram Protocol (UDP) and Transmission Control Protocol (TCP)) above the waist. Similarly, the core services form the stable waist of the DREAMS architecture and encapsulate all the domain-dependant and domain-independent services needed in three mixed-criticality targeted application domains: Avionics, Windpower, and Healthcare. Figure II.F12 shows the DREAMS waistline architecture.

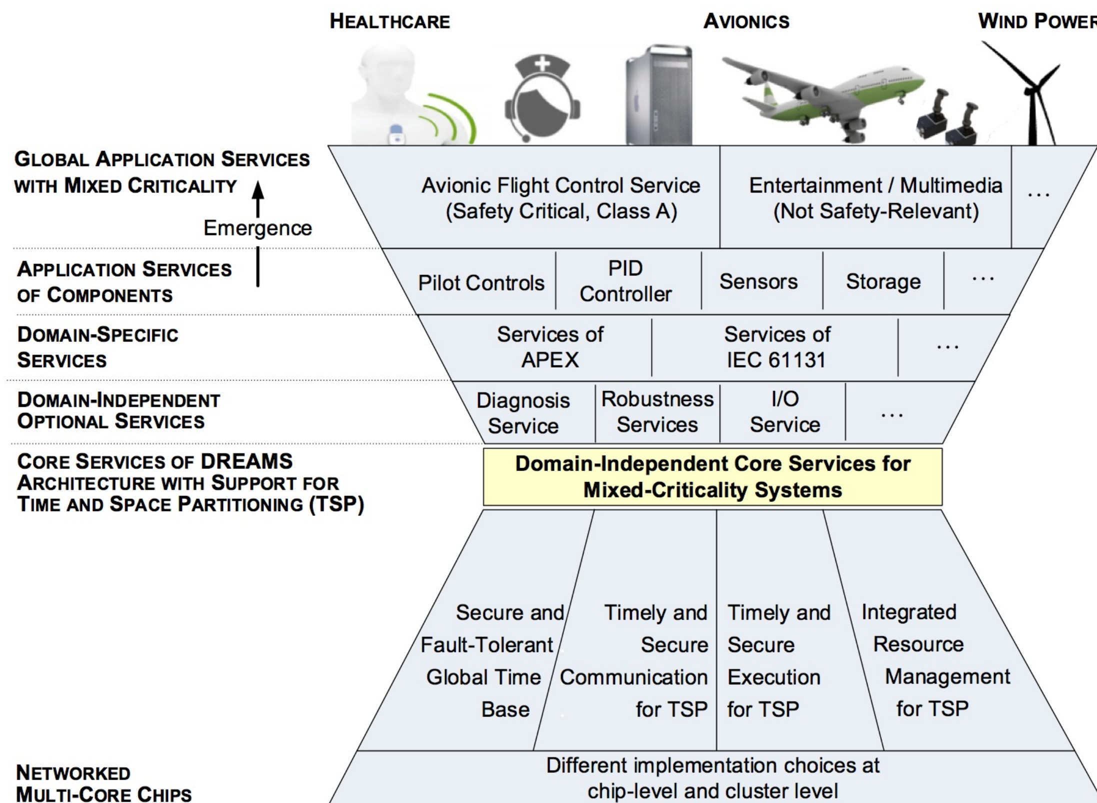


Figure II.F12: *DREAMS Architecture [2]*

Four core services are required to instantiate a DREAMS architecture and build higher-level services to maintain the desired architecture properties, such as TSP. These four services are important for MCS and all three application domains. The four core services are:

1. *Secure and fault-tolerant global timebase:* The global timebase core service provides a local time to each node and other components of the distributed system. The global timebase is synchronized among all distributed system nodes and within all sub-components of each node. Similar to the TTA, the global timebase facilitates the temporal coordination of activities, establishes deterministic communication infrastructure, and links timestamps from different components.

2. *Timely and secure communication services for TSP:* This core service helps to create end-to-end channels for message-based communication between components over a hierarchical and heterogeneous network with mixed-criticality traffic. This core service ensures that the DREAMS architecture meets the safety and security requirements of the MCS and facilitates TSP based on prior knowledge about the time and value domain behavior of applications and components in the.

3. *Timely and secure execution for TSP:* This core service ensures static temporal partitioning of CPU cores and spatial partitioning at the memory level. Offline

tools create a pre-computed scheduling table for use by this core service. DREAM architecture realizes this core service using virtualization techniques.

4. *Integrated resource management for TSP:* On single nodes, pre-planned assumptions may be possible to ensure the safe and timely execution of multiple mixed-criticality applications. However, in DREAMS, multiple mixed-criticality applications execute in a distributed system with multiple heterogeneous nodes, each with multiple resources. In such systems, pre-planned assumptions become less meaningful. If the applications, resource availability, or system configurations change, obtaining realistic resource availability assumptions becomes difficult. Thus, the DREAMS architecture provides a global integrated resource management core service to handle system-wide constraints, such as end-to-end timing or reliability, without incurring the complexity and overhead of individual negotiations among resources directly. It also ensures system-wide adaptivity of mixed-criticality applications consuming multiple resources.

The work presented in this dissertation contributed to the development of the integrated resource management core service and its adaptation for the mixed-criticality avionics use case.

II.23.1 Avionics Demonstrator

During the DREAMS project, an avionics demonstrator was developed together with Thales Research & Technology and the French aerospace lab - ONERA. It was based on a realistic avionics use case and followed the DREAMS reference architecture. The demonstrator deployed five applications, three safety-critical and two non-critical, on top of networked multicore and MPSoC nodes. The critical applications were: Flight Management System (FMS), Display Management System (DMS), and Sensor Data Provider (SDP). The non-critical applications were In-Flight Entertainment (IFE) and Passenger and Cockpit panels.

The demonstrator highlighted the capabilities of the DREAMS core services, especially the reconfiguration capabilities of the integrated resource management core service. This dissertation contributed towards the tooling and integration of the resource management core service in the avionics demonstrator. Chapter VI provides a detailed description of the complete avionics demonstrator together with the integrated resource management core service.

II.24 SECREDAS Project

SECREDAS (Product Security for Cross-Domain Reliable Dependable Automated Systems) Projects, funded by ECSEL Joint Undertaking, aims to improve security, safety, and privacy in automated systems across three application domains: automotive, railway, and healthcare. The project has 70 partners from renowned academic institutes and industries across Europe. The main goal is to develop and validate multi-domain architecting methodologies, components, and suitable integration approaches for automated

systems. The project also considers standardization, certification, and qualification in the three domains, combining high security and privacy protection while preserving functional safety and operational performance.

II.24.1 Common Technology Elements (CTEs)

One of the key aims of the SECREDAS project is to identify, validate and, if needed, further develop several Common Technology Elements (CTEs) to ensure safety, security, and operational performance in safety-critical domains like railway. CTEs are existing industrial proven technologies (starting from Technology Readiness Levels (TRL) 7 [151]) and can be used to develop new security solutions in the different domains in SECREDAS. The CTEs are domain-independent and are not limited to concrete implementations but can also contain best practices and protocol specifications. Some examples of CTEs are virtualization technologies, cryptography libraries, or blockchain. The railway use case of SECREDAS leverages the hypervisor CTE to enable railway operation as a cloud-based service. This dissertation contributed towards the identification and experimental validation of the hypervisor CTE for the railway use case.

II.24.2 Railway Use Case

The railway use case of SECREDAS covers various safety and security aspects. This use case involves a fault-tolerant computing Platform called TAS [129], a technology platform developed by Thales to support various safety-critical transport applications. The TAS Platform hosts railway-specific applications and provides them with fault tolerance services such as time synchronization, membership service, voting, and fault management.

Safety-critical systems often suffer from hardware obsolescence and scalability issues. Moreover, they require high availability, redundancy, and ease of hardware re-usability and dynamic reconfiguration. Cloud computing can help resolve these issues and requirements. Thus, the SECREDAS project railway use case aims to explore hosting safety-critical VM in a cloud. The safety-critical VMs must execute TMR safety-critical railway applications using the Thales TAS platform [129]. TAS is a technology platform developed by Thales to support various safety-critical transport applications. The TAS Platform hosts railway-specific applications and provides them with fault tolerance services such as time synchronization, membership service, voting, and fault management. The ultimate goal of the railway use case is to enable railway operation as a cloud-based service.

For achieving railway operation as a cloud-based service, the use case first aims to show the technical feasibility of a virtualization approach using the hypervisor CTE. A vital advantage of this approach is hardware abstraction and the ability to run multiple virtualized safety-critical applications on one or more servers with COTS multicore processors.

The main contribution of this dissertation is towards the identification of requirements for safe and timely execution of safety-critical railway VMs, and the selection of cloud

virtualization technologies (generic hypervisors) for hosting these VMs. However, as demonstrated in this dissertation, cloud virtualization technologies alone are insufficient to ensure the predictable execution of multiple safety-critical VMs, especially in the presence of other non-critical VMs in the cloud. To this extent, we proposed to extend a generic cloud hypervisor (KVM [110]) with a resource management layer to monitor, control, and coordinate the cloud nodes and form a Real-Time Cloud (RT-Cloud). Such a RT-Cloud can guarantee predictable execution of safety-critical VMs while reducing the impact of resource over-provision on non-critical VMs. Based on some of the concepts discussed in this dissertation, Thales Austria GmbH, together with the Austrian railways (ÖBB), presented a live demonstration about hosting a safety-critical application (railway interlocking) in a KVM-based cloud [152]. Chapter VII provides a detailed description of the railway use case and the RT-Cloud with a resource managment layer.

II.25 Problem Statements

This section presents the problem statements considered in this dissertation.

II.25.1 Resource Management in Distributed Real-Time and Mixed-Criticality System

Multicore processors and MPSoCs nodes have become essential in RTS and MCS because their additional computing capabilities help reduce SWaP, required wiring, and associated costs. In distributed systems, a single shared multicore or MPSoC node executes several applications, possibly of different criticality levels. However, as seen in Section II.14, there is interference between applications due to contention in shared resources such as CPU core, cache, memory, and network. Therefore, such a system requires careful planning and management of resources to ensure interference in shared resources does not prevent the real-time applications from meeting their deadlines nor prevent safety-critical applications from meeting their safety assurance levels. Besides, it is necessary to ensure fault isolation and error contamination despite sharing of resources.

As explained in Section II.9 and II.8, many allocation and scheduling methods exist for RTS and MCS. These methods rely on implicit assumptions of the constant availability of individual resources, especially the CPU. Classical scheduling algorithms like Round-Robin (RR) or EDF assume control over the entire CPU or a single core. Methods such as XtratuM hypervisor [15] or PikeOS [16] schedule to fixed proportions and allow hierarchical scheduling. As mentioned in Section II.15, a variety of models, analysis techniques, and resource allocation approaches exist to deal with contention in shared resources. Most of these approaches do not address all the shared resources of a node and aim to resolve contention in only a specific shared resource or a set of specific shared resources. Moreover, they handle a limited number of events such as task arrivals and task completions and assume a constant amount of the availability of resources and demands of applications to provide guaranteed progress of tasks. Such assumptions may be possible on single nodes; however, they become less meaningful in distributed systems with several nodes, each having multiple resources. If the applications, resource

availability, or system configurations change, obtaining assumptions about resources becomes complicated. Moreover, it is challenging to meet end-to-end constraints by considering each node, resource, or application individually.

Existing OSs and hypervisors only consider a single node at a time, i.e., OSs and hypervisors on a node make local resource allocation decisions and manage tasks locally without a global view of the system. As a result, they cannot leverage the availability of resources in other nodes to make resource allocation decisions to meet end-to-end constraints, improve fault-tolerance, improve QoS of non-critical (or best-effort) applications, or increase the overall system-wide efficiency.

Modern RTS and MCS have other crucial differences compared to classical RTS resource allocation. For example, RTS in the healthcare domain and other indoor networks often consist of heterogeneous nodes, each with a diverse set of resources and nodes communicating on different types of networks. Using previous approaches that tightly couple global management of the entire network with node-level resource allocation is unsuitable for adaptability in the system as it has high overheads and requires a fixed set of devices and schedulers.

Finally, RTS and MCS have limited availability of resources which must host not only the real-time applications but also the schedulers and resource managers.

Such RTS and MCS with dynamically changing availability and demand of resources need ***resource management*** that maintains a global (system-wide) view of resources and applications and coordinates and dynamically adapts system-wide resource allocations. In addition, resource management can dynamically adapt applications to changing availability of resources. Moreover, the resource management should have low overheads and must decouple global management of the entire system with local (node-level) resource allocation.

The overall aim of resource management is to ensure real-time applications meet their end-to-end deadlines even in the presence of faults and changing environmental conditions and ensure efficient resource utilization to improve the QoS of BE (or non-critical) applications.

II.25.2 Exploitation of Multicore Processors for Fault-Tolerance (in Avionics)

As explained in II.20, safety-critical domains such as avionics and railway use classical redundancy-based solutions such as TMR to safeguard the systems against failures. Such redundancy-based techniques require substantial hardware replication to achieve the required degree of fault tolerance. Unfortunately, they do not exploit the distributed system's residual resources to improve reliability.

DIANA [135] and SCARLETT [134] projects proposed reconfigurable-IMA for predictable fault-handling in avionics systems. These approaches consider single-core processors. However, the introduction of multicore processors opens up more avenues to reconfigure the system while ensuring safety. For example, a complete multicore node is not faulty when one core encounters a fault. The system can still allocate other non-faulty cores to safety-critical applications. Moreover, as seen in projects such as DREAMS [10], the avionics domain is also exploring the use of heterogeneous nodes in mixed-criticality

avionics networks. Previous approaches tightly couple global management of the entire network with node-level resource allocation and are not suitable for adaptability in such heterogeneous systems.

Newer resource management techniques must consider heterogeneous nodes and leverage multicore architectures for further improving reliability while ensuring the current safety levels. In addition, resource management must provide low overhead reconfiguration strategies. Such strategies will help to improve system performance further and avoid unscheduled maintenance and associated costs.

II.25.3 Simultaneous Execution of Mixed-Criticality (Avionics) Applications

Avionics industries aim to co-execute applications of different criticality levels. Hence, critical applications such as a FMS can execute simultaneously on the same node as non-critical applications such as a IFE. Similarly, a real-time and a best-effort application can co-execute on the same node. However, safety-criticality applications must adhere to stringent industry standards such as [33, 43]. Thus, to co-execute applications of mixed-criticality on a single node requires strict temporal and spatial isolation. Achieving this isolation is not that straightforward in multicore processors or MPSoCs, primarily because of the numerous shared resources. If applications of different criticality levels are allowed to access the shared resources randomly, then a lower criticality application accessing a shared resource can block the accesses of any simultaneously executing high criticality application and severely affect the response time of the high criticality applications. Thus, it is difficult to quantify the impact on response time because of the limited knowledge about the behavior of low criticality applications in the system. Moreover, some hardware resources in COTS platform, such as on-chip interconnect arbitration, are available to the system designer (see Section II.14).

In MCS, there are conflicting goals for critical and non-critical (or best-effort) applications: the pessimistic WCET estimations of safety-critical (real-time) applications under-utilize the resources significantly in the average case. On the contrary, the non-critical or best-effort applications require efficient resource utilization to provide the best possible QoS. It further adds complexity to the existing resource management problem. Industrial concepts and standards, such as ARINC 653 for Integrated Modular Avionics (IMA) [45] have recognized these issues.

As an initial step in avionics, safety-critical single-core avionics applications were ported to a multicore processor by preserving the original schedule as well as the source code and executing them on only one core. Historically, the avionics domain uses cyclic scheduling. Some recent approaches such as Burns et al.[153] propose to extend cyclic scheduling to multicore. The authors propose to coordinate schedules on all cores so that all cores release the MiF at the same time and allow only applications of the same criticality to execute concurrently. However, these approaches do not allow executing applications of different criticality levels at the same time.

II.25.4 Safety-Critical Operation as a Cloud-Based Service

As seen in recent projects such as SECREDAS[154], a new trend of deploying real-time applications on cloud computing platforms is emerging in safety-critical domains. Hosting applications in the cloud support ease of re-usability, reconfiguration, and scalability while providing high availability, less need for maintenance, and reduced running costs. Moreover, cloud computing helps organizations to reduce their carbon footprint by letting them reduce resource over-provisioning. The cloud abstracts the underlying hardware from the applications. This property of cloud computing helps to deal with hardware obsolescence issues and makes possible the reuse of existing applications without modifications .

As explained in Section II.18, the cloud hosts applications as VMs. Multiple VMs run on each node in a cloud and share the underlying node resources. Cloud virtualization environments, such as Xen or KVM, allow partitioning off some of these resources to each VM. However, in a cloud with several nodes, each with several resources, partitioning some resources per node is insufficient to ensure predictability for real-time applications. It is tough to obtain realistic assumptions because the applications, availability of resources, and system configurations can keep changing in a cloud. Moreover, cloud hypervisors only consider a single node to make local resource allocation decisions and manage tasks locally without a global view of the system. Thus, no resource allocation guarantees are given to VMs in clouds. As a result, it is not possible to ensure (timing) predictability of VMs in clouds. However, the safety-critical domains need predictability achieved by careful management and allocation of resources to safety-critical VMs. These VMs must adhere to stringent safety standards. Thus, running such safety-critical applications on existing clouds is difficult.

Safety-Critical Railway Operation as a Cloud-Based Service

The railway domain has high reliability and availability requirements given by the CENELEC standards (e.g., [155, 156, 157]) used to certify safety-critical railway applications. The long lifespan of such applications of more than 25 years needs to be considered, as any change to the system should not compromise the safety requirements. Railway operation is a layered and complex business. Indoor components (e.g., Radio Block Centers (RBCs)) can control several interlocking stations parallelly to operate large railway networks. Also, RBCs are responsible for the radio communication towards European Train Control System (ETCS)-L2 operated trains and have widely increased in number over the last decade due to the roll-out of high-speed lines. Thus, the demand for a more scalable architecture has risen in the railway domain. Such architecture should follow the cloud computing principles while still ensuring the highest safety and security levels. Additionally, hosting applications in the cloud will support ease of re-usability reconfiguration while providing higher availability, less need for maintenance, and reduced running costs. Furthermore, the virtualization layer of the cloud abstracts the underlying hardware from the applications. This property helps to deal with hardware obsolescence issues and makes the reuse of existing applications without modifications possible. Virtualization can also ensure the required long application lifetime through

continuous operation during updates and permit mixed-criticality applications in the cloud. Therefore, the choice of suitable virtualization and cloud technology is essential for ensuring the performance, predictability, availability, and safety of such a system. An initial outlook on virtualization for safety-critical railway systems has been conducted in [158], focusing on a contract-based safety approach for the possibility of mixed-criticality as well as initial experiments on the real-time capability of different hypervisors. In [159], the author predicts future developments in the railway industry without presenting many technical details. He advocates the coded monoprocessor principle [160] to create a cloud-based railway control system, highlighting the advantages of increased availability, scalability, and performance with such an approach.

II.25.5 Open-world Assumptions: Safety- and Security-Aware Approaches

In the past, RTS and MCS were mainly concerned by safety and reliability requirements as they were physically isolated and ran on dedicated hardware platforms. Nowadays, many of these systems are networked and are moving towards an open-world assumption. For example, the trend of Internet of Things (IoT) is coming to automotive systems; passenger comfort and infotainment features continue to progress through the advancement of in-vehicle networks and connectivity of the automotive system with its environment. As a result, these systems can no longer be bounded by static system structures but need to consider components entering and leaving the system at runtime. The challenge here is not only to consider distributed and networked RTS and MCS but also to allow dynamic system structures and open-world assumptions without compromising the safety and reliability requirements of the systems. In addition, these systems have become a promising target for active and passive attackers. In the worst-case, compromising a few components can bring down the entire system. Therefore, an important aspect is ensuring the continuous unmaintained real-time operation of the system while guaranteeing safety and reliability requirements.

From a safety and security viewpoint, the resource management mechanism is a crucial point of the system and a vulnerable target for attackers. The system can completely lose resource allocation abilities even if single faults occur in the resource management. In the worst-case, faults in the resource management can lead to incorrect resource allocations directly impacting the execution of real-time applications leading to system failure. Attackers can obtain sensitive system information from the resource management and the communication among its components. Attackers can masquerade as a resource management entity and manipulate the system by making wrong resource orchestration decisions or provide incorrect information to other components. An attacker can cause system-wide failures by manipulating key components of resource management. Thus, the resource management mechanism must itself be both safe and secure. Existing global resource management frameworks for RTS and MCS do not consider both safety and security.

II.26 Relevant Resource Management Frameworks for RTS and MCS

This section presents existing resource management frameworks for RTS and MCS relevant to this dissertation.

II.26.1 Matrix Resource Management Framework

The Matrix framework [21] proposes real-time resource management to efficiently transport decoupled video streams with acceptable playout quality in dynamic environments such as home networks with limited availability of resources in heterogeneous nodes. The Matrix framework decouples the actual node-level scheduling from the system-wide resource allocation for efficient information processing at appropriate levels. The framework consists of a *Resource Manager (RM)* that makes system-side resource allocation decisions and node-level *Order Managers (OMs)* that perform the local monitoring and scheduling of CPU and network resources.

The RM maintains a *status matrix* with a system-wide view of resources based on updates from node-level OMs. It makes system-wide decisions based on the information in the status matrix. However, to keep the network and CPU overhead low, the OMs updates the status matrix with the minimum relevant information about states of nodes as needed by the RM instead of maintaining a fine-grained, accurate, and fresh view of the system state. The framework accomplishes this by using only a few discrete (abstract) Service Levels (SLs), such as HIGH, MEDIUM, and LOW, to indicate the availability of resources and QoS achieved by applications. As a result, the framework reduces the overheads in the system state determination and dissemination and abstracts fluctuations, which could overload the scheduling of resources on individual nodes.

Each time a new connection is required, the RM checks the status matrix to determine if the availability of resources is sufficient to satisfy the desired QoS of the new connection without violating the QoS of existing ones. If the RM finds enough resource availability in the status matrix to support the requested connection, it adds orders for resource reservation into an *order matrix*. If a variation in the availability of resources affects an active stream, the RM makes adjustments to QoS streams and allocates resources in the order matrix. Finally, the OMs maps the resource reservation constraints (orders) by the RM (placed in the order matrix) to the concrete scheduling specification of the local schedulers on their nodes. The communication among the OMs and the RM in the Matrix framework follows the publish/subscribe paradigm.

II.26.2 ACTORS Project Resource Management Framework

The ACTORS project [22] provided a generic framework for adaptability within a single multicore device. The project managed applications by automatically assigning them to *Virtual Processors (VPs)*. VPs isolated applications from other applications and abstracted the underlying physical hardware resources. A central *Resource Manager (RM)* allocated resources among VPs.

ACTORS project introduced the concept of *Service Levels (SLs)* for applications. Each application can have multiple SLs, and each SL corresponds to a specific resource requirement for an application and the QoS achievable by those resource requirements. Moreover, each application has an *importance* parameter that a system designer can use to indicate the more favorable applications. Finally, each application has a *happiness* parameter to indicate to the RM if it achieved the QoS associated with the SL.

At runtime, the RM monitors the applications in the VPs and dynamically adjusts the parameters of the VPs (including SLs of the applications) according to the monitored resource consumption and taking into account happiness and importance parameters. If a new application enters the system, it must inform the RM about its SLs, and the RM triggers a reallocation of resources and corresponding reassignment of SLs.

The ACTORS project considered both the CPU bandwidth requirements and the worst-case delay, after which the application requires CPU allocation again. The project proposed using abstract service levels of CPU availability and application demands to reduce runtime resource management overheads. The main application considered in the project was an adaptive real-time MPEG-4 video decoding and streaming application.

II.26.3 Real-Time Adaptive Resource Management (RTARM)

RTARM [24] is a framework for adaptive resource management in real-time distributed systems with heterogeneous COTS nodes. RTARM consists of Hierarchical *Service Managers (SMs)* to manage individual resources and complete nodes. At the bottom of the hierarchy, *Lower-level Service Managers (LSMs)* manage individual resources, such as CPU and network, and controls access from applications to the resources. At the top of the hierarchy, *High-level Service Managers (HSMs)* build services based on the LSMs. In addition, HSMs coordinate with each other and negotiate end-to-end resource allocations to provide guaranteed QoS to applications. Thus, the hierarchy allows complex QoSs representations on top of basic services while simplifying application design and facilitating consistent resource management across all applications in the system.

An application requests an acceptable QoS range for each quality parameter to an HSM. Firstly, the HSM translates this request into individual QoSs requests for its services and LSMs. Then, it propagates the translated requests to its LSMs and receives possible QoSs values between in the acceptable range that the LSMs can provide (if any). Next, the HSM performs a reverse translation and adds the QoSs values from LSMs into its own QoS representation. Finally, it checks the QoSs representation to ascertain if the status of the service reservations meets the requirements and accordingly commits or aborts the transaction. If there are insufficient resources to accept requests from high criticality applications, then the HSM adapts the low criticality applications to the minimum QoSs values and uses the newly released resources for the high criticality application.

Each service manager consists of the following essential sub-components: 1) a *negotiator* to broker admission control and delegate responsibilities to other components, 2) a *translator* to translate QoSs requests and replies to and from LSMs, 3) a *scheduler* to

determine the feasibility of allocation of resources, 4) an *allocator* to allocate and release resources, an adaptor to adapt QoSs of applications when required, 5) an *enactor* to enforce the changes in application QoSs or status, 6) a *monitor* for monitoring the QoSs of applications, and 7) a *detector* for detecting conditions such as overloads and resource underutilization.

II.26.4 Resource Allocation and Control Engine (RACE) Framework

RACE [161] is an adaptive resource management framework for real-time systems to allocate resources efficiently and adapt the system to changes in resource availabilities and demands. RACE prevents the need to redevelop or modify the framework for new resource management strategies by decoupling resource management algorithms from the middleware implementation and providing configuration options to support a wide range of algorithms.

RACE framework consists of the following main components: 1) *resource monitors* to monitor system-wide resources, 2) *QoS monitors* to monitor application QoS, 3) *resource allocators* to allocate resources to applications as per their requirements and current resource availabilities, 4) *configurators* that configure QoS parameters of applications, 5) *controllers* to make end-to-end adaptation decisions that ensure the system meets the QoS requirements of the applications, and 6) *effectors* that enact adaptation decisions of the controller.

In addition, RACE provides QoS specification models that hide the low-level platform-specific details from the system designers. This feature helps to shield system designers from the tedious and error-prone process of determining platform-specific QoS configurations for applications.

II.26.5 ACROSS Project Resource Management Framework

In the ACROSS project [23], a *Trusted Resource Manager (TRM)* and a *communication interface* provide the possibility to contain faults and reconfigure communication on an FPGA-based MPSoC with TT NoC. The communication interface is an element between the TT NoC and an MPSoC component. It acts as a guardian for the respective component and ensures the component can only send messages at predefined time points according to a TT schedule. A component cannot modify the TT schedule in a communication interface. Hence, a failure of a component cannot impact the timing or messages of other components.

The TRM configures all communication interfaces with the TT schedule via a specific channel on the NoC that directly provides access to register files and protected memories on communication interfaces, which are not visible to the component. Moreover, the TRM can dynamically reconfigure the communication interfaces to modify the communication schedule. The TRM accepts new communication schedule proposals from components and checks the validity of the schedule to ensure it is collision-free. The TRM rejects schedules where a collision may occur or violates the fault containment hypotheses. If the TRM accepts the schedule, it writes it transparently to the communication interface

of the corresponding component. A component with special access to the NoC hosts the TRM. Thus, if the TRM fails, it can take down the entire NoC and thus cause the failure of the MPSoC. Therefore, the fault model of the ACROSS tightly couples the TRM and the NoC.

II.26.6 Hierarchical Distributed Resource-management Architecture (HiDRA)

HiDRA [162] proposes a framework for simultaneous management of processors and network bandwidth by a control-theoretic approach to handle fluctuations in resource demands and availability. The resource management architecture is designed for a system consisting of multiple wireless sender nodes that act as data sources and transmit data to a receiver node. The receiver node acts as a data sink and performs post-processing on the data. The architecture aimed to guarantee the end-to-end delay between the time the source node produced the data to the time the receiver node produced an output.

There are three main components in HiDRA: 1) *Monitors* to update controllers with resource utilization periodically, 2) *Controller* to implement algorithms that compute adaptation decisions for each application and ensure the desired system resource utilization and 3) *Effectors* to modify the applications and achieve the adaptation suggested by the controllers.

To ensure that the applications meet end-to-end deadlines, the controller has a processor control loop to handle the processor utilization at the receiver node and bandwidth control loops located at the sender nodes to manage the bandwidth utilization. If the processor and network control loops are isolated from each other, then it is not possible to consider the coupling between network bandwidth and processor utilization required for assuring the application requirements. Thus, HiDRA arranged these control loops hierarchically with the processor control loop as an outer loop and the bandwidth control loops on each sender node as inner loops.

II.26.7 Real-Time Common Object Request Broker Architecture (RT-CORBA)

RT-CORBA [163] is a middleware solution specification for providing end-to-end QoS support in a fixed-priority real-time distributed system. TAO [164] is an example of a RT-CORBA specification implementation.

RT-CORBA specification provides the following interfaces and QoS policies for applications to manage and configure resources in the entire system: 1) priority mechanisms, thread pools, intra-process mutexes, and global scheduling service for CPU, 2) protocol properties and explicit bindings for network, and 3) buffering requests in queues and bounding the memory size of thread pools for memory.

RT-CORBA Object Request Brokers (ORBs) facilitate transparently handling communication requests between clients and servers. In addition, ORBs provide interfaces that allow applications to specify their QoS policy requirements.

A salient feature in RT-CORBA is an *OS-independent priority scheme* that allows the system designer to use global RT-CORBA priorities for application in a heterogeneous distributed system. These priorities help to relate the OS priorities of each node consis-

tently while ensuring that no priority inversion occurs in the network for communication requests between a client application and a server application. Each client application invokes a service request to the server via an ORB. The ORB maps the client's native OS priority to an RT-CORBA priority-level and transmits the RT-CORBA priority to the server as part of the message. ORBs along the path between the client and the server also map the client's CORBA priority to native OS priority and processes the request at this priority. This propagation of priorities across the distributed system results in a distributed priority inheritance and end-to-end predictability about the priorities of applications.

The *global scheduling service* is responsible for allocating system-wide resources to meet the QoS needs of the applications that share nodes. Applications can use this service to specify parameters, such as WCET or period. To support multi-threading, RT-CORBA specifies a thread pool model for servers designers to pre-allocate a group of threads with specific thread attributes, such as priority levels. Thread pools are helpful for ORBs and applications to leverage multi-threading while bounding the allocated memory resources.

RT-CORBA leverages policies and mechanisms of the underlying communication infrastructure that support resource guarantees. In addition, it provides interfaces for allowing applications to select and configure transport-layer protocols and properties. Moreover, the specification defines *priority bands* that contain any set of RT-CORBA priorities. Priority banded connections allow the system designer to separate traffic types by priority and dedicate separate connections between the client and the server for each priority band.

RT-CORBA 2.0 [163] supports a dynamic scheduling framework by defining distributable threads with one or more execution parameters, such as deadlines and importance. The distributable threads can span across multiple nodes and carry their scheduling parameters beyond node boundaries. On each node, the middleware maps the distributable thread onto a native OS thread. DynamicTAO [165] is an example of a RT-CORBA 2.0 specification implementation.

II.26.8 Game-Theoretic Resource Management Framework for Real-Time Applications

In the game-theoretic resource management framework [166, 167], a *central Resource Manager (RM)* makes resource allocation decisions for shared resources on a COTS multicore platform. Similar to the ACTORS project, the RM handles applications by assigning them to VPs. This framework also considers that applications can function at multiple SLs with different resource requirements. However, contrary to the ACTORS project, this framework considers that the RM is unaware of the resource requirements and SLs of applications due to intellectual property rights. Moreover, as the meaning of quality is dependant on individual applications, the framework accounts for the RM's inability to compare quality delivered by different types of applications. Thus, the framework decouples resource allocation decision-making from service level assignment and leaves the assignment of service levels to the application level.

The framework uses ideas inspired by the game theory to achieve decoupling. The RM provides each application with a "payoff" (resource allocation via VPs), and each application adjusts its SL internally to maximize the "revenue" (performance). Eventually, the interaction establishes an equilibrium between the SLs of applications and resources allocated by the RM. An advantage of such a decoupling is the reduction in resource management complexity, and thus, the involved overheads. However, the applications needed significant instrumentation.

II.26.9 Resource Manager for Guaranteeing End-to-End Deadlines over Dynamic Topologies

Millnert et al. [168] presented a theoretical framework from modeling dynamic topologies, such as IoT, Cloud, Edge, and Fog. They ensured the system always met deadlines by limiting the way nodes and applications leave or join the network.

This framework considers applications as *flows* of packets through a network of nodes. Flows are a set of interconnected services offered by nodes. A *Resource Manager (RM)* handles all the requests from nodes and flows to join or leave the network. When a flow sends a request to the RM for joining the network, the RM accepts or rejects the flow based on the proposed theorems. A flow is allowed to notify and immediately leave the network at any time. The RM can also force a flow to leave in certain conditions; for example, a node along the path of the flow requests to leave the system.

Contrary to flows, a node can notify and immediately join the network without further permission from the RM. However, when a node wants to leave the network, it must send a request to the RM. Upon receipt of such a request, the RM notifies the flows along the path to exit. Once the notified flows exit properly, the RM permits the node to exit the network.

II.26.10 Miscellaneous

There exist many other generic resource management approaches that are **not** suitable for use with RTS and MCS. For example:

- Hydra [17] is a multi-agent-based runtime distributed resource management framework designed to integrate runtime services in a distributed way. It provides resource reallocation if an application does not achieve the required QoS or resource failures occur.
- M-Hub [18] is a resource management framework for the internet of mobile things to assign Complex Event Processing tasks to different devices based on availability and characteristics.
- Xu et al. [169] proposed a hierarchical resource management system to allocate resources in virtualized data centers. The lower level of the hierarchy used fuzzy logic-based approaches to deal with varying resource availability and demands. The higher level of the hierarchy decided the resource allocation based on a profit model to maximize the data center profit.

- Tärneberg et al. [170] presented a set of resource management challenges for a mobile cloud network based on a model that captures cost and capacity-heterogeneity of the infrastructure. In addition, they proposed an algorithm for dynamic resource management that considers network link capacity, desired user latency, user mobility, data center resource utilization, and server over provisioning costs.

- The resource management framework proposed in [19] uses blockchain to maintain a record of available and spent resources in cloud nodes. Furthermore, the framework uses blockchain transactions to perform VM migrations to save energy in the cloud. The framework ensures a migration only occurs if enough CPU/GPU cores, DRAM and hard disk space, or I/O devices are available on the destination node (as per the record in the blockchain).

- Xu et al. [20] highlighted the potentials of the blockchain for resource management in future Sixth-Generation (6G) networks for scenarios such as IoT.

II.27 Contributions

II.27.1 Global Resource Management Framework for Real-Time and Mixed-Criticality Distributed System

We propose a domain-independent global resource management framework[2] for distributed MCS and RTS consisting of heterogeneous nodes based on multicore processors or MPSoCs. Our resource management framework combines the benefits of local and global resource management strategies and keeps the overheads low by decoupling global resource management from local resource management. It efficiently reallocates the resources and adapts the QoS or modes of applications upon fluctuations and changes in operating conditions. Furthermore, it supports the reallocation of resources at runtime upon the occurrence of resource failures. In addition, it provides 1) monitoring service to monitor the behavior of applications and availability or operational status of resources, 2) scheduling service to deterministically schedule access from applications to resources and ensure the application requirements are met, and 3) local and global reconfiguration services to allocate resources and adapt applications based on the current availability of resources and the operational conditions. Our resource management architecture allows multiple monitoring and scheduling techniques without tightly coupling them with the implementation of the framework. This design allows a system designer to select the appropriate monitoring and scheduling technique for each resource as per requirement without significant modifications to the framework implementation.

Our resource management architecture is scalable and manages a distributed system consisting of heterogeneous nodes with different operating speeds and locations in the system structure. It can be error-prone and tedious for a system designer to

[2] We initially developed this framework together with the French Aerospace Lab - ONERA and Thales Research & Technology during the DREAMS project and later extended it during SECREDAS and other internal projects.

correctly configure resource management according to each platform's low-level details in heterogeneous distributed systems. Therefore, our resource management framework allows the system designers to provide resource management configuration parameters abstractly and select Local Resource Monitor (MONs) and Local Resource Schedulers (LRSs) for each platform without the need to know or set fine-grained platform-specific configurations.

II.27.2 Exploitation of Multicore Processors for Fault-Tolerance (in Avionics)

We address fault-tolerance on distributed MCS and RTS with nodes consisting of multicore processors or MPSoCs via our global resource management framework. Our framework is capable of detecting core failures at runtime and reconfigure the system as per requirement. Our framework detects core failure by running a newly developed software monitoring service running on each core. Based on the current availability and demand of resources, the global resource management framework reconfigures the system partly or entirely upon core failures to ensure that all the critical applications remain active.

In the avionics use case, we use a strategy based on offline defined reconfiguration graphs (generated by ONERA's GREC tool[171]) for our global resource management framework to reconfigure the system upon core failures.

II.27.3 Simultaneous Execution of Mixed-Criticality (Avionics) Applications

We address the simultaneous execution of mixed-criticality applications via our framework. Our framework includes a regulation solution inspired by the runtime WCET controller [172]. The framework enables the concurrent execution of critical and non-critical applications on a multicore node. It ensures the deadlines of critical applications are met while improving the QoS of best-effort applications and overall resource utilization of a multicore node. For this purpose, our framework continuously monitors the progress of the critical applications. If a potential deadline overrun is detected, it stops the concurrently executing non-critical applications until the end of the critical application's time window.

II.27.4 Safety-Critical Operation as a Cloud-Based Service

To enable real-time industries to use cloud computing and enter a new market segment, *safety-critical operation as a cloud-based service*, we extend the global resource management framework to make it possible to develop a RT-Cloud for hosting RTS and MCS. We also present an implementation for a low latency memory bandwidth aware Time-Triggered (TT) scheduler for a cloud node running the KVM hypervisor. The scheduler ensures that the safety-critical VMs are provided CPU execution time and memory bandwidth precisely as per their requirement. Furthermore, we can use this Time-Triggered (TT) scheduler with our global resource management framework.

II.27.5 Railway Operation as a Cloud-Based Service

We explored virtualization technologies and cloud computing for migrating an existing real-time safety-critical railway use case from dedicated hardware solutions. We examined existing virtualization technologies for deploying a (private) RT-Cloud on COTS server hardware to run an existing railway use-case while meeting stringent safety and security requirements. Based on the examination, we provide an insight into using existing virtualization technologies with our global resource framework architecture to safely and securely execute the railway use case applications.

II.27.6 Safety- and Security-Aware Global Resource Management Framework

We consider safety and security for our global resource management framework itself. In addition to using redundancy and periodic heartbeat signals for fault tolerance in the framework, we propose a solution for secure communication among the resource management components. This solution provides multiple levels of security and various types of security algorithms that a system designer can select depending on the use case's requirements. Further, we propose distributing the global resource management decision-making among a minimum number of different hardware platforms. We propose implementing this solution using the Sawtooth blockchain platform to achieve Byzantine fault-tolerance for the global resource management decisions and security for the communication among the resource management components. We also experimentally evaluate the secure resource management communication (in an avionics use case) as well as the blockchain-based distributed global resource management (in a cloud-based use case).

III

Resource Management Framework

"Think globally, act locally."

– René Dubos

This chapter presents our resource management framework for distributed MCS and RTS initially developed with the French Aerospace Lab - ONERA and Thales Research & Technology during the DREAMS project and later extended during the SECREDAS and other internal projects. Our framework is inspired by the Matrix [21] and ACTORS [22] resource management frameworks. We aim to manage RTS or MCS consisting of heterogeneous nodes with multicore processors and MPSoCs.

III.1 System Structure

The system structure (Figure III.F1) consists of a set of clusters $\{C^0, C^1, \ldots, C^c\}$. Each cluster C^κ, has heterogeneous nodes $\{N_0^\kappa, N_1^\kappa, \ldots, N_{n_\kappa}^\kappa\}$, connected by an off-chip network with a topology such as a star, bus, or ring. In addition, intercluster gateways act as the connection between different clusters.

Each node is a multicore processor or an MPSoC containing multiple tiles connected by an on-chip network such as a shared bus, crossbar, or NoC. If the on-chip network is an NoC, each tile will have an Network Interface (NI) to communicate via the NoC. A tile can be simply a single CPU core, a complex cluster of CPU cores with private caches, or IP cores. Additionally, tiles can also be resources, such as memory controllers or I/Os, that are shared by several other tiles of the node. Finally, at least one tile is a gateway that connects the off-chip network with the on-chip network.

III.2 Resource Management Framework

We took into consideration various requirements and challenges for the design of our resource management framework. This section presents these requirements and challenges and explains how we solve them in our framework.

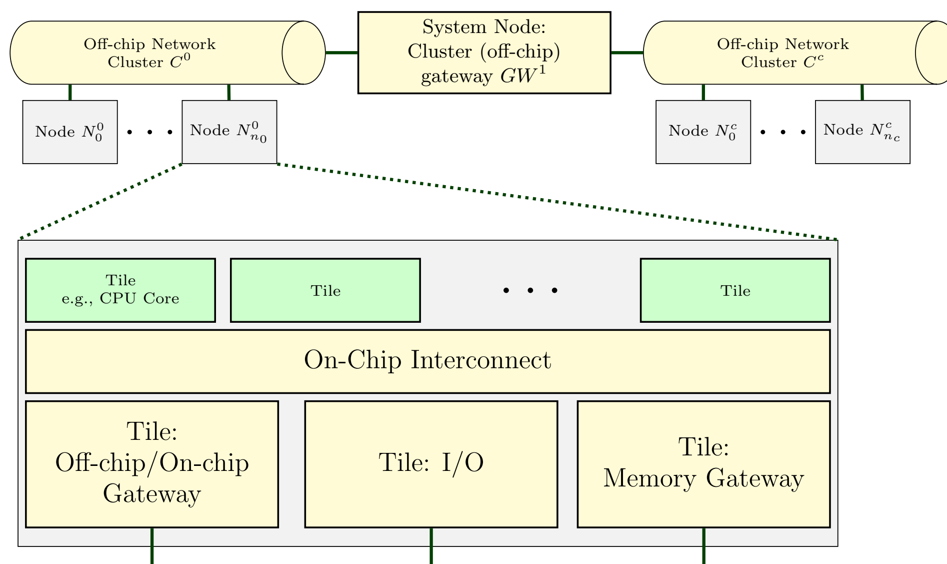


Figure III.F1: *System Structure*

III.3 Low Overhead Resource Management

Many existing methods perform resource management either at the node-level (e.g., [22], [23], and [166]) or at the system-level (e.g., [162], [161], and [168]). In node-level (local) strategies, the resource management performs local resource allocation and adaptation based on the local availability and demand of resources. The main advantage is that resource management can use domain-and node-specific information for making resource allocation decisions. However, these decisions are limited by the absence of a system-wide view of resources and applications. For example, each node can only execute applications in a timely manner if there are enough available resources on that node. If one node is overloaded, the system cannot benefit from the availability of resources in another node to execute the applications. Thus, these approaches are unsuitable for a complex system structure such as the one we consider in this dissertation.

In the system-level (global) strategies, resource management has a global view of applications and the availability of resources. Therefore, it can make global resource allocation and adaptation decisions to ensure the system meets the applications' constraints. Moreover, it can leverage global resources to improve the overall system efficiency. However, such approaches often have high overheads as nodes frequently communicate the demand and availability of resources to the resource management. This problem worsens if the system has highly fluctuating demand and availability of resources; the nodes not only need to communicate more often with the resource management due to the fluctuations but also the resource management must make more decisions and communicate all these decisions back to the nodes. Thus, such approaches are unsuitable for adaptability in the proposed system structure due to high overheads.

Solution Our resource management architecture combines the benefits of local and global resource management strategies and keeps the overheads low by decoupling global resource management from local resource management. The architecture consists of a ***Global Resource Manager (GRM)*** in combination with a set of a ***Local Resource Manager (LRM)*** as shown in Figure III.F2.

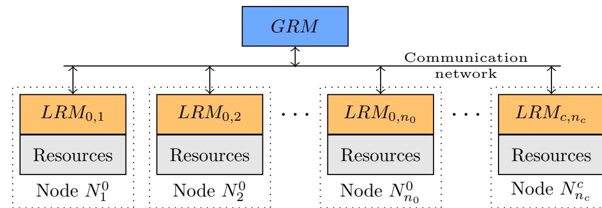


Figure III.F2: *Basic Architecture of the Resource Management Framework*

- The GRM handles long-term changes in the demand of applications and the availability of resources. Moreover, it manages the system upon resource failures that the LRMs cannot manage. Conceptually one GRM exists in the system, although distribution is possible for scalability and fault-tolerance (as explained in Chapter V). For simplicity, we assume the GRM as a single entity in this chapter. The GRM gathers resource and application *updates* from the LRMs, and provides new resource allocation *orders* to the LRMs, if necessary. The GRM orders can include different pre-computed configurations for resources (e.g., time-triggered schedules) or parameter ranges (e.g., resource budgets). Alternatively, the GRM can dynamically compute new configurations.

- The LRMs takes care of temporal overloads and short-term fluctuations in demands of application and availability of resources. The resources can be physical resources such as a single processor core, core clusters, memory (space and bandwidth), caches, I/O devices, hardware accelerators, and network bandwidth. The resources can also be virtualized. Moreover, they can accommodate the failure of resources up to a certain extent. In addition, the LRMs send *updates* to the GRM or request global adaptation by the GRM upon significant changes in demands of applications and availability of resources (as explained in Chapter IV).

III.4 Adapting to Fluctuations in Availability and Demand of Resources

Real-time applications can miss their deadlines if their resource requirements are not met (on time). Moreover, the pessimistic WCET estimations of safety-critical applications under-utilize the resources significantly in the average case. On the contrary, the non-critical or best-effort applications require efficient resource utilization to provide the best possible QoS. Hence, significant under-utilization of the system is unacceptable as

it reduces overall efficiency and increases operational costs. Thus, the availability and demand of resources, such as CPU, memory bandwidth, cache, and network bandwidth, may fluctuate over time and vary significantly when changes in operating conditions occur.

Solution Our resource management efficiently reallocates the resources and adapts the QoS or modes of applications upon fluctuations and changes in operating conditions. Thus, our resource management architecture the following services:

- monitoring service to monitor the behavior of applications and availability of resources,
- scheduling service to deterministically schedule access from applications to system resources and ensure the application requirements are met, and
- local and global reconfiguration services to allocate resources and adapt applications based on the current availability of resources and the operational conditions.

III.4.1 Adapting to Failures in System Resources

Failure in individual system resources, such as a CPU core or network, or complete failure of nodes can occur in a system due to changing environmental conditions or phenomena, such as wear-out and infant mortality [173]. RTS or MCS need reallocation of resources at run-time upon the occurrence of resource failures with minimal (or no) interruption of safety-critical applications.

Solution Our resource management provides the following special services:

1. monitoring operational status of resources,
2. prioritization service to prioritize applications (as per their safety-assurance levels), and
3. local and global failure reconfiguration service to reallocate resources locally or globally under reduced resource availability.

Chapter IV and V explain local and global reconfiguration services with more details.

III.5 Modular Monitoring and Scheduling of Resources

As explained in Section II.15, a variety of resource allocation approaches exist to deal with contention in shared resources of a node. However, most of these approaches do not address all the shared resources of a node and aim to resolve contention in only a specific shared resource or a fixed set of shared resources. None of these algorithms are suitable for all cases or perform better than all other algorithms, and each algorithm has different associated overheads and limitations. Besides, different strategies may be

preferable depending on the application domain and applicable standards. However, many existing approaches (Section II.26) tightly couple global resource management with local resource allocation and require a fixed set of schedulers. Similar to scheduling techniques, different monitoring solutions exist to monitor the availability of system resources and the behavior of applications. However, none of the existing solutions are suitable for all cases or perform better than all other algorithms. Each solution also has different associated overheads and limitations.

Solution Our resource management architecture must allow multiple monitoring and scheduling techniques without tightly coupling them with the implementation of the framework. Such a design will allow a system designer to select the appropriate monitoring and scheduling technique for each resource as per requirement without any significant modifications to the framework implementation. Thus, our resource management architecture has modular LRM design with two types of pluggable modules:

1. ***MON***: This type of modules monitor resources and applications, and

2. ***LRS***: This type of modules schedule and control access of applications to resources.

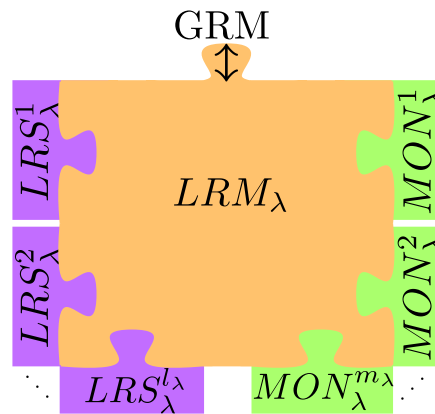


Figure III.F3: *Modular LRM design*

As shown in Figure III.F3, each LRM_λ provides interfaces to connect multiple MON modules, $\{MON^0_\lambda, MON^1_\lambda, \ldots, MON^{m_\lambda}_\lambda\}$, and multiple LRS modules, $\{LRS^0_\lambda, LRS^1_\lambda, \ldots, LRS^{m_\lambda}_\lambda\}$. Thus, a system designer can connect different types and number of MON and LRS into the LRMs as per requirement. For simplicity, henceforth we will refer to MON modules and LRS modules as MONs and LRSs. The next chapter explains them in more detail.

III.6 Tackling Heterogeneous Nodes, Complex System

Structures, and Scalability

In the basic architecture shown in Section III.3, the GRM is at the top of the hierarchy with a complete view of the entire system and directly supervises and controls all the LRMs. All LRMs stand at the same level in the hierarchy. Together with MONs and LRSs, each LRM manages a node. It directly communicates with the GRM. We refer to this architecture as flat resource management architecture. The main advantage of flat architecture is the simplicity of the design. Figure III.F4a shows an example of the flat architecture.

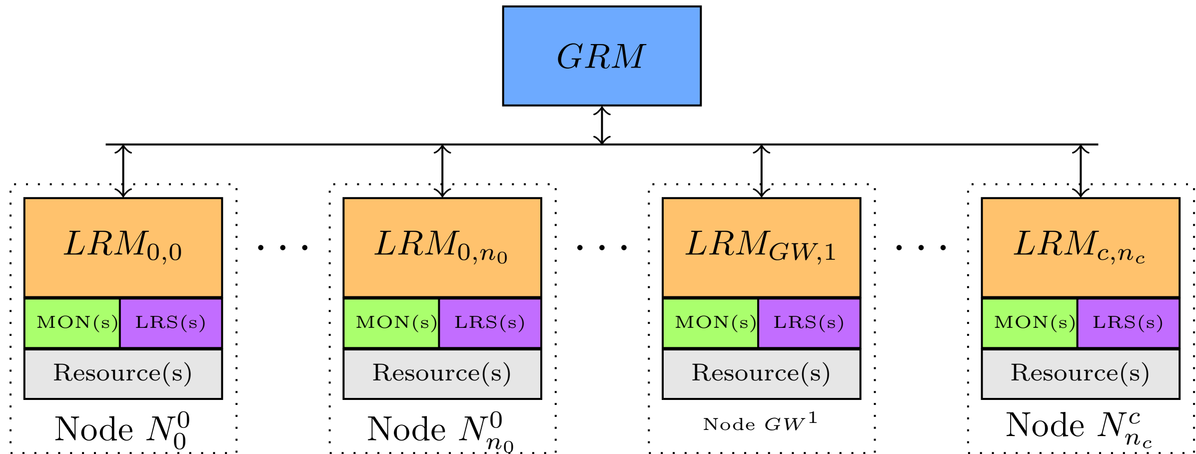
The considered system structure consists of heterogeneous nodes, each with a diverse set of resources, and nodes can communicate on different networks and realize their activities at considerably different speeds. The flat architecture disregards the location or operational speed of the nodes or resource(s). Thus, it cannot cope with granularity issues, especially from a timing perspective. Different resources and nodes realize their activities at considerably different speeds. When all LRMs are treated equally by the GRM, it is not possible to consider that fact. Furthermore, there can be conditions that require a reconfiguration of only a subset of resources, e.g., all resources inside a single node. In a flat architecture, such conditions and the subsequent reconfigurations are only addressable by a resource management component with a system-wide view: the GRM. Finally, in a system with many nodes, the updates and reconfiguration requests from LRMs can overwhelm the GRM in a flat architecture. As a result, the flat architecture has limited scalability.

Solution An alternate to flat architecture is a hierarchical resource management architecture where the GRM sits at the top of the hierarchy and LRMs are present at different levels in the hierarchy. Figure III.F4b shows an example of the hierarchical architecture. The GRM directly communicates with the LRMs at the second-highest level of the hierarchy. In turn, these LRMs communicate with the LRMs below them. Each LRM communicating to another LRM or set of LRMs introduces a new level in the architecture. This structure allows the LRMs to act as a granularity interface and hides fine-grained activities of a sub-system from the GRM's view. As a result, the GRM receives a limited number of resource updates. LRMs send reconfiguration requests to the GRM only when a reconfiguration of the entire system is necessary. The final result is a scalable resource management architecture that manages a distributed system consisting of heterogeneous nodes with different operating speeds and locations in the system structure.

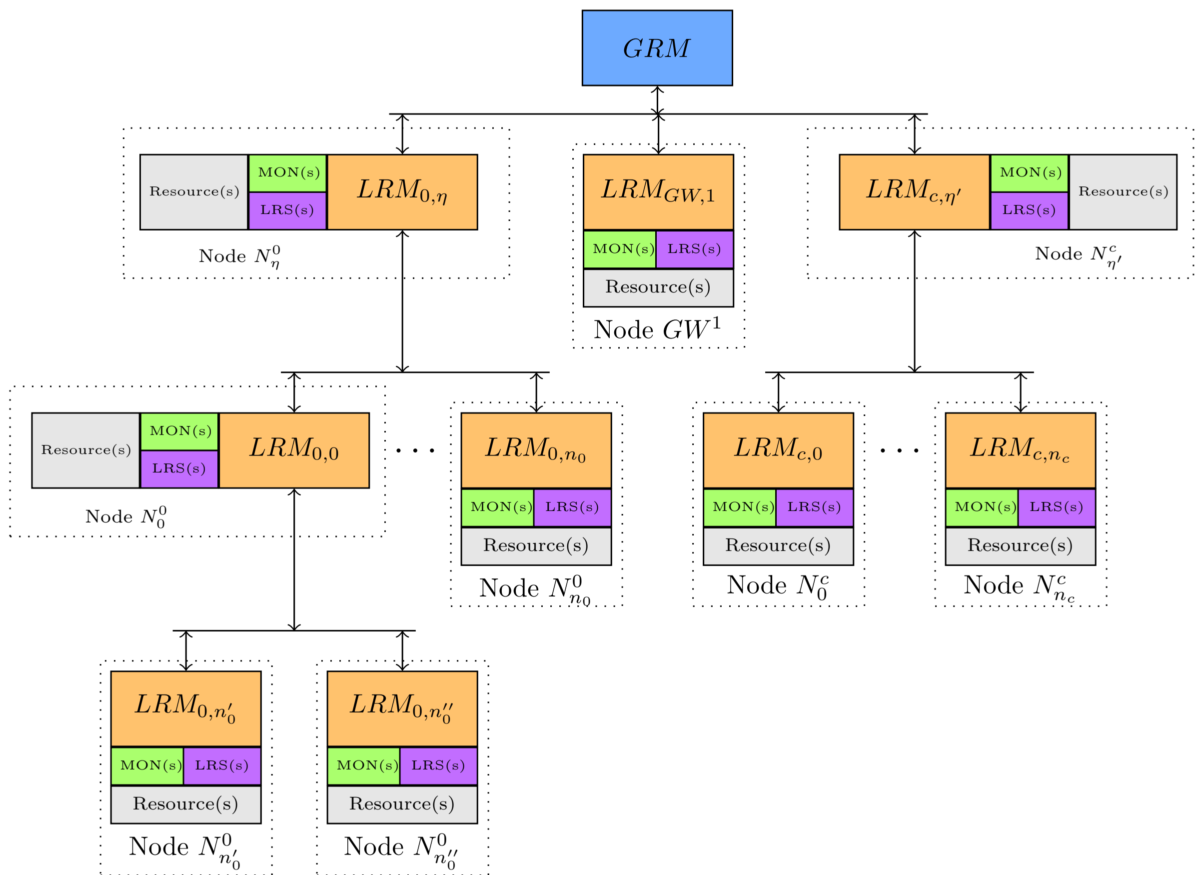
III.6.1 Effective System reconfiguration

The resource management architecture must be flexible enough to accommodate heterogeneous nodes and resources while providing the essential services for reconfiguring the system:

- System-level reconfiguration is required for reallocating resources (e.g., cluster



(a) *Flat*



(b) *Hierarchical*

Figure III.F4: *Examples of Flat and Hierarchical Architectures*

and global network) to applications in case of long-term changes in availability (including failures) or demand for resources. Applications can be entirely removed from the system or redeployed to another cluster.

- Cluster-level reconfiguration is required for reallocating cluster resources (e.g., node and cluster network) to applications within the cluster in case of changes or fluctuations in availability (including failures) or demand for resources. The reconfiguration can redeploy applications (or VMs) on a different node in the cluster.
- Node-level reconfiguration is needed for reallocating node resources (e.g., CPU cores, memory, and cache) to the node's applications in case of changes or fluctuations in availability (including failures) or demand for resources. If the node employees virtualization, then node-level adaptation can change the mapping of physical resources to virtual resources to meet the demand and availability of resources.
- Suppose a virtualization layer is present on nodes. In that case, the virtualization-level reconfiguration is needed to reallocate virtual resources (e.g., virtual CPU (vCPU) and Virtual memory) to VMs in case of fluctuations in availability or demand for virtual resources. A failure of physical resources can lead to a reduction in the availability of virtual resources. Virtualization-level reconfiguration can reallocate virtual resources to VMs in such a case as well. Note that resource management treats the VMs similar to applications in the other (higher) levels.
- An application consists of several tasks. An application-level reconfiguration is required for reallocating (virtual) resources among tasks of an application. It can also remove tasks or throttle the QoS (or change the parameters) of one or more tasks as per requirement.

The resource management architecture must perform the reconfigurations at all levels in the system in a stable and coordinated manner. In a complex system structure such as the one we consider in Section III.1, the GRM will have large overheads to coordinate the reconfiguration at all levels. As a result, it may not perform reconfigurations at different levels in a timely manner. From a temporal perspective, local reconfiguration in a sub-system can be initiated by an LRM much sooner without waiting for communication with the GRM. Moreover, the LRM can mitigate issues temporarily while waiting for instructions by the GRM to implement a more permanent solution. In addition, a reconfiguration at a lower level must only impact a higher level when it cannot internally handle the change in demand/availability of resources or operational conditions. For example, the resource management architecture must perform cluster-level reconfiguration only when node-level reconfiguration is insufficient.

Solution We take advantage of the hierarchical resource management architecture to introduce the concept of the resource management domains for effective reconfiguration at all levels. We consider five different domains in the system structure to perform resource management: System Domain, Cluster Domain, Node Domain, Virtualization

Domain, and VM/application Domain. The domains represent the composition of the system from the resource management perspective. Conceptually, they also correspond to the architecture's hierarchy levels.

The GRM controls and coordinates the LRMs in the cluster domain. It corresponds to the highest level of the hierarchy of resource management components. In the other domains, an LRM is in charge of coordinating and controlling the resources and lower-domain LRMs. The resources can be controlled indirectly through communication with a lower-level LRM or directly by communication with MONs and LRSs of the individual resources of that domain. Figure III.F5 presents the composition of the system in terms of resource management domains. It is important to note that Figure III.F5 does not intend to show where each GRM or LRM is physically implemented or where they execute. Instead, the figure presents an abstract view of the resource management domains established via the hierarchical architecture. In general, all resource management building blocks can have hardware or software implementations, or a combination of both, depending on the domain and type of resource.

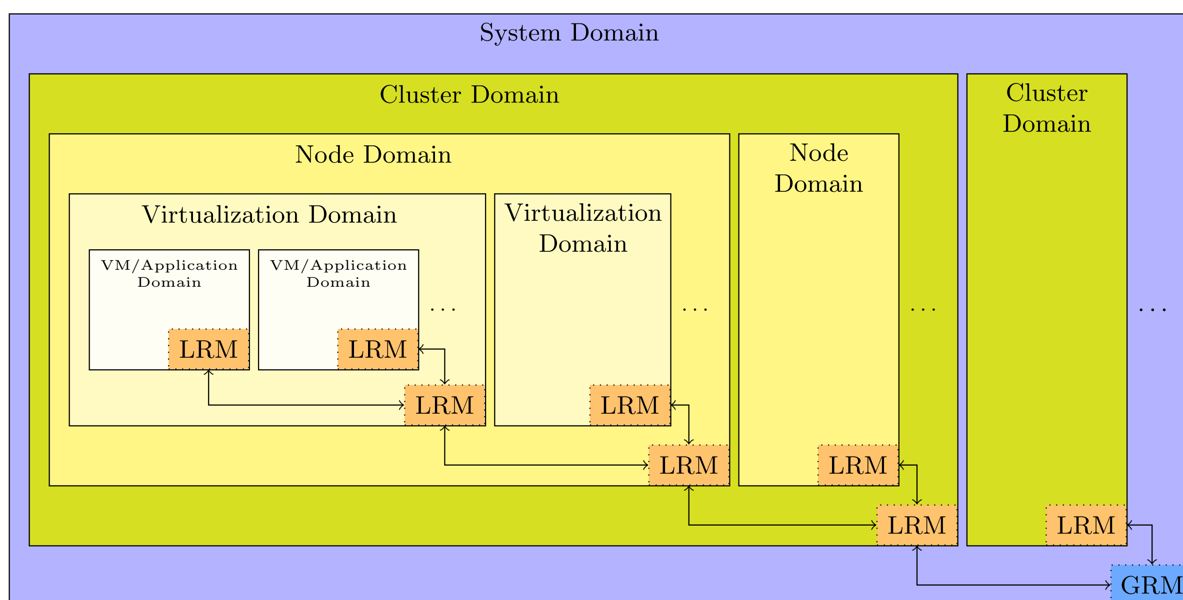


Figure III.F5: *Resource Management Domains*

It also depicts the LRM and LRMs in the system and the scope of their actions. We can differentiate between LRMs in four types:

1. Cluster domain LRM
2. Node domain LRM
3. Virtualization domain LRM
4. VM/Application domain LRM

A domain's LRM manages all change internal to the domain without influencing other domains. Any two LRMs of the same domain type cannot directly influence or communicate with each other. Suppose a domain's LRM cannot internally resolve an issue. In such a case, the LRM immediately performs the most urgent subsystem reconfiguration (e.g., prioritizing available resources to critical applications or switching to a safe state) to ensure the system's safety. Then it requests the next higher domain (LRM or GRM) for a reconfiguration with a broader scope to efficiently resolve the issue (a reconfiguration at a higher domain can be slower that the lower-domain reconfiguration as it needs to redeploy applications between sub-systems). As a result, the GRM only receives reconfiguration requests from the cluster domain LRMs.

The GRM does not need to perform the fine-grained adaptation in other (lower) domains. The GRM only performs reconfiguration at the system level, while the LRMs are responsible for adaptation in their respective domains based on order from the GRM. Similarly, for example, the cluster domain LRM is only responsible for reconfiguration at the cluster-level, while the node-level LRM is responsible for the fine-grained allocation of node-level resources, such as CPU cores, memory, and cache, based on the order from the cluster-domain LRM.

It should be noted that an instantiation of the resource management architecture can contain the system domain and multiple other (local) domains. However, an instantiation must not contain all the domains. For example, a minimal instantiation of the resource management architecture requires the system domain and only one of the other domains (flat architecture).

III.7 Dynamic addition or modification of applications

A system designer can initialize the RTS or MCS with a set of applications related to the system's primary objectives. The initial resource allocation is determined based on these objectives and related applications. However, many RTS and MCS exhibit multiple operation phases, each with different behavior and run different sets of applications. For example, in an avionics system, different applications execute during take-off, landing, and cruising. Moreover, the objective can change dynamically at run-time, and new objectives may be added. As a result, the resource requirement of existing applications may change, or reconfiguration may be required to allow the execution of new applications. Nevertheless, the system should achieve these modified or added objectives under the current availability of resources.

Solution The GRM manages an external input that can, in turn, trigger a global reconfiguration. Such input could be given locally (I/O peripheral directly connected to the GRM node) or remotely (via off-system Ethernet). A user can use this input to provide new application(s) or constraint(s) to the system. The GRM can determine a new configuration that satisfies the constraint or allocates the required resources to the application(s). The input can also be an absolute reconfiguration decision. The GRM directly communicates to the LRMs in the cluster domain, which in turn trickle down

the appropriate reconfiguration decisions to the lower levels (and so on). Similarly, the GRM can also trigger mode change in the system based on the external input.

III.8 Safety and Security for Resource Management

The resource management architecture is a crucial part of the system and a vulnerable target for attackers. Attackers can obtain sensitive system information from the resource management and the communication among its components. Attackers can masquerade as a resource management entity and manipulate the system by making wrong resource orchestration decisions or providing incorrect information to other components. An attacker can cause system-wide failures by manipulating key components of resource management. Moreover, the system can completely lose resource management abilities even if a single resource management component is faulty. In the worst-case, faults in resource management can lead to incorrect resource allocations, directly impacting the execution of real-time applications leading to system failure.

Solution We can use redundant LRMs for fault tolerance in local resource management domains. We provide an example of using redundant LRMs in the avionics use case of Chapter VI. Moreover, LRMs can send periodic heartbeats to the next higher resource manager (an LRM or the GRM) in the hierarchy, as explained in Chapter V.

This dissertation proposes two solutions for the system domain (global resource management). The first solution offers only security for the communication amongst a single central GRM and the LRMs by adding a security layer to the resource management architecture. This solution provides multiple levels of security and various types of security algorithms that a system designer can select depending on the use case's requirements. In addition, the solution has low overheads. Chapter V explains both these solutions in detail.

The second solution proposes distributing the GRM among a minimum number of different hardware platforms. The GRM makes the global resource management decisions via coordination amongst its distributed components. This solution can achieve Byzantine fault-tolerance for the global resource management decisions and security for the communication amongst the distributed GRM components and the LRMs. Chapter V explains both these solutions in detail.

III.9 Configuring Platform-Specific Resource Management Components

Meeting the application requirements depends on the configurations in the underlying hardware and software platforms, for example:

1. the configurations, such as the number of CPU cores, memory size, shared-interconnect bandwidth, and settings of QoS enabled components, of multicore processors or MPSoCs of the node,

2. configurations of hypervisors (or OSs) such as supported scheduling and allocation strategies for various resources and CPU models, and

3. network configurations such as QoS settings or traffic-classes.

These configurations are specific to a software platform or a hardware node. Therefore, system designers need to be aware of the detailed configurations of each platform and configure the resource management accordingly to deal with these platforms. However, in heterogeneous distributed systems, with different types of processors, networks, and underlying software, it can be error-prone and tedious for a system designer to correctly configure the resource management for all the low-level details of each platform. Therefore, the system designers must be able to specify these resource management configuration parameters abstractly. In addition, the system designer must have the ability to easily select MONs and LRSs for each platform (as discussed in Section III.5) without the need to know or set fine-grained platform-specific configurations.

Solution As shown in Figure III.F6, the resource management framework has access to a library containing the configurations of available hardware and software platforms, and a library with security protocols for resource management. In addition, it has access to MONs and LRSs libraries. These libraries can contain entire software implementations for monitoring or scheduling on nodes. Alternatively, they can provide interfaces for the resource management to configure/read/write existing platform-specific monitors (e.g., Hardware performance monitor counters and health monitors in hypervisors) or schedulers (e.g., QoS setting in QoS-enabled IP blocks or network interface, application priorities in OS and VM scheduling in the hypervisor). Moreover, the resource management framework can be interfaced with offline scheduling tools, if needed, to generate offline schedules for applications and resource management components. Similarly, the framework provides an interface for tools that can generate a configuration for the selected hypervisor platform (if any). The system designer must provide a high-level system configuration providing information such as types of nodes, the interconnection between the nodes, and name of the hypervisor or OS of the node. The system designer must not specify the detailed configuration of the individual nodes (and hypervisors/OS). Instead, the resource management framework gathers them directly from the platform configuration library. In addition, the system designer must provide parameters and configurations of applications to the resource management framework. Finally, the system designer must specify the resource management configuration containing information such as resources to consider, the names of the required MONs and LRSs per node (or in all system nodes), resource management domains and hierarchy, the location (node) of LRMs and GRM, and the safety and security protocols. Based on all the inputs and the platform configurations, the resource management framework selects the appropriate MONs and LRSs from the corresponding libraries. Then, it generates the platform-specific c-code for the GRM and the LRMs and selects the appropriate platform-specific compilers from the available ones to compile the generated code. The final result is resource management binary files that a system designer can deploy on the respective platforms.

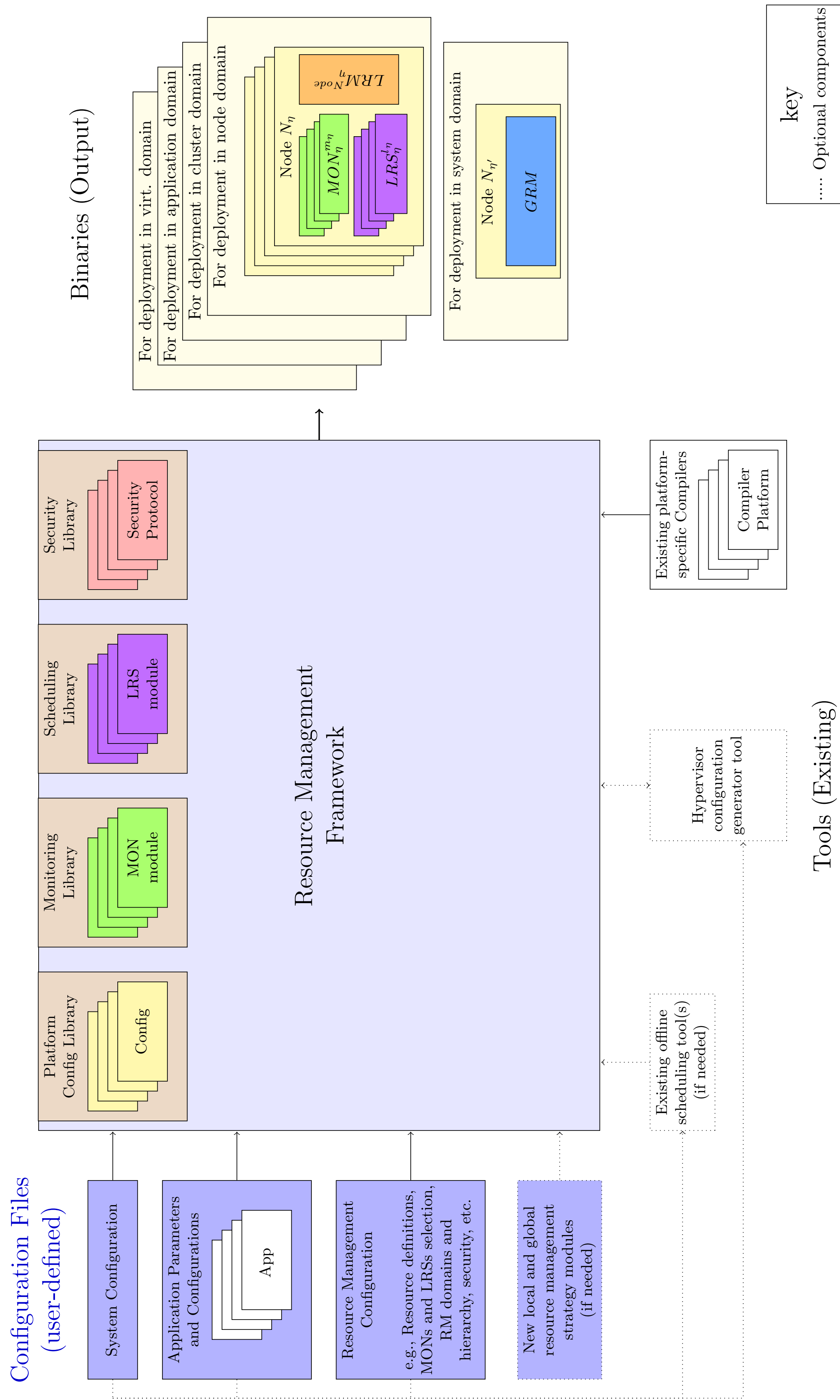


Figure III.F6: *Configuration of Resource Management Components*

Listing A.L1 and A.L2 of Appendix A show examples of a platform configuration (library) file and system configuration file (user-defined). Listing A.L3 and A.L4 of Appendix A show examples of resource management configuration file (user-defined) and application configuration file (user-defined) similar to those used in the avionics use case of Chapter VI.

The next chapter explains LRM, LRSs, and MONs in more detail, while Chapter V elaborates the GRM and the (secure) communication among LRMs and the GRM.

IV Local Resource Management

"We don't believe we've solved the multicore-programming problem. But we think we've built an environment in which a certain class of problems can take advantage of the multicore architecture."

– Rob Pike

This chapter elaborates on the LRM component of the resource management architecture. As explained in Chapter III, LRMs contain two main modular types of sub-components:

1. MONs provide *monitoring services*, such as monitoring availability and detecting errors. This chapter explains the monitoring services in more detail. In addition, it provides an overview of hardware and software-level monitoring opportunities, existing work on monitoring for RTS or MCS, and new MONs that we developed for RTS and MCS.

2. LRSs provide *scheduling services* and *configuration services*. This chapter explains both the services and presents the LRSs that we developed for different use cases.

Finally, this chapter explains the *local resource management services* provided by the LRMs and policies to manage permanent core failures and potential deadline overrun in critical applications while improving the QoS of best-effort applications and overall resource utilization of a multicore node.

IV.1 Monitoring Services

Each resource or application managed by the LRM has one or more MONs, each providing different monitoring service. MONs can provide the following type of generic monitoring services:

Availability Monitoring

Several distinct state parameters, such as the utilization or the operational status of a resource, can be associated with the availability of the resource. Some examples for MONs that provide services to monitor the availability of resources are:

- MONs for determining memory, network, or CPU utilization
- MONs for detecting the resource's operational status (such as CPU core failure)
- MONs performing diagnostics tests on resources can help to deduce the resource availability and status.

Behavior Monitoring

Monitoring services are required to monitor resources, hypervisors/OSes, or applications to obtain an insight into their behavior. Some examples of MONs with these services are:

- Resource MONs to find out the number of waiting messages in a queue or pending read/writes in the memory controller
- Hypervisor/OS MONs to determine the keep track of tasks/VMs in the ready queue or the number of context switches in a given period
- Application MONs to determine the QoS of applications or the impact of multicore contention on real-time tasks

Reliability Monitoring

There is a close link between errors and the reliability of the systems. Hence, RTS or MCS require MONs that provide monitoring services to detect:

- Errors in temporal domain, such as violations of task period, arrival pattern of messages, or (potential) deadline miss of by real-time task
- Errors in value domain, such as errors in the body of a message, errors in results of computation at the application level, or corrupt memory space.

The LRMs can also extract resource reliability from the results of diagnostics tests routinely performed in the system.

Energy Monitoring

Due to the limited battery size, it is essential to save energy in battery-operated RTS or MCS, such as portable media players or drones. Energy-saving is also essential in non-battery-operated RTS or MCS due to electric costs, cooling costs, and hot-spots that impact system reliability. LRMs can use MONs to determine the energy consumption of resources or applications and the battery level at runtime.

IV.2 Monitoring opportunities

There are numerous existing hardware and software monitoring solutions that we can use within our resource management framework. These opportunities are present inside COTS multicore processors and MPSoCs and at the OS/hypervisor-level. To take advantage of these opportunities for resource management, we need to encapsulate and interface them by designing simple platform-specific MONs for the monitoring library.

IV.2.1 Hardware Performance Monitoring Counters

In most multicore processors and MPSoCs, there are a few special-purpose registers built into the hardware to store the count of specific events, such as clock cycles and cache misses, happening inside the processor or the SoC. In this dissertation, we refer to them as individually as Performance Monitor Counters (PMCs). A set of PMCs packaged as a single unit called Performance Monitor Unit (PMU) are present inside CPU cores and some (or all) shared resources of a processor or a SoC.

Taking inspiration from the terminology found in technical reference manuals for Intel x86 processors, we classify the PMUs into two main types depending on their location in the processor or the SoC (Figure IV.F1):

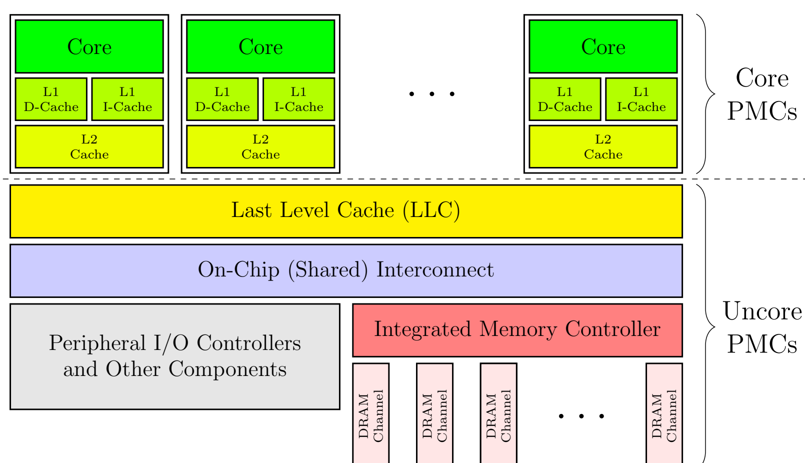


Figure IV.F1: *Classification of PMUs*

Uncore PMU

Uncore PMUs refers to PMCs residing in components and logic that are outside the CPU cores (and their private memory) but still on the same die. Uncore components and logic are usually shared resources. They include, among others, the cache coherency logic, shared interconnect, Last Level Cache (LLC), I/O controllers and devices, power control unit, and the Integrated Memory Controller (IMC). An example of such a PMU can be encountered in a IMC. The PMU in IMC can count events such as the number of DRAM refreshes and the number of DRAM commands (e.g., ACT, PRE, and CAS).

The PMCs of uncore PMUs provide a total event count from all cores. If multiple cores are simultaneously accessing the shared resource, it is not possible to attribute the value of a PMC in that shared resource to a single core. Thus, we can use uncore PMU for determining the overall utilization, operational status, and behavior of the corresponding shared resource itself.

In Section II.4, we provided examples of three hardware platforms: NXP T4240 [12], Xilinx Zynq 7000 [13], and Intel Xeon Gold 5218 [31]. Unfortunately, the information on uncore PMUs is not available publicly for the T4240. For Zynq 7000, Xilinx provides an uncore PMU IP block called the AXI Performance Monitor (APM) [174] that can be synthesized for the Zynq Programmable Logic (PL) (on-chip FPGA). This PMU can be used to monitor the transactions on the AXI bus (shared interconnect of the Zynq 7000). It should be noted that such a PMU cannot be used with other COTS platforms that do not have an FPGA on the same die. The information on other uncore PMUs (if available) is not documented. Lastly, the Xeon Gold 5218 provides uncore PMUs in various shared resources of the platform [175], for example:

1. Integrated Memory Controller (IMC)
2. Caching/Home Agent (CHA) (component managing the interface between core, LLC, on-chip interconnect and the I/O controller)
3. UltraPath Interconnect (UPI) Link Layer (on-chip interconnect)

Core PMU

Core PMUs are present inside CPU cores. They can count events related to the following:

1. The core: for example, clock cycles, mispredicted branches, and instructions retired/architecturally executed.
2. The private memory of a core (usually Translate Lookaside Buffer (TLB) and private caches): for example, L1-Data/Instruction cache hits/misses and TLB-refills.
3. Offcore: Offcore events refer to the activities of the core that are on the way to the uncore. Examples of such events are access to shared interconnect, access to LLC and L2-prefetch.

PMCs of core PMUs are beneficial to monitor the availability and behavior of a core and its private memory. In addition, they are helpful to monitor the behavior of a task running on the core. For example, we can develop a MON that uses a PMC of a core PMU to estimate the progress of a task by counting the clock cycles or instructions retired while that task is executing on the core (example in Section IV.4). Moreover, it is also possible to develop MONs that monitor the reliability of applications based on the event counted by the on core PMUs. We provide an example of such a MON in Section IV.4.

The three hardware platforms from Section II.4 contain Core-type PMUs in each CPU core. Each PMU in a CPU core contains a fixed number of PMCs. PMCs either

count a fixed event or can be programmed to count events from a list provided by the corresponding semiconductor manufacturer. Table IV.T1 provides the number of PMC on each CPU core for the three platforms.

Table IV.T1: *PMCs per Core PMU*

Platform	CPU core	Programmable PMCs (per core)	Fixed PMCs (per core)	Comments
T4240	e6500 [176] (dual threaded)	12	-	6 programmable PMCs per hardware thread
Zynq 7000	Cortex-A9 [177]	4	-	
Xeon 5218	Xeon Scalable 2nd Gen. [178] (dual threaded)	8	3	4 programmable PMCs per hardware thread. Fixed PMCs are common for both threads

PMU Events in Multicore Processors and MPSoCs

Appendix B provides examples of some interesting PMU events with respect to resource management on the three hardware platforms:

1. Table B.T1 in the appendix provides examples of PMU events for the e6500 cores [176] of NXP T4240 [12]

2. Table B.T2 and Table B.T3 in the appendix present examples of Cortex A9 core PMU events [177] and the AXI shared interconnect uncore PMU (APM IP block [174]) events in the Xilinx Zynq 7000 [13]

3. Table B.T4 and Table B.T5 in the appendix supply examples of Intel Xeon Gold 5218 core and uncore PMU events [179]

Discussion

Most modern processors and SoCs contain uncore PMUs. However, the information, such as their addresses, events, and protocols to set them up, are often not present in the public domain for many hardware platforms. In our experience, many semiconductor manufacturers are usually reluctant to provide this information. Thus, it is not possible to use uncore PMUs on such hardware platforms.

All hardware platforms support counting many different types of events. However, some hardware platforms implement very few PMCs in a PMU. For example, ARMv7a only implements four PMCs in a core PMU but supports upwards of fifty events [180]. As a result, ARMv7a cores can only count a maximum of four events together despite support for many more events.

We expect the PMCs to provide deterministic event counts. However, in reality, to keep the implementation and validation cost low, PMCs generally have a small degree of inaccuracy [181]. PMCs are known to show variation between runs of the same benchmark in strictly controlled experiments. Furthermore, they often overcount events in x86 architecture [182]. As these effects are counter-intuitive and unpredictable, it becomes difficult to use them for resource management strategies that require a precise event count.

IV.2.2 Intel CMT and MBM Monitoring Technologies

The Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM) hardware technologies present in the Intel Xeon Scalable processors (2nd Gen.) [30] provide the possibility to monitor the LLC occupancy and memory bandwidth usage of an application running on a CPU core (or a hardware thread). The resource manager, OS, or hypervisor can assign each scheduled application or core/thread with a Resource Monitoring ID (RMID) (hardware feature). CMT and MBM monitor the LLC occupancy and memory bandwidth for each assigned RMID. RMID[0] points to the overall LLC occupancy and memory bandwidth usage. The Xeon Gold 5218 supports 127 RMIDs for monitoring.

IV.2.3 XtratuM Hypervisor

XtratuM [15] maintains a Health Monitoring (HM) log and provides an API for user partitions (VMs) to register application-level errors in the log. HM service is also capable of detecting and reacting to certain anomalous states and events. However, these states and events are only those that cannot be handled at the application level, for example, execution of undefined instructions. Therefore, XtratuM does not manage all other partition-level events and states. However, applications can use one of the ten generic error categories, such as internal error, unexpected trap, scheduling error, and overrun, or one of the nine architecture-specific categories, such as undefined instruction and prefetch abort. In addition, it provides an API for system partitions to check the log. Moreover, XtratuM provides the possibility to map predefined actions (such as partition reboot) to each HM event via a configuration file.

IV.2.4 Linux/KVM

The main aim of the Linux Ftrace [183] is to trace the kernel execution flow by attaching callbacks to the start of kernel functions. However, we can use it to attach MONs to the start of kernel functions to collect monitoring information in addition to tracing.

Kprobes [183] allow breaking into any Linux kernel routines to collect information at runtime. Most parts of the kernel can be trapped using Kprobes. Thus, we can use Kprobes inside MONs to gather information from the kernel. Such a MON must be created as a Kernel module.

A tracepoint [183] is a small piece of code placed in the Linux kernel that provides a hook to attach a function at runtime. In applications with custom Linux kernels, we can

add tracepoints at strategic locations in the kernel code and use them to attach MONs to the Kernel at runtime.

Since KVM is based on the Linux kernel, we can use the same approaches in KVM.

IV.3 Existing Monitoring Approaches for RTS and MCS

Bellosa [86] pointed out the possibility of using core PMUs on the SPARC V9 architecture to estimate the memory bandwidth usage of a task by counting the LLC miss event when the task is executing. A similar idea has been used by approaches, such as [65] and [91], for memory bandwidth monitoring in the avionics domain. Memguard [88], a memory bandwidth regulation approach, also uses LLC miss event to regulate the memory bandwidth of a core. For achieving the same on ARMv8 architecture, authors in [81] suggested the possibility of counting L2-cache misses, L2-write backs, or bus access events of the core PMUs.

Obermaisser et al. [184] presented an implementation for TTA that provided deterministic and reproducible monitors without probe effects. The monitors were intended for trace generation and gathering information, such as application status and event detection. Bonakdarpour and Fischmeister presented an interesting tutorial focusing on time-triggered monitors for real-time systems. Bonakdarpour and Fischmeister [185] presented a tutorial on TT monitors for real-time systems. They used methods based on TT path monitoring [186] and TT run-time verification [187]. Medhat et al. [188] proposed a run-time monitor for cyber-physical systems that periodically reads the system state and evaluates the system properties expressed in a specification language. The monitor uses control theoretic techniques to change the monitoring period dynamically to ensure low jitter for monitors and maximize the system utilization.

kritikakou et al. [172] proposed to instrument the code of the critical applications by inserting observation points at fixed locations. Then, at run-time, a monitor executes at each observation point to check if the task may have a potential deadline miss due to multicore contention. The check is based on the isolation WCET and the current task progress. Neukirchner et al. [189] present an approach for monitoring dynamically changing activation patterns of tasks in MCS. The monitor checks if the number of task activations is below a safe upper bound defined by the system designer. The monitoring methodology uses an event model based on arrival functions [190].

Nolte et al. [191] proposed a monitor for Linux to keep an accurate view of the CPU budget used under a real-time Linux scheduling class. The monitor was implemented as a hook to the kernel scheduler tick function. Thus, the monitor gets activated on every scheduling tick and updates the budget consumed by the tasks if a context switch occurs.

The run-time security monitor [192] for RTS takes as input the expected system behavior model. It observes the actual system behavior at run-time and raises the alarm if there is a deviation from the expected behavior.

IV.4 New MONs for RTS and MCS

This section describes the new MONs that we created for use with our resource management framework.

IV.4.1 Hardware MONs

We designed this type of MONs to interface with the existing monitoring features in the hardware. Depending on the events we configure for the PMU, we can use the MONs for directly or indirectly monitoring availability, behavior, reliability, or energy (in specific platforms). These monitors are hardware platform dependent. We implemented three different MONs for use with the following hardware platforms:

1. Zynq 7000 MPSoC (ARM Cortex A9 cores - ARMv7a): baremetal or running XtratuM hypervisor

2. PowerPC T4240 (e6500 cores): baremetal or running XtratuM hypervisor

3. Xeon Gold 5218 (Xeon Scalable 2nd Gen. processor cores): running Linux or KVM hypervisor.

The MONs can interface with PMUs on all the respective platforms. We implemented the MONs in C and partly in Assembly. In each MON implementation:

- A header file contains the list of PMU events.

- The MON code provides functions for the LRM to initialize, reset and read the PMU values. Listing C.L1 in Appendix C shows a pseudo-code of a hardware MON. It should be noted that the subroutines in each MON are internally different as they deal with the hardware registers in the platforms.

There were two main challenges in the design of the MONs:

Challenge 1

Each platform support many common PMU events. However, these events have different names and event numbers; for example, the event for counting clock cycles in e6500 core is called Processor cycles and had event number 1, while the same event for Cortex-A9 is called CPU_CYCLES and has event number 0x11. This makes it difficult for a system designer to configure the resource management framework.

Solution To reduce the difficulty for the system designer to configure the resource management framework, the MONs use the same names for such common events. Still, they point to the correct event number internally. Listing C.L2 in Appendix C shows a code section of the header file for T4240 hardware MON.

Challenge 2

The LRM can periodically read the hardware MON to get the values of the PMCs. Periodically reading the values may be enough for determining the average resource consumption of the applications/VMs. However, we are interested in precisely determining the resource consumption for each application/VMs. Hence, the MON must record values of resource consumption only for the time an application/PMCs is scheduled by an LRS on the CPU core.

Solution for XtratuM XtratuM schedules partitions using a cyclic scheduler (see Section II.13.1 and Section II.17.4). An application (composed of multiple tasks) runs in a partition during the assigned MiF(s) in every MaF. The LRM must execute the hardware MON via an application-level task LRS at the start and end of every partition (or upon completion of all application tasks in the partition) to gather application-level resource consumption. Similarly, the task LRS can run the MON at the start and end of every task to gather task-level resource consumption as shown in Figure IV.F2. The basic idea is to read the PMC values at the start and end of a partition (or every task) and only record the difference between the two values to determine the actual resource consumption of the application (or task). The hardware MON provides two routines for gathering resource consumption of the application as shown in Listing C.L3 (Appendix C). We use and evaluate this approach in the avionics use case (Chapter VI).

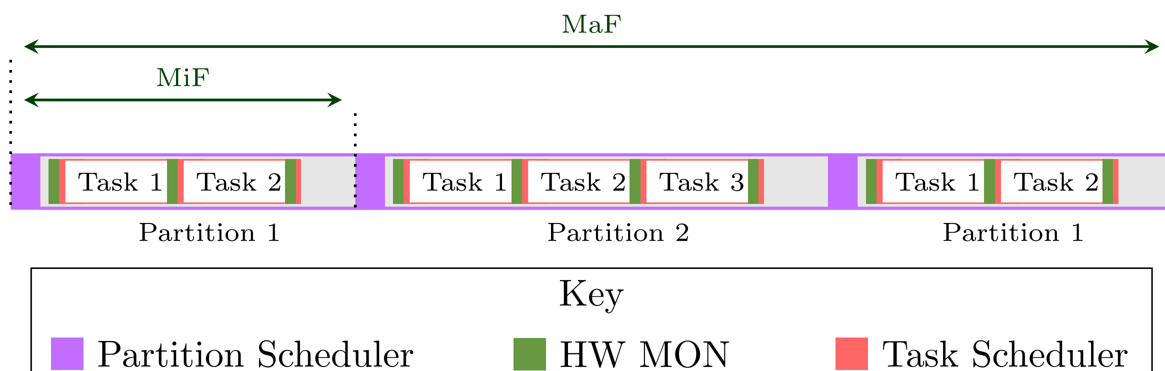


Figure IV.F2: *Hardware MON for XtratuM Partitions*

Solution for Linux/KVM For Linux/KVM, we use tracepoints to register a monitoring hook for a Linux/KVM scheduling (context switch) event, as shown in Listing C.L5 (Appendix C). We chose tracepoints as they have low overheads and are directly usable with the mainline Linux kernel. The monitoring hook (handler) is activated every time a context switch takes place. This monitoring hook saves the difference in the PMC values between two context switches for the task running in between them. Listing C.L4 and Listing C.L5 show pseudo-code for MON hooks in Linux/KVM on x86 platforms (Xeon) and ARMv8a. We use this approach in the railway use case (Chapter VII).

IV.4.2 Hardware Local Resource Monitor (MON) for Intel Cache Monitoring Technology (CMT) and MBM

Apart from the MON for PMU, we implemented a MON for the Xeon platform (running Linux/KVM) to interface with MBM and CMT features. This MON is useful in monitoring availability and behavior of the IMC and the LLC. The basic idea of the implementation is similar to Listing C.L1. Since we can use RMIDs to precisely track the LLC and memory bandwidth usage per task/VM, we do not need to use a monitoring hook for determining the resource consumption of each task. We use this MON in the railway use case (Chapter VII).

IV.4.3 Core Failure MON

Permanent core failure can occur in a system due to changing environmental conditions or phenomena, such as wear-out and infant mortality [173]. An LRM needs to detect core failure(s) to ensure that such failures do not impact the execution of real-time applications.

We[1] designed this MON to detect permanent core failures on multicore platforms. Thus, this MON helps to monitoring the availability (operational status) of a core. The basic idea is to execute a MON instance on each core regularly. If the core is operational, the service operates as expected and updates a shared data structure. However, if there is a core failure, the MON instance cannot execute. Thus, the core cannot update the data structure, and the LRM can catch the failure. For accurate and timely detection of core failures, the MON instances must be precisely scheduled and order by the LRM (via the LRS). With proper planning, we can use such an approach with any scheduler; however, this approach is easy to use with TT scheduling as we can ensure the arrival time and order of the LRM and MON instances. We use and evaluate this MON in the avionics use case (Chapter VI).

As shown in Figure IV.F3, the MON instances running on each core updated a shared data structure (synchronously or asynchronously). However, the MON instances execute at fixed timepoints in every hyperperiod/MaF. All MON instances in the same hyperperiod/MaF write the same value (either 0 or 1) to the data structure. The MON instances in the next hyperperiod/MaF write the toggled value as compared to the previous hyperperiod/MaF. The LRM runs towards the end of the hyperperiod/MaF to detect if a core has failed. For fault tolerance, the LRM instances run on each core. In Figure IV.F3, core 1 fails during the second hyperperiod/MaF. As a result, the core does not update the shared data structure. The LRM instances of the working cores detect this and subsequently consider core 1 as failed (see Section IV.9.1 further explanation). The pseudo-code in Listing C.L19 (Appendix C) shows the essential parts of the core failure MON.

Finally, we can execute the MON (and LRM) instances multiple times in a hyperperiod/MaF on each core to reduce the core failure detection time. The choice of the MON

[1] Together with ONERA and Thales R&T.

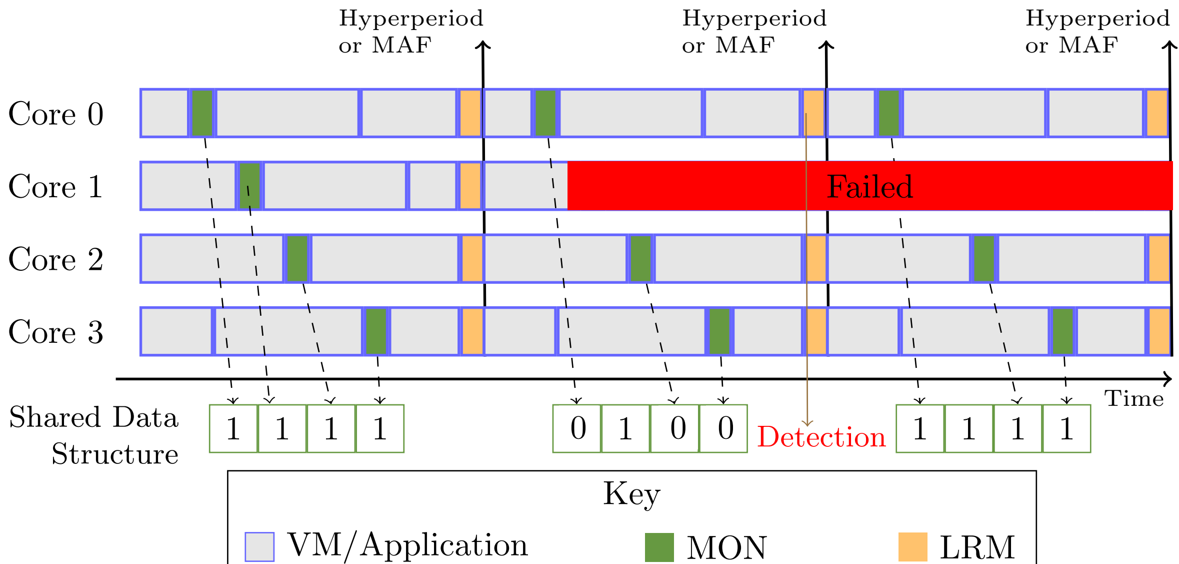


Figure IV.F3: *Core failure MON*

and LRM frequency depends on the system requirements and the available time slots for its execution, i.e., if there is a lot of spare time, the system can use it to execute multiple instances.

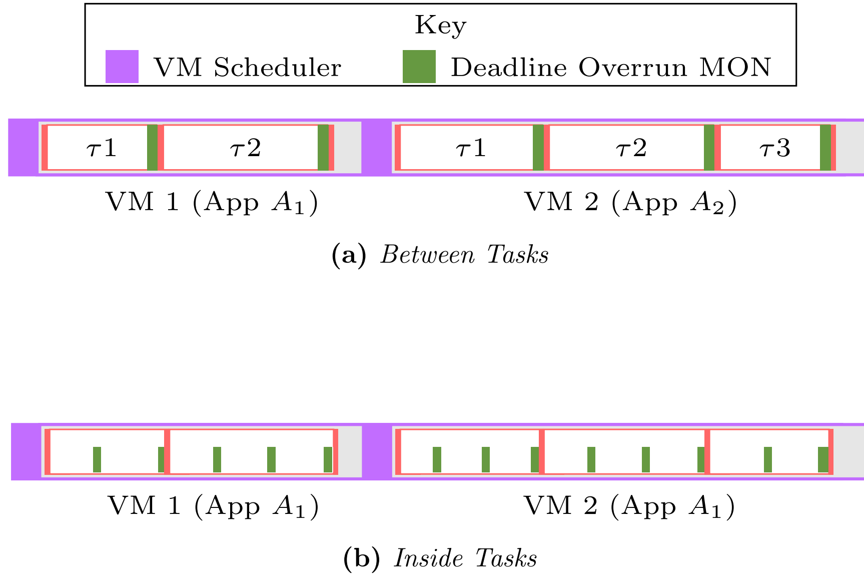
IV.4.4 Deadline Overrun MON

As explained in Section II.25.3, there are conflicting goals for critical and non-critical applications in a MCS: the pessimistic WCET estimations of critical (real-time) applications under-utilize the resources significantly in the average case. On the contrary, the non-critical applications require efficient resource utilization to provide the best possible QoS. We[2] designed this MON to address the issue of running critical tasks concurrently with non-critical tasks. The MON extends the ideas proposed in Kritikakou et al. [172] to allow the detection of potential deadline overrun of critical VMs. Previous work only considered standard tasks sets, and the schedule consisted of executing a task alone on a core. Contrarily, we designed our MON for critical VMs (or ARINC 653 partitions) and provided an interface for use with the LRM. Once a MON detects a potential deadline overrun (reliability monitoring) due to the interference of low criticality applications, the LRM can take preventive measures, such as suspending the execution of low-criticality applications.

We assume that a critical VM run an application A consisting of multiple tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. The critical application uses deadline overrun MON instances to check the execution of its own tasks and checks if the application is in danger of overrunning its deadline. In the case, if a MON instance detects such a danger, it notifies the LRM.

The basic idea is that the application-level LRS schedules MON instances at fixed observation points. The approach in [172] placed an observation point at every instruction.

[2]Together with ONERA and Thales R&T.



(a) *Between Tasks*

(b) *Inside Tasks*

Figure IV.F4: *Deadline Overrun MON Location*

Contrarily, to keep monitoring overheads low, we designed our resource management framework to allow placing the observation points by using one of the following two methods:

1. The easiest option is to place them in between two tasks of a critical application (Figure IV.F4a). The main advantage of this approach is that it does not require instrumentation of critical applications. However, the MONs can only detect potential deadline overruns at task boundaries.

2. Another option is to place observation points inside each task of the application (Figure IV.F4b). In this approach, the MONs can detect potential deadline overruns sooner than the previous approach. However, this approach requires instrumentation of critical application tasks similar to [172].

The MON checks a safety condition to evaluate whether a critical application can tolerate interference by co-executing non-critical applications. The safety condition is given by Equation IV.1.

$$RWCET_A^{iso}(x) + t_{MON} + t_{LRM} \leq D_A - ET_A(x) \tag{IV.1}$$

where:

- $RWCET_A^{iso}(x)$ is the remaining WCET of an application, A, in isolated execution from the observation point x until the end

- t_{MON} is the maximum time from detection by MON until the MON informs the LRM

- t_{LRM} is the overhead of LRM for performing preventive measures

- $ET_A(x)$ is the execution time used by A until the observation point x, as monitored using a Hardware MON (from Section IV.4.1)
- D_A is the deadline of A.

If the application-level LRS in the VM use offline (cyclic) scheduling, we can place observation points at offline determined timepoints. This helps us to simplify the safety check performed by the MON. Equation IV.2 presents the simplified safety check.

$$ET_A(x) \leq D_A(x) \tag{IV.2}$$

where $D_A(x)$ is the offline determined maximum possible internal deadline of application A at observation point x.

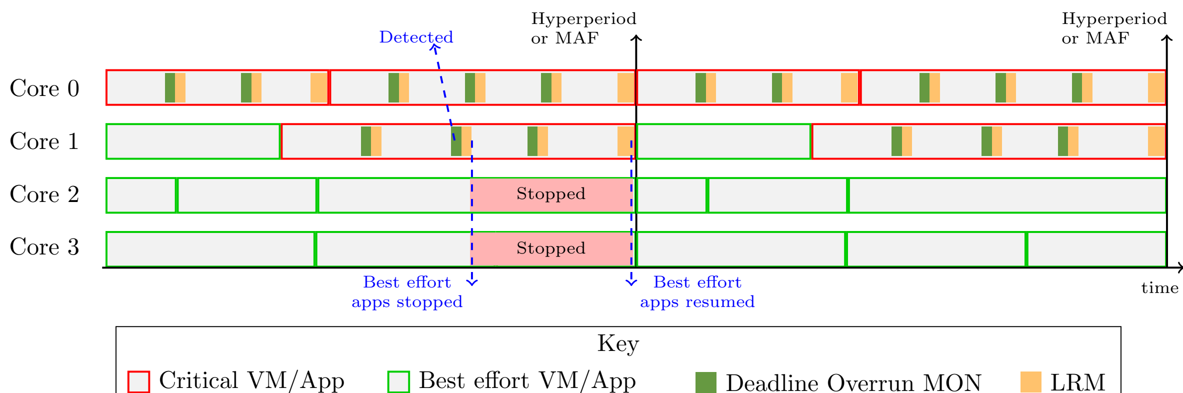


Figure IV.F5: *Deadline Overrun MON*

Figure IV.F5 illustrates this approach with an example. In the example, only core 0 and core 1 run critical tasks, while the remaining cores execute only non-critical tasks. A MON in an application on core 0 detects a potential deadline overrun. It informs the application-level LRM, which in turn takes preventive actions (stopping co-executing non-critical applications on core 2 and core 3). The pseudo-code in Listing C.L7 (Appendix C) shows the essential parts of the this MON using the first method (observation points between two tasks). The observation points can be specified via the system designers using the resource management configuration file as shown in Listing C.L8.

IV.4.5 Health MONs (XtratuM)

We designed these application-level MONs for use with XtratuM hypervisors partitions (reliability monitoring). However, a similar idea applies to other hypervisors as well. We provide application-level MONs that can use the XtratuM HM API to write application status or errors (not handled by XtratuM) to the HM log (Listing C.L9 in Appendix C). An LRM running in a system partition can read this log file and perform advanced error handling. Furthermore, our resource management framework (Section III.9) accepts as

input (optional) an HM event and action mapping from the system designer (Listing C.L10) and generates the XtratuM configuration, including these mappings.

IV.5 Local Resource Scheduler Services

Each resource managed by the resource management architecture is paired with a LRS. Each LRS schedules the use of the resource and controls application access to the resource. The LRS services and the LRS implementation are specific to each resource. In general, the LRSs support different scheduling policies, which can be classified at a high level as offline scheduling and online scheduling. Examples of scheduling services are dispatching time-triggered messages, scheduling tasks according to offline tables, setting online scheduling parameters, allocating cache or memory space to VMs, execution of VMs on top of a virtualization layer, execution of tasks inside a VM, processing of queued memory and I/O requests, and dispatching of time-triggered and aperiodic messages at the network interfaces. (Re-)configuration services refer to the ability of an LRS to accept orders from LRM for executing changes and updates on its own configuration.

IV.6 Scheduling Opportunities

All OSs provide some mechanism to schedule tasks. Similarly, hypervisors provide means to schedule VMs. Many OSs and hypervisors also provide means to allocate memory space to a task/VM. There are also some existing opportunities at the hardware level for scheduling and allocation of resources.

IV.6.1 XtratuM Hypervisor

XtratuM [15] schedules partitions using cyclic scheduling (ARINC 653 scheduling policy). Cyclic scheduling ensures that a partition cannot use a CPU core for longer than the allocated time. Thus, a partition cannot impact the scheduling of another partition. Time slots for each partition must be defined in the XtratuM configuration file. A system designer can define the time slots for each partition easily via the resource management framework configuration file(see Section III.9) as shown in Listing C.L12 (Appendix C).

If several tasks are assigned to a partition, the partition must implement its own scheduling algorithm (hierarchical). Section IV.7.1 presents an application-level LRS that we implemented to schedule tasks in a bare-metal partition or a VM. This LRS can be used to schedule tasks in an XtratuM partition.

As explained in Section II.16, RTS and MCS exhibit multiple operation phases and require mode changes. XtratuM implements mode changes via scheduling plans. We exploit this feature to provide fault-tolerance upon failure of core(s) by changing XtratuM plans (see Section IV.7.2). Scheduling plans must be defined via the XtratuM configuration file. A system designer can define these plans easily via the resource management configuration file (Listing C.L12 in Appendix C).

IV.6.2 Linux/KVM

Recently, industry and academia have directed consideralbe effort towards Real-time Linux. Linux supports a variety of new real-time features, such as SCHED_DEADLINE scheduling policy (based on EDF and CBS) [120], priority inversion control (in locking algorithms) and PREEMPT_RT (a fully preemptible kernel mode) [193]. Older real-time scheduling policies in linux include SCHED_RR (based on RR scheduling) and SCHED_FIFO (based on FIFO scheduling) [183]. Both these policies implement static priorities.

Since KVM is based on the Linux kernel, we can use the same scheduling algorithms in KVM to schedule VMs. Moreover, KVM supports features, such as allocation of CPU cores and memory space to VMs. Chapter VII gives an overview of many other KVM features.

IV.6.3 Intel CAT and MBA

Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) [102] are hardware functionalities supported by Intel Xeon Scalable (2nd Gen.) processors [30] to allocate LLC space and memory bandwidth to a core, thread, application, or VM [102]. Both techniques work based on an intermediate construct called CLasses Of Service (CLOS). A CLOS acts as a resource control tag and can be assigned to a single or group of cores, threads, applications, or VMs. The members of a CLOS can only use the resource capacity (amount of LLC space or memory bandwidth) assigned to the CLOS.

To allow cache allocation to a CLOS, Xeon provides capacity bitmasks that indicate how much cache can be used by a CLOS. In addition, to support MBA, Xeon has a programmable request rate controller for memory bandwidth allocation between the private caches and the high-speed interconnect to the LLC, as shown in Figure IV.F6. A user can configure this rate controller with a discrete delay value. The figure also shows an example of cache allocation to CLOS with a 20 bit mask.

Our selected hardware (Intel Xeon Gold 5218) (Section II.4) supports MBA and CAT as shown in Table IV.T2 [31]. A MBM delay value of 0 adds no delay to the request, while a delay value of 90 adds the maximum possible delay. Unfortunately, Intel does not provide the relation of discrete delay values to the actual inserted delay (in time units) for memory requests in the public domain.

IV.6.4 TT-NoC Scheduler

Ahmadian et al. [195, 196] proposed a TT NoC synthesizable for the PL of Xilinx Zynq 7000 MPSoC and developed an LRS for use with our resource management to interface with the NIs of the NoC. The avionics demonstrator of the DREAMS project used this TT NoC and LRS.

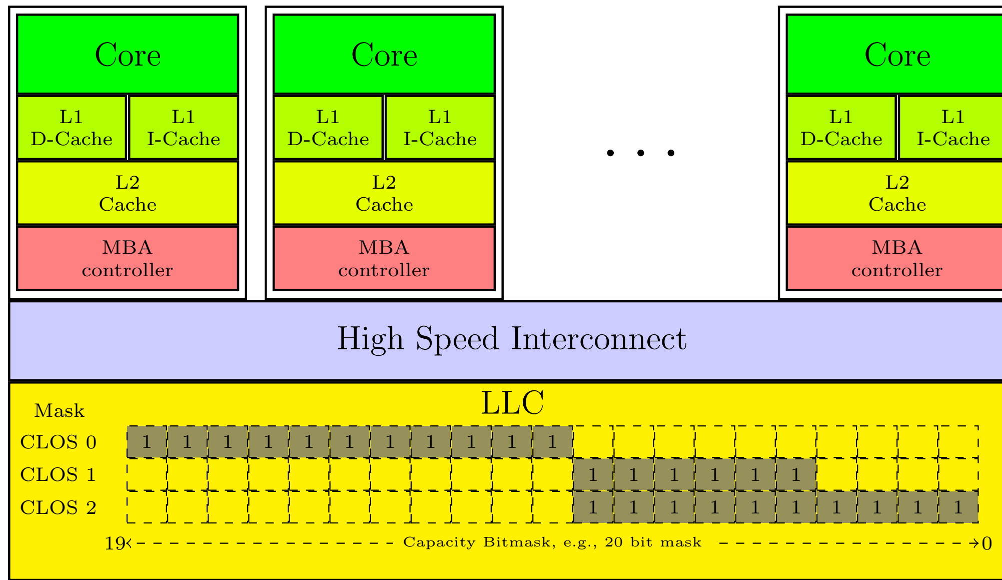


Figure IV.F6: *Intel CAT and MBA*

Table IV.T2: *MBM and CAT properties Intel Xeon Gold 5218*

Feature	MBA	CAT	Errata
CLOS	8	16	-
Capacity bit mask	-	0x7ff	-
Discrete delay values	0, 10, 20, 30, . . . , 90	-	Values $>$ 10 and $<$ 40 written to the MBA delay value register (MSR 0xD50 to 0xD57) may be read back as 10 [194]. We experimentally evaluated that this issue is present in our platform. Delay value of 10, 20, and 30 have the same effect on the memory bandwidth allocation.

IV.6.5 Existing Approaches for Scheduling RTS and MCS

- Section II.7 and II.8 presents some common real-time and mixed-criticality task models for scheduling.
- Section II.11 gives an overiew of TT and joint TT-ET scheduling techniques.
- Section II.15 provides a comprehensive survey about approaches addressing shared resource contention in multicore systems.
- Section II.16 presents an overview of scheduling techniques that implement mode changes.
- Section II.20.2 presents existing approaches that provide fault-tolerance via reconfiguration.

- Section II.13 gives an example of TT network scheduling.

IV.7 Newly Implemented LRSs

IV.7.1 LRS for Critical Partitions (XtratuM)

We[3] created this LRS for use with a safety-critical (avionics) application running bare-metal in a ARINC 653 partition (for example, XtratuM VM). The LRS runs in at least two modes:

1. The LRS starts in the initialization mode during its first execution instance. It launches the application initialization to set up its internal state for execution, followed by initializing the application task schedule for different execution slots (MiF) and planned modes (or configurations). The pre-computed schedules can be stored in an additional configuration file or supplied to the LRS directly via the application.

2. Once the LRS completes the initialization, it changes to a execution mode and waits for the beginning of a slot (assigned to the application). In this mode, the LRS sequentially executes the application tasks as per the predefined schedule. Once all tasks assigned to a slot have finished execution, the LRS stops the further execution even if there is time remaining in the slot. For simplicity, we assume that the execution starts and ends in the same slot. However, conceptually, it is possible that the execution can span multiple slots, i.e., it can start and end in different slots. The LRS repeats the execution in each new slot in this mode unless the LRM reconfigures it to execute in a different mode.

The critical partition application must follow a specific design to use this LRS. The LRS applies the schedule to the application tasks by providing the following interfaces:

1. Application callback interface for application initialization.

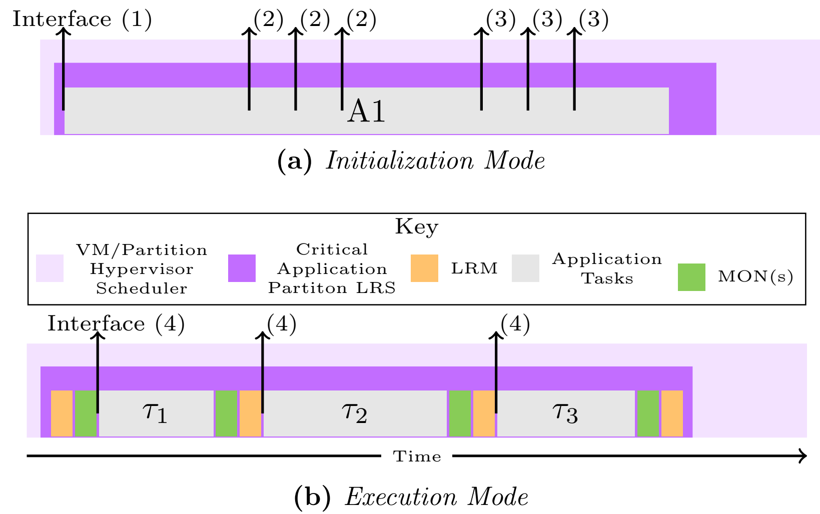
2. Interface for applications to declare tasks.

3. Interface for applications to declare the offline computed schedule of tasks.

4. Task callback interface for the LRS to launch the task execution.

Figure IV.F7 shows an example of the LRS executing a application $A1$ with tasks $\{\tau_1, \tau_2, \tau_3\}$ during initialization and execution modes. In the initialization mode, the LRS uses Interface 1 to initialize the application, Interface 2 and 3 to declare the three tasks and their schedules. In the execution mode, LRS first executes the (application/VM domain) LRM. The LRM can send any stop signals to critical applications if required by the deadline MON from the previous instance. Next, the LRS executes the Hardware

[3]Together with ONERA and Thales R&T.



(a) *Initialization Mode*

(b) *Execution Mode*

Figure IV.F7: *LRS for Critical Partitions*

MON to initialize it for this application instance. After executing the MON, the LRS executes the first task according to the offline defined schedule using Interface 4. This is followed by executing the hardware MON and deadline overrun MON to check for any potential deadline miss. Next, the LRS executes the (application/VM domain) LRM to check if any preventive actions are required (e.g., stopping co-executing non-critical applications). Other tasks are executed similarly. After executing the last task, the LRS runs the hardware MON and the LRM one last in that slot. The hardware MON updates the application statistics and status. Finally, the LRM undoes any preventive measures (if applicable) and sends required updates to the higher-domain LRM (or GRM). To end, the LRS stops any further execution in the current slot. The pseudo-code in Listing C.L11 (Appendix C) shows the essential parts of this LRS. We use this LRS in the avionics use case (Chapter VI).

IV.7.2 LRS for Online Reconfiguration (XtratuM)

We designed this simple LRS to assist the LRM in reconfiguring scheduling plan (modes) in XtratuM. XtratuM supports immediate and deferred plan changes. The pseudo-code in Listing C.L13 (Appendix C) shows the essential parts of this LRS. We use this LRS in the avionics use case (Chapter VI).

IV.7.3 LRS for Intel CAT and MBA

We designed this LRS to provide an interface for the resource management to interact with the CAT and MBA hardware features. Figure IV.F8 shows the essential functions of this LRS. It takes configuration orders from the LRM and performs two major functions:

1. It allocates the memory bandwidth and LLC space to each CLOS by setting the delay values and bitmasks, respectively.

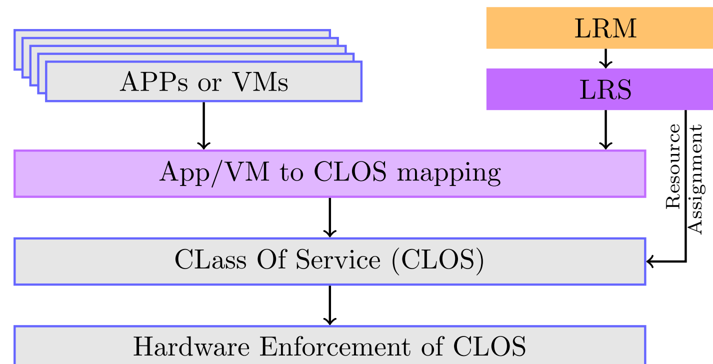


Figure IV.F8: *LRS for Intel CAT and MBA*

2. It maps applications to CLOS.

The pseudo-code in Listing C.L11 (Appendix C) shows the essential parts of this LRS. We designed this LRS for the railway use case (Chapter VII).

IV.7.4 TT LRS for Linux/KVM

We designed this LRS to support scheduling of TT tasks/VMs in Linux/KVM (with PREEMPT_RT patch). The LRS introduces a new scheduling policy, *SCHED_TT_LRS*, in Linux. The CPU cores running TT LRS have a *sparse timebase*. Thus, an event can occur only at some fixed instances in the timeline, and there are time intervals between the fixed instances when no events can occur. We refer to these minimum scheduling quanta as *slots* in the context of the TT LRS.

The new scheduling policy has the highest priority and sits on top of the hierarchy of (native) Linux scheduling modules: $TT_LRS > RT$ (Real-Time policy) $> CFS$ (Completely Fair Scheduling policy) $> IDLE$. If there are no runnable tasks of TT LRS scheduler module, the Linux scheduler will look for a runnable task of each module in decreasing priority order.

Unlike the existing scheduling modules in Linux, the LRM must explicitly enable the TT LRS on a set of CPU cores. The LRM must also provide the LRS a list of TT tasks to run in each slot (and core) as per the scheduling table (see Figure IV.F9). The LRM runs as a real-time Linux task on a non-TT (housekeeping) core and has a period of one slot length. Hence, an LRM instance occurs once in every slot. Note that similar to Section IV.7.1, the LRS runs hardware, core failure, and deadline overrun MONs (optional) before executing a task/VM. The figure omits these MON for simplicity.

Linux has a modular scheduling framework, and it implements each module as a set of functions specified in the SCHED_CLASS structure. Listing C.L15 (Appendix C) shows the definition of LRS_TT scheduling class, which implements the TT_LRS module.

The fields in the class are functions that act as a call back upon occurrence of specific events. The following two are the most important functions of this scheduling class:

1. The task_tick_TT() function is called by Linux every scheduling tick (1ms) on each TT CPU core. Every time the function is called, the TT_LRS of the core

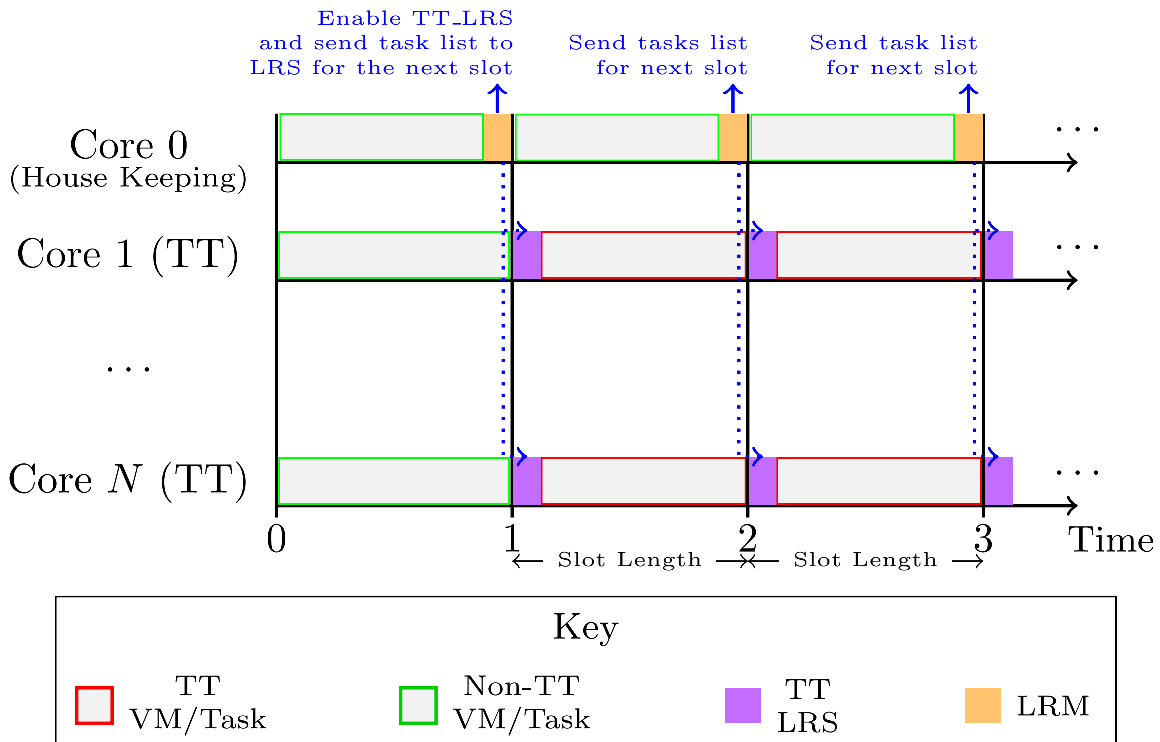


Figure IV.F9: *TT LRS for Linux/KVM*

reduces a counter (tick_countdown) by 1. The counter is initialized as shown in Equation IV.3.

$$tick_countdown = \frac{Slot_size(inms)}{Tick_size(inms)} \tag{IV.3}$$

In our case, the tick size is $1ms$. Thus, for example, to set a slot size of $1s$, we need to set the value of tick_countdown to 1000.

Only when the tick_countdown is zero (end of a slot), the LRS checks if the LRM has assigned a new task for its CPU core. If there is a new task available, then it marks the core for rescheduling. Listing C.L16 (Appendix C) shows the essential parts of this function.

2. The pick_next_task_TT() function is called by the Linux __schedule() function on each core. __schedule() is main entry point into the Linux task scheduler. The main job of __schedule() is deciding which task to run next and then execute it. To decide which process to run next, __schedule() uses the pick_next_task_TT() function, when a TT class task is available. pick_next_task_TT() function returns the task_struct of the task to be scheduled in the upcoming slot. If the task to be scheduled resides on the run queue of another core, then this function migrates the task to the run queue of the current core. Listing C.L17 (Appendix C) shows the essential parts of this function.

Finally, the LRS provides a function TT_LRS_set_schedule() to be used by the LRM

to provide the TT_LRS running on each core with a task id every slot. Moreover, this function stores the list of tasks running the previous core for use by task_tick_TT() function. We reduce the overheads of task_tick_TT() by using the list of previous tasks as we do not need to mark the core for rescheduling if the next task to be executed is the same as the previous task. Listing C.L18 (Appendix C) shows the essential parts of this function. We designed this LRS for the railway use case (Chapter VII).

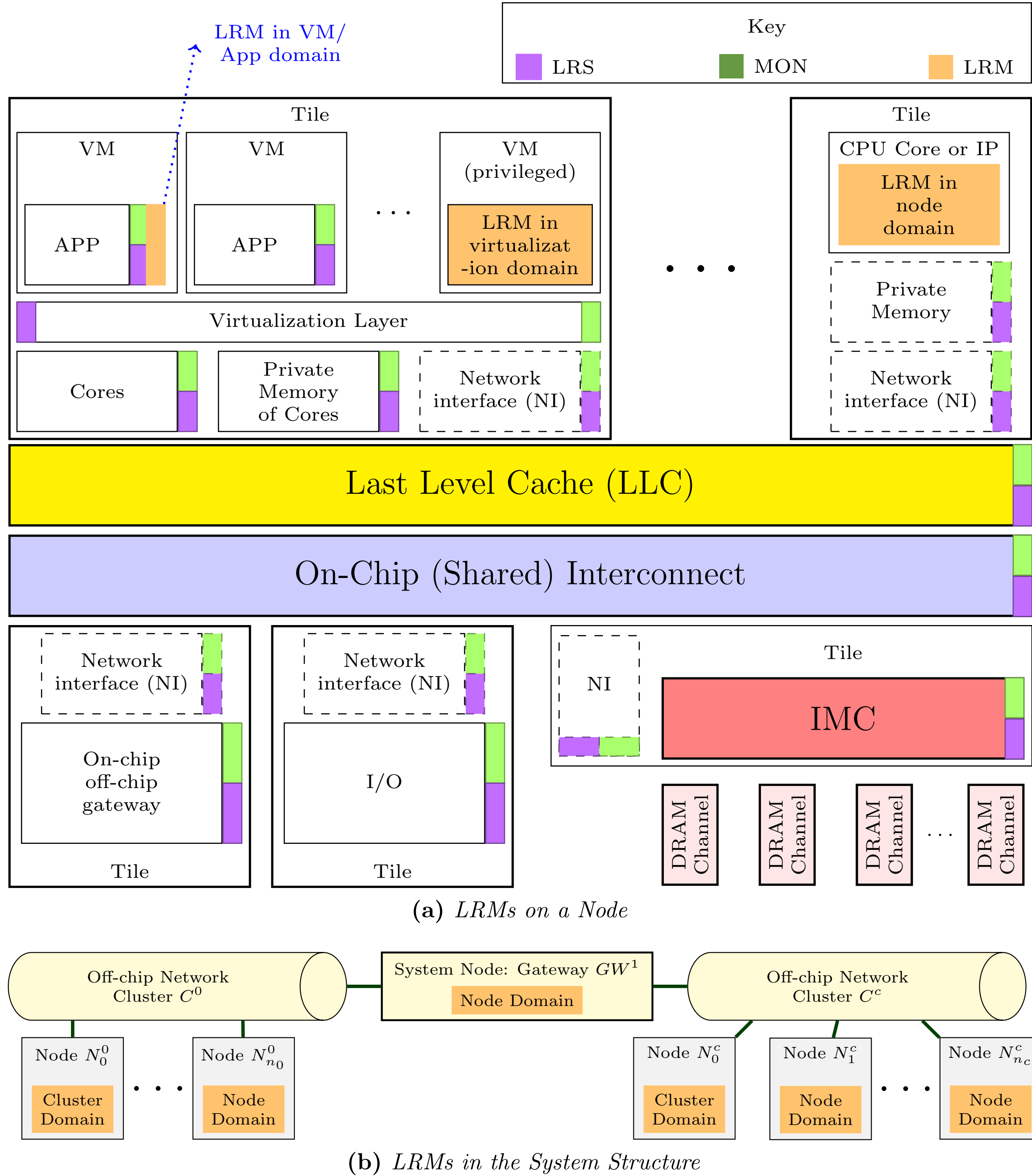
IV.8 Local Resource Management Services

In general, all local resource management components can have hardware or software implementations, or a combination of both, depending on the domain and type of resource. The actual physical implementation of the local building blocks is use case-specific. Figure IV.F10 presents examples of where it is possible to implement LRMs in different domains. The LRM in the node domain (Figure IV.F10a) can be implemented in hardware as a IP block or as a software (application) executing directly on a CPU core. This LRM supervises and controls the LRMs in the virtualization layer(s) of the node and all non-virtualized resources. That includes the LRSs and MONs of the memory hierarchy (private caches, shared LLC and IMC), I/O component(s), on-chip/off-chip gateway, and NI (if applicable). The LRM in the Virtualization Layer domain must be a privileged VM that can (re-)configure the hypervisor. This LRM is in charge of supervising and controlling the LRSs and MONs (e.g., VM scheduler and health Monitors) for virtual resources provided by the hypervisor. This LRM may also directly supervise and control application components (e.g., guest OS scheduler or application QoS monitors) if the guest does not have its own VM domain LRM. Finally, Application/VM domain supervises and controls application components. Figure IV.F10b shows an example of where it is possible to implement LRMs in the system structure.

Figure IV.F11 summarizes the important local resource management services provided by the LRMs in various resource management domains. These services include the following:

Read Information from MONs

Two paradigms are possible for gathering information from MONs: interrupt and polling. In the first case, the MONs send information periodically to the LRM, while in the second case, the LRM requests information from the MONs. Of course, a combination of the two approaches is also possible. Alternatively, it is also possible that the LRM triggers/executes diagnostics routines that the Monitoring services would evaluate. For example, LRM can trigger DRAM Tests, instruction execution tests on a core, Cyclic Redundancy Checks (CRC) on memory regions, Digital I/O Tests(such as toggling output signals and reading them back), and Analog Digital Converter (ADC) Converter test by applying predefined signals.



(a) *LRMs on a Node*

(b) *LRMs in the System Structure*

Figure IV.F10: *Implementation of LRMs in Different Domains*

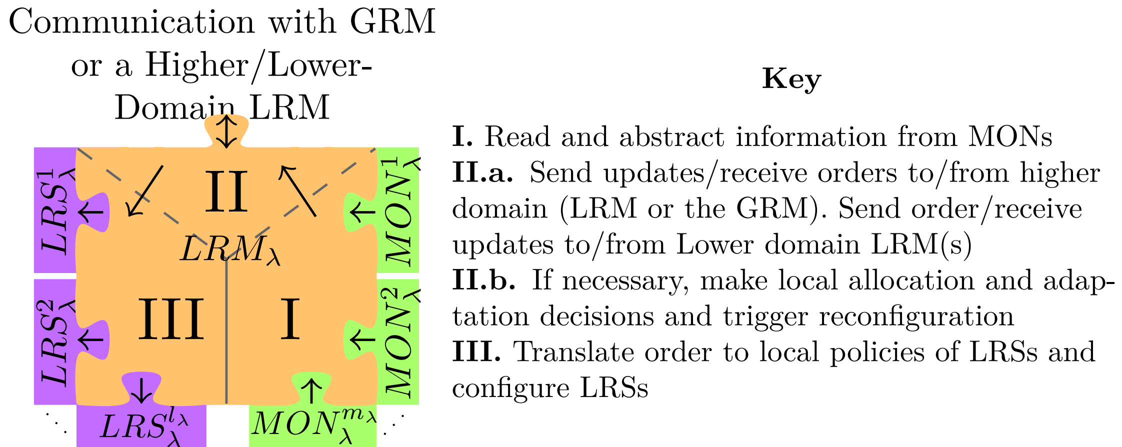


Figure IV.F11: *LRM services*

Calculate Abstract Resource State

As explained in Section III.3, the resource management overheads must be kept to a minimum. Suppose the LRMs communicate the change in resource availability to the GRM (or a higher domain LRM) at every fluctuation in demand or availability of the resources. In that case, the resource management will have very high overheads. Moreover, the resource management will not be able to keep up with the fluctuations, and the resource management decisions from the GRM (or a higher domain LRM) will be made too late. Only the long-term changes in resource availability are of interest to the next higher domain. Thus, taking inspiration from the Matrix framework [21] and the ACTORS project [22], the LRMs calculate abstracted state variables of the resources based on monitored information from the MONs. Abstract state variables can be energy, availability, reliability, behavior, among others. Figure IV.F12 shows an example of resource availability abstraction. In the figure, the LRM abstracts the resource availability in one of the three resource levels: *HIGH*, *MEDIUM* or *LOW*. The level of abstraction depends on the specific resources and available monitors. This approach provides a resource view on an abstract level to reduce the overhead of disseminating the low-level monitor variables and only present the minimum information required for resource management by the GRM (or the next higher domain LRM). For example, in Figure IV.F12, the LRM only sends a higher-domain (LRM or the GRM) if the resource availability level (*HIGH*/*MEDIUM*/*LOW*) changes, instead of sending an update at every fluctuation.

Send Updates/Receive Orders to/from the GRM or a Higher-Domain LRM

Each LRM transmits abstracted updates of resources in its domain to a higher-domain (LRM or the GRM), whichever is next in the hierarchy. The actual transmission takes place via the network and middleware.

The LRMs can only receive orders from a higher-domain LRM or the GRM, whichever

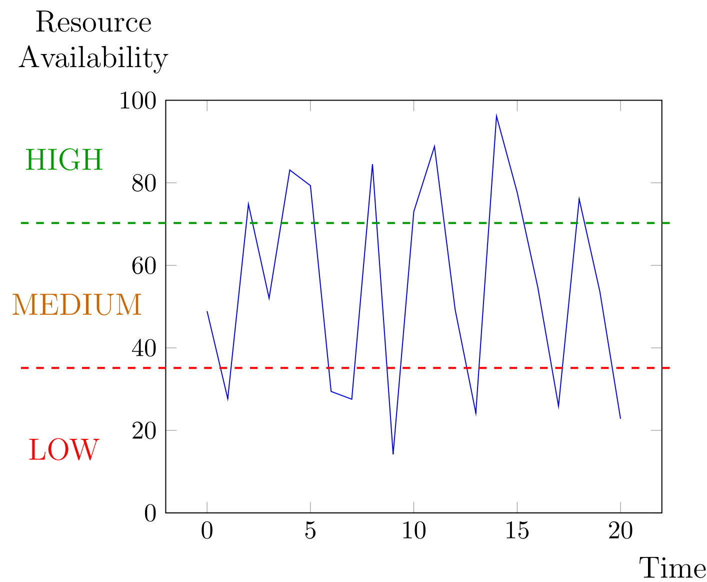


Figure IV.F12: *Example of Resource Abstraction by LRM*

is next in the hierarchy. However, an LRM never receives orders from applications, other system components, or LRMs at the same or lower levels in the hierarchy.

Send Orders/Receive Updates to/from Lower-Domain LRM(s)

LRMs receive updates and send orders (when necessary) to lower-domain LRM(s) directly below them in the hierarchy.

Local Decision-Making and Reconfiguration

Small changes or fluctuations in the availability of resources or demands of applications can be handled locally by an LRM in its domain. For example, a virtualization-domain LRM can suspend a low-criticality VM that only communicates with other low-criticality applications on the same node. In that case, the LRM will report the new state of the resource (the application tile) to the GRM to maintain coherence in the system's state. An LRM only requests a reconfiguration from a higher domain if it cannot handle a (long-term) change locally.

Translate Orders to Policies of LRSs

After receiving an order from the GRM or a higher-domain LRM, the LRM maps it to the local scheduling policies of the LRS of the resource. For LRSs that implement online scheduling of resources, the LRM can convert the orders to scheduling parameters for the corresponding LRSs. On the other hand, the LRM can select or search appropriate scheduling tables for the LRSs based on the received orders in the case of table-driven scheduling policies. This approach is based on the conceptual separation between the

implementation details of the resource schedulers and the abstract view of the resources that the GRM maintains.

Configure LRSs

The LRM configures the LRSs in its domain (and sends orders to the LRMs in the lower domains). For example, the LRM can set the selected parameters or provide the scheduling table (directly or indirectly via a reference) to the LRSs.

IV.9 Local Resource Management Policies

The required local resource management policies are use case specific. In this dissertation, we[4] present LRM policies to manage the following:

1. permanent failure of CPU cores, and
2. node-level temporal overload situation (by potential deadline overrun monitoring).

IV.9.1 Core Failure Management

Permanent core failure can occur in a system due to changing environmental conditions or phenomena, such as wear-out and infant mortality [173]. When a core fails, the applications or VMs hosted on it cannot execute further. Such a situation can be mitigated by an active redundancy (Section II.20.1) or by performing a reconfiguration (Section II.20.2) to reallocate the cores to applications. Since modern multicore platforms provide a high number of CPU cores, we decided to manage core failures via reconfiguration. This section proposes a LRM policy to provide core failure management.

Suppose, an LRM detects core failure(s) via a MON (section IV.4.3). Then, the LRM can locally reconfigure the node. Local reconfiguration involves reallocating the available cores to the applications, translating the allocation to LRS (Task scheduler of OS or VM scheduler of hypervisor) policy, and configure the LRSs accordingly. The LRM can reallocate cores based on:

1. Offline pre-computed schedules: for example, the LRM can contain multiple scheduling tables, each considering a different number of available CPU cores. The LRM selects an offline table with an appropriate number of available cores. We use this approach in the avionics use case (Section VI.4).
2. Online scheduling decisions: Instead of selecting from available offline schedules, the LRM calculates new schedules online based on the reduced availability of cores. The LRM may find a schedule where all applications are schedulable. Otherwise, it will drop some applications.

In both online and offline scheduling, the LRM may or may not find a suitable schedule.

[4]Together with ONERA and Thales R&T.

- If the LRM finds a suitable schedule that can host all the existing applications, it reconfigures the LRS for using this new schedule. In offline scheduling, the LRM will send (directly or indirectly via a reference) the new scheduling table for the LRS and configure the LRS to use this table. In online scheduling, the LRM will change some LRS parameters, such as the number of available CPU cores. After reconfiguring the LRS, the LRM sends an update to the GRM informing about the local reconfiguration.

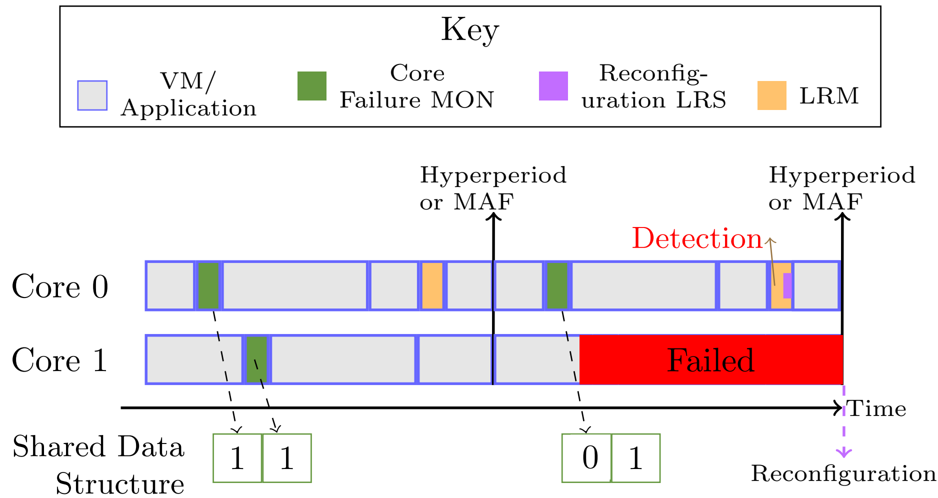
- If the LRM finds no schedule for local reconfiguration that can schedule all existing applications on the node, then it cannot host all the applications. Therefore, it searches for a schedule by removing some applications. In general, the LRM removes the best-effort applications before the critical ones. The LRM removes critical applications in order of their priority. The system designer must configure the order of removal and priority of the applications.

  - For offline scheduling, this implies that different scheduling tables may schedule a different number of applications. For example, the first scheduling table allocates N cores to M_1 applications (or VMs), the second table allocates $N-1$ cores to M_2 applications, and so on. $M_1 = M_2 = \ldots$ until there is enough CPU capacity available to schedule all applications while satisfying the constraints; otherwise $M_1 > M_2$. The offline tables, by design, ensure that the LRM removes low priority applications before high priority ones. After finding a new table and reconfiguring the LRS, the LRM sends an update to the higher domain (GRM or LRM), informing about the local reconfiguration, including the selected table (by sending it directly or via a reference).

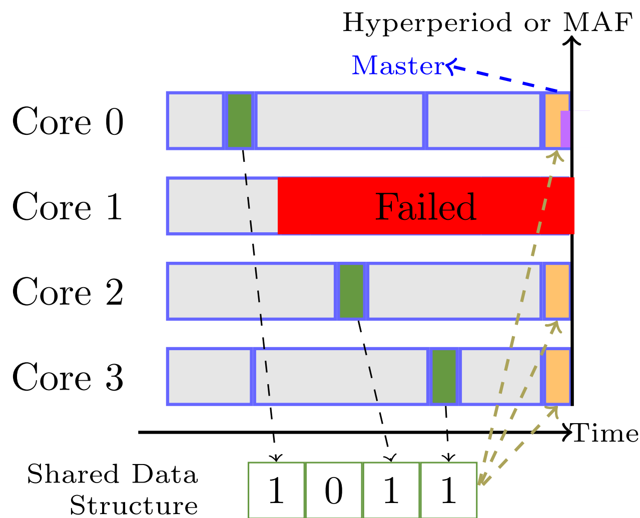
  - For online scheduling, the LRM removes applications based on a heuristic and changes scheduling parameters (if necessary). Then, it performs schedulability tests to check whether the new set of applications are schedulable. Finally, after finding a new table and reconfiguring the LRS, the LRM sends an update to the higher domain (GRM or LRM) informing about the local reconfiguration, including dropped applications and new scheduling parameters.

  The higher domain (LRM or GRM) must address the applications removed from the node. We assume that only complete applications can be removed from a node or redeployed to another node.

In all the mentioned cases, the LRM can either perform an immediate or a deferred (at the end of hyperperiod or MaF) reconfiguration (similar to immediate or deferred mode changes). The relationship between the execution of a core-failure MON and the execution of the LRM directly influences the response time for local reconfiguration upon core failure. For immediate reconfiguration, the least response time occurs when the LRM executes directly after the MON instance detecting the core failure. On the other hand, for deferred reconfiguration, the best response time occurs when the LRM executes after the MON in the same hyperperiod or MaF (Figure IV.F13a). We present a concrete implementation for this approach in the avionics use case (Section VI.4).



(a) *Example of Deferred Reconfiguration by the LRM*



LRM Master Order=
[Core 0, Core 1, Core 2, Core 3]

(b) *Synchronous LRM execution on each core to deal with core failures*

Figure IV.F13: *Core Failure Management by LRM*

A core executing the LRM can also fail and render the system without resource management capabilities. We execute an LRM instance on each core at the end of every hyperperiod/MaF to deal with such a situation. The LRM instances in every hyperperiod/MaF run synchronously (Figure IV.F13b). Among the LRM instances, one of them is a *master*. Only the master can perform reconfiguration and communicate with the higher domain LRM or the GRM. If the core executing the master fails, another LRM instance on an active core becomes the master. A system designer must add this order of selection in the resource management configuration file (Listing C.L19 in Appendix C).

IV.9.2 Temporal Overload Management

As explained in Section II.25.3, if we provide resources to application based on their WCET, we under-utilize the multicore platform in an average case as the Average Execution Time (AET) is much lower than the WCET. Thus, it is practical to over-utilize a multicore platform to reduce SWaP and costs. This section presents the temporal overload management strategy[5] (via potential deadline overrun monitoring) for use by an LRM.

Let us assume we have critical applications $\{A_1^c, A_2^c, \ldots, A_I^c\}$ and best-effort applications $\{A_1^{be}, A_2^{be}, \ldots, A_J^{be}\}$. Three conditions must be fulfilled to over utilize a multicore platform with N cores, without endangering the critical applications.

1. The worst case utilization (U) of the critical applications must be less than the number of cores on the hardware platform.

$$\sum_{i=0}^{I} U_i^c < N \tag{IV.4}$$

2. The sum of the Average Utilization (AU) of the critical applications and the best-effort applications must be much lower the number of cores on the platform.

$$\sum_{i=0}^{I} AU_i^c + \sum_{j=0}^{J} AU_j^{be} << N \tag{IV.5}$$

3. The sum of the worst case utilization (U) of the critical applications and the best-effort applications must be greater than the number of cores on the platform.

$$\sum_{i=0}^{I} U_i^c + \sum_{j=0}^{J} U_j^{be} \geq N \tag{IV.6}$$

As a result, only best-effort applications can lead to overutilization of the multicore platform.

[5]Develped together with ONERA and Thales R&T.

Suppose a deadline overrun MON (Section IV.4.4) detects a potential deadline miss. Then, the LRM interrupts best-effort applications to ensure that the critical ones can execute safely. The LRM resumes the best-effort applications once the the application finishes its execution in the current hyperperiod /MaF.

The execution of LRM with respect to the MON has a tremendous influence on the adaptation response time. The adaptation may be occurring too late if the LRM is not being executed when the MON detects a potential overrun. Such an adaptation will be pointless. Thus, we consider three options to execute the LRM:

1. Dedicate an entire core to execute the node domain LRM (Figure IV.F14a): This approach provides almost immediate adaptation, i.e., $t_{LRM} = 0$ in the safety condition (Equation IV.1) of the MON; however, the approach has lower performance as an entire core is dedicated to the LRM.

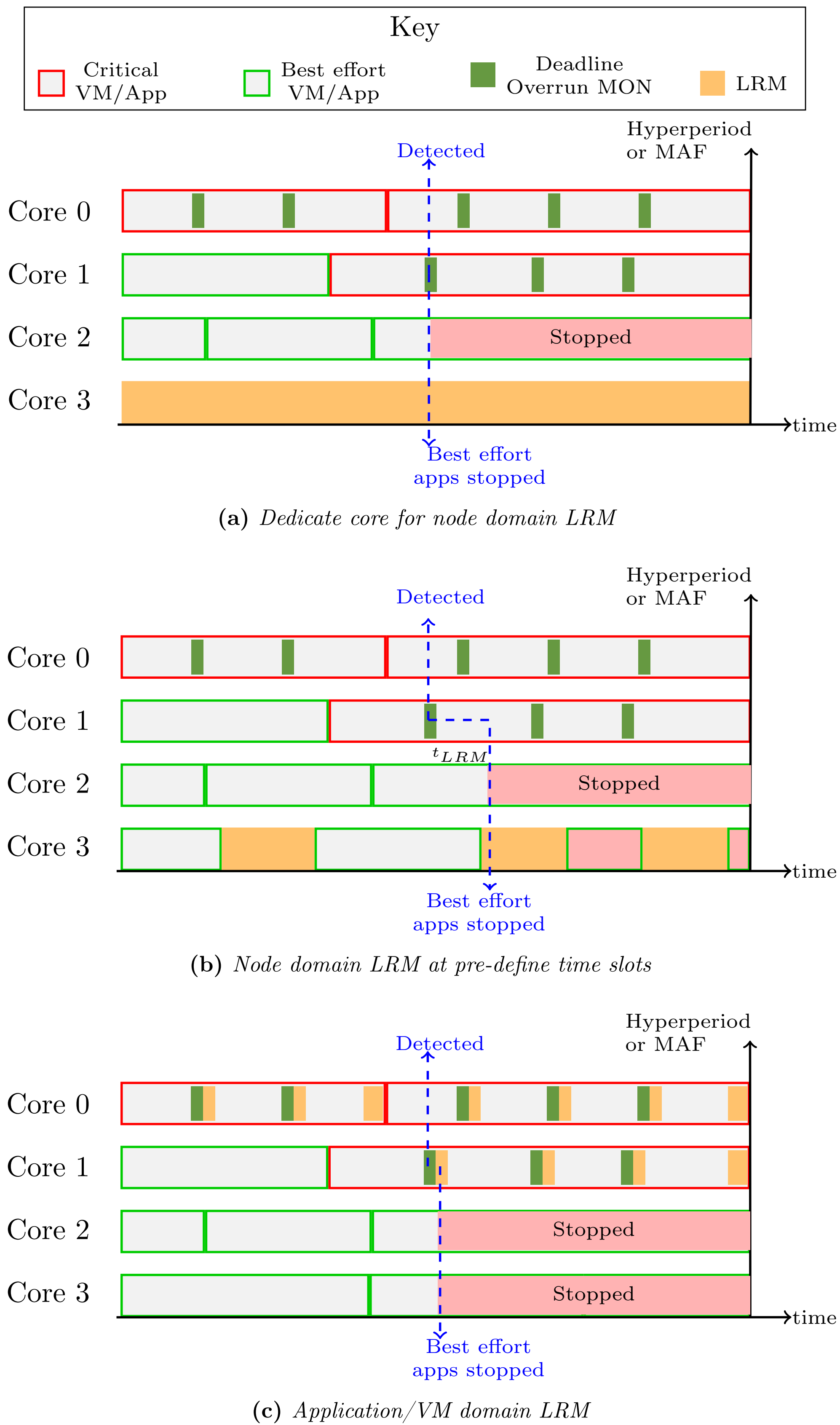
2. Execute a node domain LRM at predefine time slots (Figure IV.F14b): This approach utilizes the CPU better; however, it has a much longer delay from the time the MON detects the overrun to the time the LRM applies the adaptation. In the safety condition (Equation IV.1) of the MON t_{LRM} = time to next LRM instance from the execution of the MON. Another disadvantage of this strategy is that it requires careful time planning of MON and LRM instances.

3. Adaptation by application/VM domain LRM executing directly after the MON instance (Figure IV.F14c): In this approach, the application/VM domain LRMs provide the adaptation. The adaptation is almost immediate as the LRM executes directly after the MON, i.e., $t_{LRM} = 0$. Moreover, this approach has very low performance impact. The only restriction on use of this approach is that the underlying hypervisor must allow reconfiguration from within a VM. Such features can be found in real-time hypervisors; for example, XtratuM allows changing modes (or reconfiguring the node) from system partitions.

Thales R&T evaluated temporal overload management by an LRM for the avionics use case due to confidentiality issues in providing the complete code of the avionics applications. This evaluation is presented in Chapter VI.

Under-utilization due to Deadline Overrun Management

As illustrated in Figure IV.F15a, the LRM suspends the best-effort applications running on core 1 to 3 as a MON of a critical application detected a potential deadline overrun. The LRM only resumes the best-effort applications once the critical application has completed its execution. However, this approach has a considerable impact on the QoS of the best-effort applications and the significant reduction in the utilization of the node's resources.

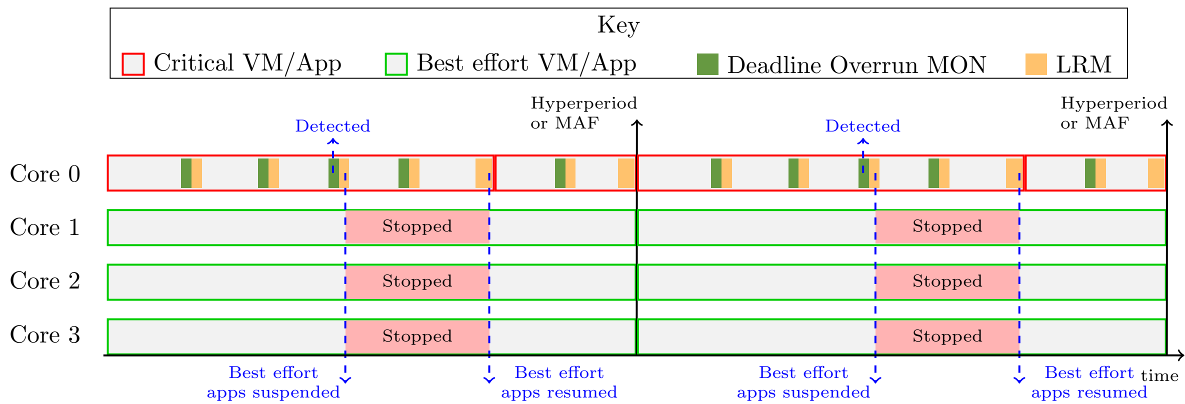


(a) *Dedicate core for node domain LRM*

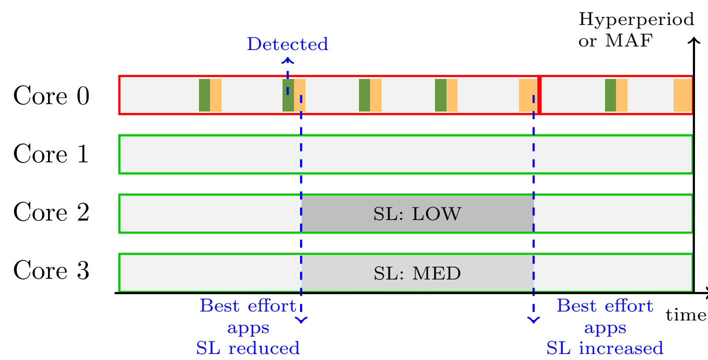
(b) *Node domain LRM at pre-define time slots*

(c) *Application/VM domain LRM*

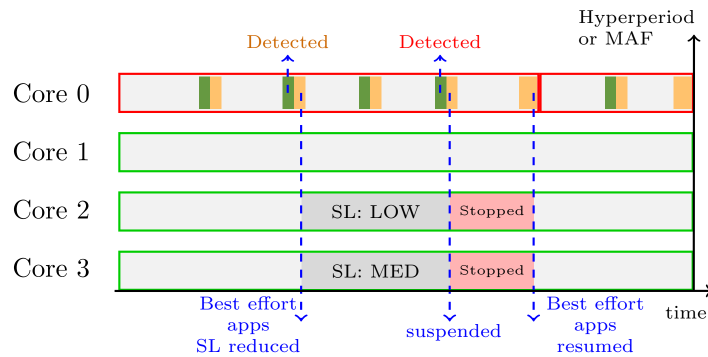
Figure IV.F14: *Deadline Overrun Management by LRM*



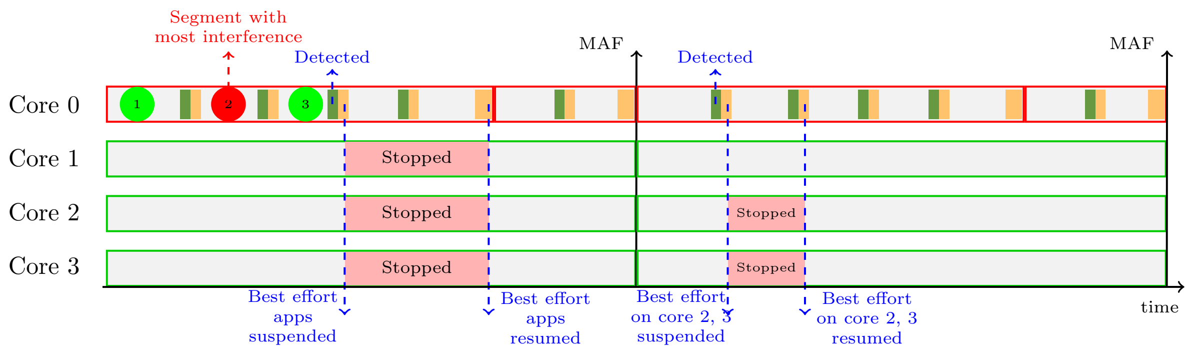
(a) *Under-utilization due to Deadline Overrun Management*



(b) *Solution 1 - Throttling SLs of Best-Effort Applications*



(c) *Handling Potential Deadline Overrun detection with QoS Management of Solution 1*



(d) *Solution 2 (by Thales R&T) - QoS Managment for Nodes with Cyclic (TT) Scheduling*

Figure IV.F15: *QoS Management by LRM*

Solution 1 We propose using hardware MON (Section IV.4.1) for best-effort tasks in addition to critical tasks. Moreover, our solution takes inspiration from the ACTORS project [22] for using Service Levels (SLs). Each best-effort application can have multiple SLs, and each SL corresponds to a specific resource requirement for the application and the QoS achievable by those resource requirements. For example, each best-effort application can have three SLs: $HIGH, MEDIUM, LOW$; SL $HIGH$ corresponds to high resource utilization and higher QoS achievable by the best-effort application. On the other hand, SL LOW corresponds to low resource utilization and lower QoS achievable by the best-effort application.

Suppose the LRM detects a potential deadline overrun (via MON). In that case, it looks into the hardware MON information (PMU data) of the best-effort applications to find out which application(s) are interfering with the critical application. The LRM reduces the SLs of best-effort applications having shared resource access above a certain threshold (Figure IV.F15b). The LRM determines the new SL of the best-effort application(s) based on the feedback from the hardware MON for the critical application. Only if the next deadline overrun MON instance still detects a potential deadline overrun, the LRM stops the best-effort applications (Figure IV.F15c). Listing C.L20 in Appendix C shows the modified parts of the deadline overrun MON to support this solution.

Solution 2 Thales R&T proposed another solution to this issue for nodes with cyclic (TT) schedulers such as those used in avionics. We summarize this approach here. This approach also uses the hardware MONs data and deadline overrun MONs data. Using the hardware MONs data of the best-effort applications, the LRMs can determine which best-effort application interfered with the critical application. Moreover, the LRMs can also determine the deadline MONsitoring instances between which the critical application encountered the most interference. For simplicity, we shall refer to the parts of critical applications between two deadlines overrun MONs instances as segments. For example, in Figure IV.F15d, base on the deadline MONs information from the first hyperperiod/MaF, the LRM determines that the critical application had the most interference in segment 2 (marked with a red circle in the figure). It also determines that the interference was caused by applications on core 2 and 3. Thus, in the next hyperperiod/MaF, the LRMs suspends the best effort applications on core 2 and 3 only during the second segment of the critical application. As a result, the LRMs improves the overall node utilization and the QoS of the best-effort applications. We can combine this approach with the first solution for further improvement. Thales R&T did a concrete implementation and evaluated this solution [7].

IV.10 Chapter Summary

This chapter explained the LRM of the resource management architecture. There are two main modular types of LRM sub-components:

1. *Local Resource Monitor (MON)*: Each resource or application managed by the LRM has one or more MONs, each providing different monitoring services, such as

availability monitoring or reliability monitoring. We designed and implemented the following new MONs:

- MON to interface with the hardware-specific monitoring features such as PMU and Intel MBM and CMT. We proposed methods to use these MON with XtratuM hypervisor and Linux/KVM to gather information about VMs/tasks.
- MON to detect permanent core failures on multicore platforms[6].
- MON for detecting potential deadline overrun by a critical VM in the presence of concurrently executing non-critical. VMs[6].
- MON that can use the XtratuM HM API to write partition status or errors (not handled by XtratuM) to a HM log.

2. *Local Resource Scheduler (LRS)*: Each resource managed by the resource management framework is paired with a LRS. Each LRS schedules the use of the resource and controls application access to the resource. The LRS services and the LRS implementation are specific to each resource. The LRSs can support offline or online scheduling. We designed and implemented the following new LRSs:
    - LRS for scheduling tasks of a critical application running in a XtratuM hypervisor partition[6].
    - LRS to assist the LRM in reconfiguring scheduling plan (modes) of XtratuM hypervisor[6].
    - LRS to provide an interface for the resource management to interact with Intel CAT and MBA hardware features.
    - LRS to support scheduling of TT tasks/VMs in Linux/KVM (with PREEMPT_RT patch).

An Local Resource Manager (LRM) provides various services, such as reading information from its MONs, calculating abstract state of resources, sending/receiving updates/orders to/from the GRM or a lower- or higher-domain LRM, making local reconfiguration decisions, translating orders to policies of its LRSs, and configuring the LRSs. In this chapter, we presented LRM policies[6] to manage the following:

1. Permanent failure of CPU cores.
2. Node-level temporal overload situation (by potential deadline overrun monitoring).

The implementation of the presented local resource management policies is use case-specific and explained in Chapter VI.

[6]Designed and implemented together with ONERA and Thales R&T.

V

Global Resource Management

"At the end of the day, the goals are simple: safety and security."
– Jodi Rell

This chapter presents the global resource management services provided by the GRM of our resource management framework. In addition, it explains the introduced security measures for the communication among a single central GRM and the LRMs. Furthermore, we propose distributing the GRM among a minimum number of different nodes. The GRM components on different nodes make the global resource management decisions via coordination amongst the distributed GRM components. We propose to implement this solution using a blockchain and achieve Byzantine fault-tolerance for the global resource management decisions and security for the communication amongst the distributed GRM components and the LRMs.

The GRM is located at the top of the resource management hierarchy and directly controls and supervises the LRMs in the next lower domain. These LRMs act as a granularity interface and hide fine-grained activities of the subsystem from the GRM's view. Consequently, the LRMs only send updates to the GRM when necessary, e.g., when an LRM changes a configuration of its domain. In addition, when a reconfiguration in an LRM's domain is not sufficient to meet the long-term changes, the LRM requests the GRM for reconfiguration in the system domain. Henceforth, we refer to reconfiguration in the system domain as global reconfiguration.

The GRM provides the following generic services:

Gather Updates from LRMs

The GRM obtains information about the current status of the subsystem from LRMs next in the hierarchy. This LRM can be present in the cluster-domain or the node-domain. If an LRM of a subsystem can handle a change locally, the LRM only notifies the GRM of a local reconfiguration. It does not send any detailed information to the GRM. In the case an LRM requires a global reconfiguration, it sends essential information to the GRM. This information can include details such as type(s) of the resource(s), the abstracted value(s) of the monitored resource(s), the failure(s), and parameters of application(s) that need global redeployment.

Global Reconfiguration Decision-Making

The GRM has knowledge of the complete system. The knowledge can be in abstract or absolute information regarding the system components and applications. For example, the GRM has information on the applications in the system and their criticality levels, all pre-computed offline configurations of the system, system-wide global constraints, abstract resource and application QoS levels, current abstract state of the system and subsystems (received via the LRMs), and abstracted information about the availability of resources in the system. By knowing such information, the GRM can make global reconfiguration decisions when necessary. The GRM analyzes the abstracted information from the LRM at the system level and takes reconfiguration decisions that allow the system to adapt to different conditions. The GRM considers information from all highest domain LRMs to make a global reconfiguration decision that satisfies system-wide constraints.

Obtain or Compute a New Global Configuration

The GRM has access to the database of all offline pre-computed configurations provided by offline scheduling tools. The GRM can store these configurations in a distributed or centralized way. It selects one of these configurations from the database while making a global reconfiguration decision. It transmits these decisions (as orders) to the LRMs next in the hierarchy via the network (and the middleware).

Alternatively, the GRM can also compute new configurations at runtime by determining new scheduling tables or changing scheduling parameters. However, online computation of configuration is not feasible for all use cases due to large (time) overheads.

Send Orders to LRMs

Once the GRM has taken a global reconfiguration decision, it communicates the information to the LRMs that is next in the hierarchy and involved in the reconfiguration decision. These LRMs must further propagate the reconfiguration orders to the relevant lower domain LRMs. For example, suppose the GRM decides on a global reconfiguration that requires reconfiguring two nodes in different clusters. The GRM only sends orders to the cluster domain LRMs involved in the reconfiguration. These cluster domain LRMs distribute appropriate orders to the node domain LRMs of the nodes involved in the reconfiguration. If needed, the node domain LRMs may, in turn, send orders to the virtualization domain or Application domain LRMs that are below them. Orders can be in the form of a simple reference to the actual configuration, or the GRM could transmit the complete configuration via the network.

Manage External Input

As explained in Section III.7, the GRM also provides a service to manage an external input. Such input could be given locally (I/O peripheral directly connected to the GRM node) or remotely (via off-system Ethernet). A user can use this input to provide new application(s) or constraint(s) to the system. The GRM can determine a new configuration that satisfies the constraint or allocates the required resources to the application(s). The input can also be an absolute reconfiguration decision. The GRM can trigger a global reconfiguration based on this input. Similarly, the GRM can also trigger mode changes in the system based on the external input.

Figure V.F1 summarizes all the services.

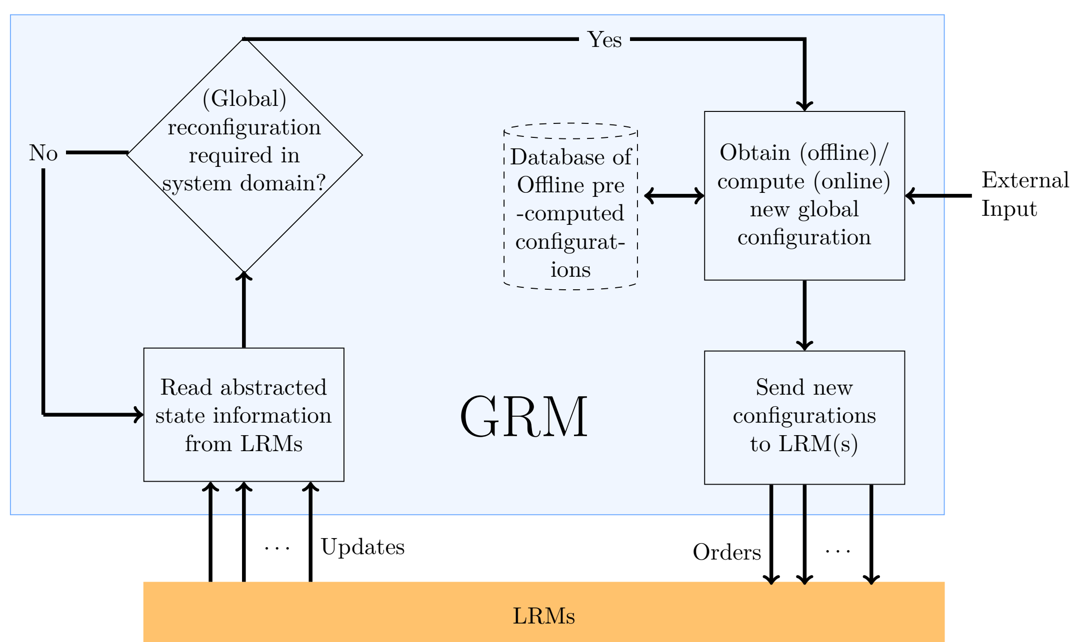
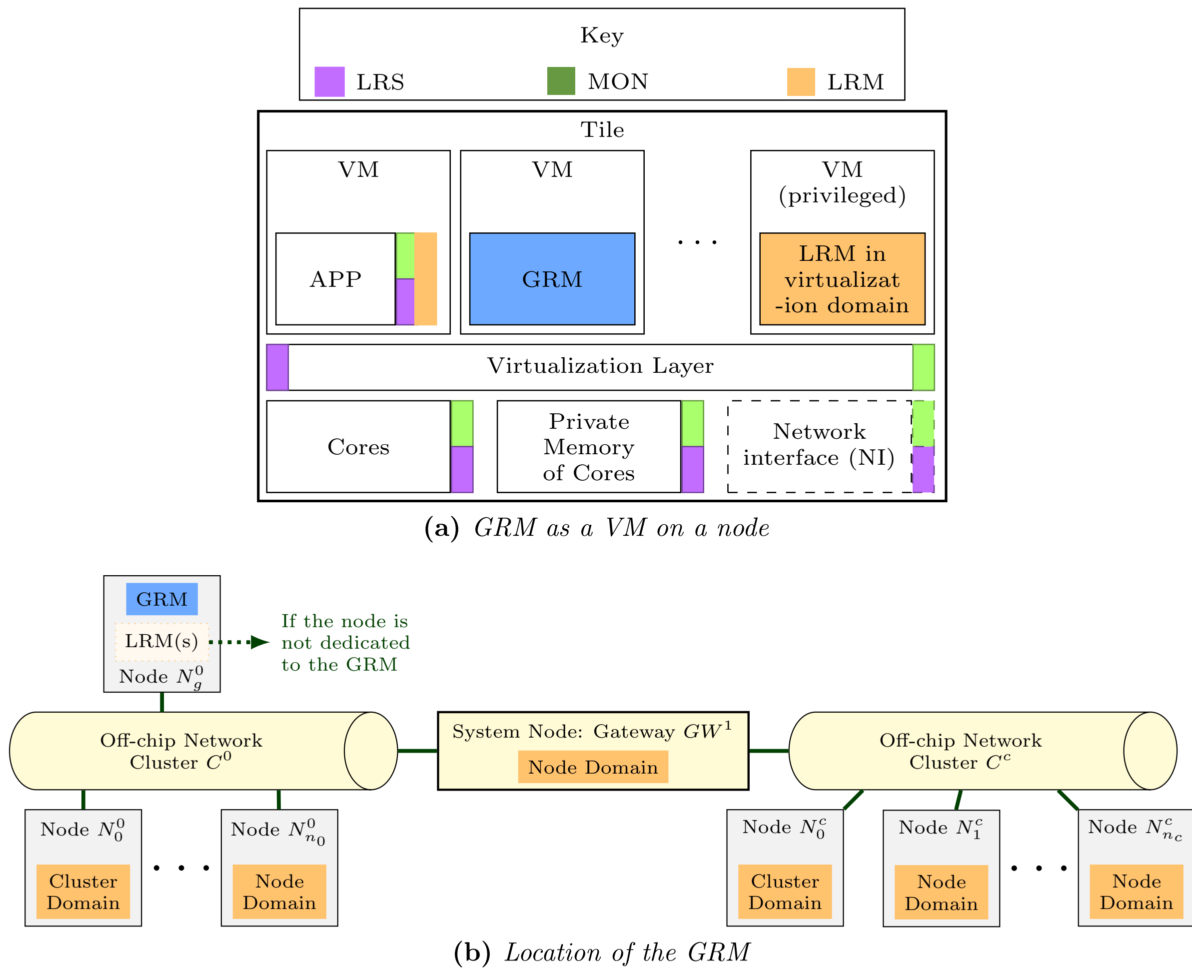


Figure V.F1: *Global Resource Manager (GRM)*

V.1 GRM Implementation Options

There are several options to implement the GRM. The first option is to implement it in software in a separate dedicated node. On the one hand, this will make the GRM's development and the reconfiguration and optimization logic easier. On the other hand, there will be some costs associated with dedicating a node exclusively to the GRM. Another option is integrating the global resource manager into an existing node in the system, e.g., assigning it to its VM. As a result, the overall cost for implementing the GRM will reduce, and communication with some LRMs, such as the LRM present on the same node, will be faster and have less overhead. However, it will make the development of GRM harder and decrease its flexibility as it will be constraint by other applications



(a) *GRM as a VM on a node*

(b) *Location of the GRM*

Figure V.F2: *Implementation Options for the GRM*

running in that node. Figure V.F2 illustrates this option. Since the GRM only decides on the global reconfiguration and the LRMs enforce the GRM orders, the GRM VM does not need special privileges. Finally, a hardware implementation is possible, but it would be too costly and not beneficial for implementation on most COTS platforms.

Conceptually, one GRM exists in the system, although distribution is possible for fault tolerance and scalability. Therefore, the GRM can be realized either by a single node or a set of nodes. Section V.6 explains the possibility of distributing the GRM among a minimum number of different nodes and making global reconfiguration decisions via distributed coordination. However, for simplicity of explanation, we will refer to the GRM as a single entity for now.

V.2 Global Resource Management Communication

The GRM and the LRMs communicate with each other to exchange resource management information:

1. The LRMs send abstracted resource updates (or abstracted subsystem status) to the GRM or next higher domain LRM. They can also send a reconfiguration request.

2. The GRM or the LRMs in the higher domain send reconfiguration orders to the LRMs directly below them.

3. For membership and failure detection, LRMs can send messages, such as a heartbeat (periodic message), to the GRM.

Thus, we propose three conceptual communication channels between the GRM and the LRMs (and between higher and lower domain LRMs): Update, Order, and Membership channel, as shown in Figure V.F3.

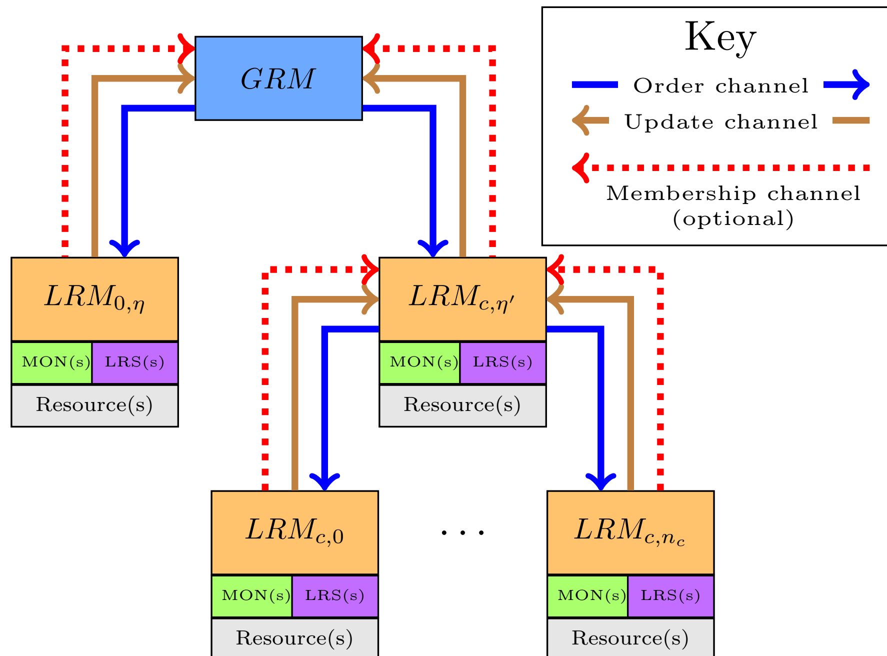


Figure V.F3: *Conceptual Resource Management Communication Channels*

V.2.1 Update Channel

The Update channel has an LRM as a source and GRM (or a higher domain LRM) as the destination. An LRM uses this channel to send status updates and global reconfiguration requests to the GRM (or a higher domain LRM). Each message on this channel consists of:

- an abstracted resource state (or state of the LRM's node), and
- the type of update, i.e., if the message is a reconfiguration request or just a status update.

The GRM (or a higher domain LRM) must receive all messages sent by the LRM on the Update channel and in the correct order so the higher domain resource manager can correctly make a reconfiguration decision.

V.2.2 Order Channel

The Order channel has GRM (or a higher domain LRM) as a source and an LRM as a destination. The GRM (or a higher domain LRM) uses this channel to send reconfiguration orders to an LRM. Each message on this channel consists of

- a new (abstract) reconfiguration that the destination LRM must apply in its subsystem, and
- the time at which the LRM must apply the new reconfiguration: immediately or at the end of the hyperperiod/MaF (deferred).

The destination LRM must receive the messages on this channel in the correct order. However, the destination LRM must only consider the last received message sent by the higher domain resource manager to ensure it applies only to the latest received order. The GRM ensures that any new orders either encompass the older orders or completely substitute them.

V.2.3 Membership Channel

The membership channel has LRM as a source and GRM (or a higher domain LRM) as the destination. An LRM uses this channel to send heartbeat messages to the GRM for membership and failure detection purposes. We chose heartbeat-based membership because of simplicity and low overheads. A heartbeat message is an empty message to indicate the operational status. Each LRM_λ periodically sends a heartbeat to the GRM (or a higher domain LRM) with a T_λ. If the higher domain resource manager does not receive the expected heartbeat once every T_λ, it declares LRM_λ as dead.

In a static system, such non-responsiveness from LRM can be simply from an LRM failure or failure of the respective subsystem; for example, a node domain-LRM can become non-responsive if the LRM's node fails or the network connecting the LRM's node to the rest of the system fails. The GRM cannot rely on this subsystem (or node) to ensure the execution of safety-critical applications. Thus, the GRM must trigger a reconfiguration.

If LRM_λ sends periodic updates to the higher domain resource manager, sending one update at least every T_λ, then the Update channel messages also serve as the Membership messages. Such cases do not require the use of an explicit Membership channel.

V.3 Global Resource Management: Core failure Scenario Example

This section presents a simple example of global resource management for a core failure scenario. This example only aims to give a basic insight into the functioning of the GRM.

We provide a concrete implementation of the GRM for the avionics use case in Chapter VI.

Let us consider a simple system structure with three nodes N_1, N_2, N_3 (Figure V.F4a). Node N_1 and N_2 have a dual-core processor and can host critical and non-critical applications. The nodes have a simple LRS that assigns only a CPU core to an application. Thus, we can execute only one application per core. In addition, LRM and core-failure MON run on each core periodically (as explained in Section IV.9.1). Initially, Node N_1 runs a critical application on core 0 and a best-effort application (BE1) on core 1. (Figure V.F4b), while Node N_2 runs only a best-effort application (BE2) on core 0 (Figure V.F4c). N_3 hosts only the GRM. The GRM expect messages from LRM_1 and LRM_2 with a maximum interval of T_1 and T_2. If a message does not arrive from an LRM within the given period, it assumes the corresponding node to be dead. LRM_1 and LRM_2 send either of the following messages for this purpose:

1. If there was no local reconfiguration since the last sent message, the LRMs send an (empty) message to the GRM via the membership. channel.

2. LRMs send an update message to the GRM via the update channel if a local reconfiguration occurred since the last sent message.

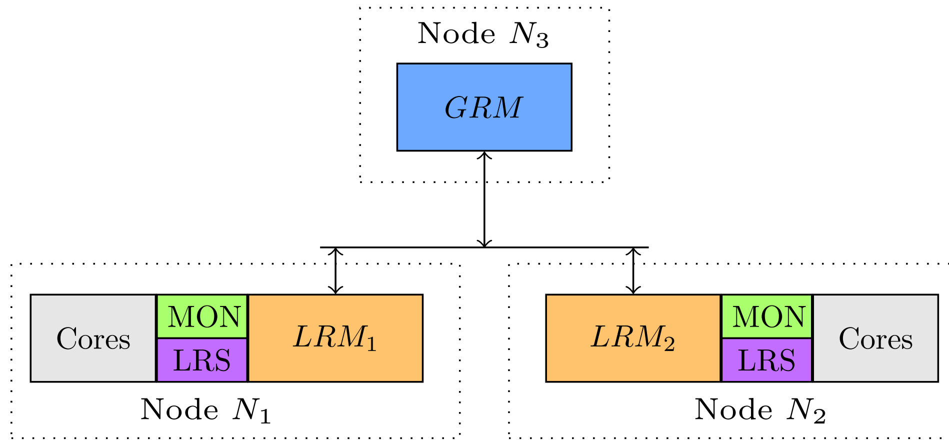
On node N_1, the LRM detects that core 0 has failed at t_0. Since this core was hosting a critical task, the LRM orders its LRS to stop executing BE1 on core 1 and orders the LRS to assign core 1 to the critical application to ensure the least disruption of critical services. Finally, the LRM informs the GRM of the local reconfiguration and requests for a global reconfiguration.

Upon receiving the message, the GRM looks for a node capable of hosting BE1. The GRM can achieve this by simply keeping track of the CPU utilization and availability of each node or by using fixed offline configurations (as explained in the avionics use case of Chapter VI). In this simple example, the GRM sends an order to LRM_2 to host this application. At t_1, LRM_2 receives the order from the GRM to host BE1 (Figure V.F4b). The LRM instructs its LRS to host BE1. The LRS, in turn, hosts BE1 on the idling core 1.

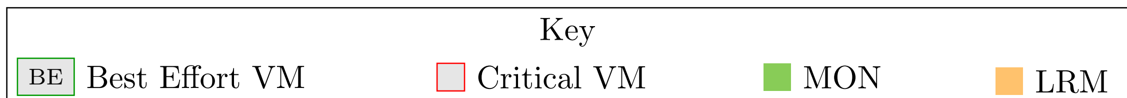
Let us assume, node N_1 fails complete. Thus, the GRM stops receiving updates or membership messages from LRM_1. After a period of T_1, with no messages from LRM_1, the GRM considers the node to be dead and sends an order to LRM_2 for hosting the critical application (Figure V.F4b).

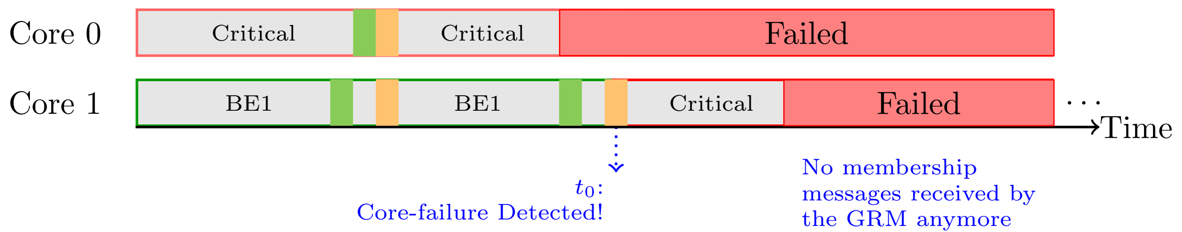
At t_2, LRM_2 receives the order from the GRM to host the Critical application. The LRM instructs its LRS to host BE1. Since there are only two cores available, the LRS picks the application according to the priority $Critical > BE1 > BE2$ assigned by the system designer. Thus, the LRS discards BE2 and executes only the critical application and BE1. The LRM informs the GRM of the local reconfiguration via an update message. However, the GRM cannot perform any further global reconfiguration due to the limitation of resources. Thus, BE2 is completely discarded.

This example considers a simple scenario with the most basic LRS. It is only meant to give an idea about the functioning of global resource management upon core failure.

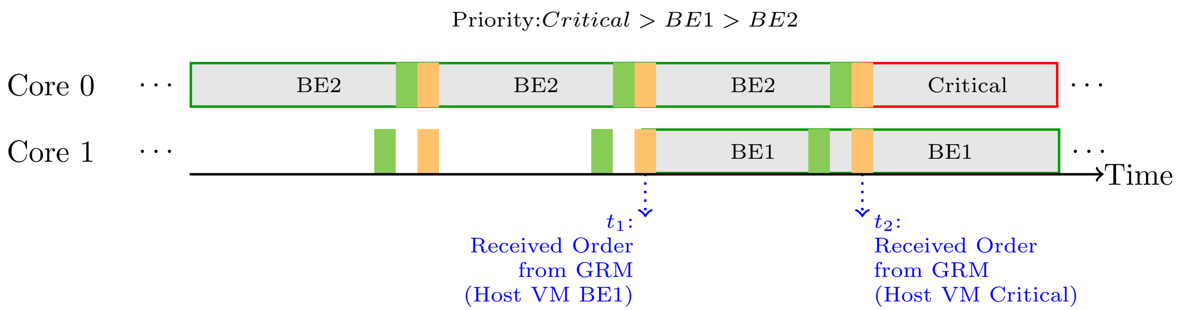


(a) *System Structure*





(b) *Node N_1*



(c) *Node N_2*

Figure V.F4: *Global Resource Management - Core failure Scenario*

Note that the global resource management only takes place if enough resources are not available locally to ensure continuity services. If enough resources are available locally, LRM performs only a local reconfiguration and informs the GRM via an update message (without a reconfiguration request). If a node does not have enough resources to ensure continuity of services, then the LRS allocates resources to applications in order of their priority defined by the system designers. By default, the resource management framework prioritizes critical applications over non-critical applications. However, system designers can pass the priority as input to the resource management via the resource management configuration file.

The global reconfiguration delay after an LRM detects a core failure depends on the relationship between the execution time of the GRM and the LRMs, and the network overhead. Suppose we can ensure that the GRM and LRM execute at fixed time instances, for example, by using TT network protocols. In this case, we can provide an upper bound to the global reconfiguration delay.

We provide a concrete implementation of the GRM (using offline computed scheduling tables) for the avionics use case in Chapter VI. The implementation also provides an upper bound on the global reconfiguration delay.

V.4 Resource Management Security

Unsecured resource management has several weak spots that an attacker can exploit. Therefore, together with the security experts from the University of Siegen, we analyzed the resource management communication, LRM, and GRM from a security viewpoint.

Attacking specific resource management components, attackers can masquerade as a GRM or an LRM, allowing them to decide on the local or global (re-)configuration. As part of a spoofing attack, an attacker can successfully identify as a resource manager and perform malicious actions or gain sensitive system information. By sending erroneous orders to the lower domain resource managers, the attackers can influence one or more subsystems. For example, an attacker masquerading as a GRM can send false global reconfiguration messages. Vice versa is true with lower-level resource managers. An attacker masquerading as a lower domain resource manager can send incorrect updates and reconfiguration requests or incorrectly configure the LRSs. Such erroneous updates can cause the higher-level resource managers to make unnecessary reconfiguration decisions. For example, as an LRM, the attacker could locally reconfigure a subsystem and send incorrect status updates to the higher domain resource manager.

In a sniffing attack, an attacker can intercept the resource management communication by capturing the network traffic and using a packet sniffer. With such an attack, the attacker can gain sensitive information about the system's behavior. In addition, an attacker can secretly relay and possibly alter the resource management messages by performing a Man-in-the-middle on the communication network. Such an attack is comparable to active eavesdropping, where a malicious entity connects multiple victims independently and relays messages between them. The victims assume that their connection is private, but the whole communication is, in fact, under the attacker's control.

An attacker can also suppress the resource management messages. For example, suppose the attacker suppresses the update messages from an LRM. In that case, the higher domain resource manager will never find out if the lower domain LRM sends a reconfiguration request or has a change in resource availability. Similarly, if the attacker suppresses messages on the membership channel, the higher domain resource manager will trigger a reconfiguration as it assumes that the subsystem (or node) is dead. On the other hand, if the attacker suppresses the orders from a higher domain resource manager, then the entire system can be jeopardized in the worst case.

In a packet injection attack, an attacker can send bogus messages to the resource managers. The attacker makes the messages look like they are part of the regular resource management communication, for example, by mimicking order or update messages. In a replay attack, an attacker fraudulently sends the captured (valid) resource management messages repeatedly. Although the message is valid, it is outdated. The GRM or the LRM receiving outdated messages may not operate as intended, e.g., the GRM can select inappropriate global configuration based on old messages.

V.4.1 Security Services

We can prevent these attacks using the following security services for resource management communication:

1. Confidentiality to ensure the privacy of information.
2. Integrity to ensure that data is not modified.
3. Authenticity to ensure that data is genuine and that the actual origin of the data is the same as the claimed origin.
4. Access Control to allow access based on permissions.

We provide these four security services by the cryptographic mechanisms.

- Encrypting message provides confidentiality and protects against the sniffing or eavesdropping attacks described above.
- For ensuring integrity, authenticity, and access control, we use authentication codes. By using authentication codes, we ensure:
  - The resource management messages cannot be modified unnoticeably (integrity).
  - The resource management messages are genuine, and the message originates from the resource manager who claims it (authenticity), i.e., the resource manager is part of the trusted resource management group.
  - The resource manager has permission to use the service (access control).

We protect resource managers against attacks by combining authenticity with access control. Without knowing the correct cryptographic key, the attacker cannot act as a credible resource manager since the attacker cannot decrypt existing resource management messages nor send legitimate messages to other resource managers. Furthermore, any message from the attacker received by a valid resource manager will fail authentication. Thus, attackers cannot masquerade as resource managers. The same prevents man-in-the-middle, spoofing, and packet injection attacks targeting the resource management communication. We prevent replay attacks by adding a time-varying parameter in the message authentication code protected message. The attacker cannot modify the time-varying parameter, and the receiver can check the time-varying parameter of the received message to detect a replayed message. If the attacker modifies the time-varying parameter, the message will fail integrity check.

Another possibility is to use digital signatures that the resource managers can verify for authenticity and access control. However, to keep the security overheads low, currently, we do not consider this mechanism.

V.4.2 Security Levels

We categorize the resource management security services into three levels, as shown in Table V.T1. Level 1 security service provides integrity to prevent the manipulation of resource management messages and authenticity to verify the source of the messages and ensure that the received messages originated from a trusted resource manager. Level 2 provides confidentiality in addition to Level 1. In addition, levels 1 and 2 both ensure protection against replay attacks by using a time-varying code (both levels provide integrity service to check the time-varying code). Lastly, Level 0 provides no security. These security levels are helpful depending on the amount of security required for different use cases. Moreover, different types of resource management communication channels can use different security levels as explained in Section V.4.3.

Table V.T1: *Resource Management Security Levels*

Security Level	Security Service
0	No security
1	Integrity and Authenticity
2	Integrity, Authenticity and Confidentiality

We can implement other security levels that provide a different combination of security services than the two levels considered in the table. However, the other combinations are not desirable. For example, using confidentiality without integrity is not meaningful. Moreover, we were informed by security experts that most security algorithms for confidentiality automatically provide authenticity. In the security domain, this is referred to as Authenticated Encryption (AE) and Authenticated Encryption with Associated Data (AEAD). Hence, we concluded that an additional level for the combination of authenticity and confidentiality is not essential. Access control is not listed in the table

because it is independent of the described security levels. However, it is used in parallel to the security levels.

V.4.3 Security Levels for Resource Management Communication Channels

This section explains which security levels our resource management framework assigns (by default) to each resource management communication channel.

Update Channel

An attacker targeting the Update channel can get information about the status of nodes and resources. Moreover, the attacker can send manipulated or fraudulent updates and reconfiguration requests, as explained earlier. Thus, we need to provide an integrity service on this channel to protect against the manipulation of messages. A higher domain resource manager must verify the origin of the update message and ensure a trusted resource manager sent it. Hence, we need to implement an authenticity service on this channel additionally. Moreover, the messages may contain sensitive system information, with which an attacker can gain an undesirable advantage. Thus, to protect the data, we need a confidentiality service on this channel. As a result, we assign security Level 2 to the updates channel. By default, the resource management framework will use Level 2. However, if the system designers deem it unnecessary for the use case, they can reduce the security to Level 1 or altogether disable it (Level 0) via the resource management configuration file (Listing C.L21).

Order Channel

It is vital to ensure that an attacker cannot manipulate or fake the messages on this channel as they contain reconfiguration orders. Moreover, reconfiguration orders contain essential system information. Hence, similar to the Update channel, we need to provide all security services resulting in security Level 2 for the Order channel. Thus, the resource management framework will select Level 2 by default here as well. However, the system designers can reduce the security to Level 1 or altogether disable it (Level 0) via the resource management configuration file (Listing C.L21).

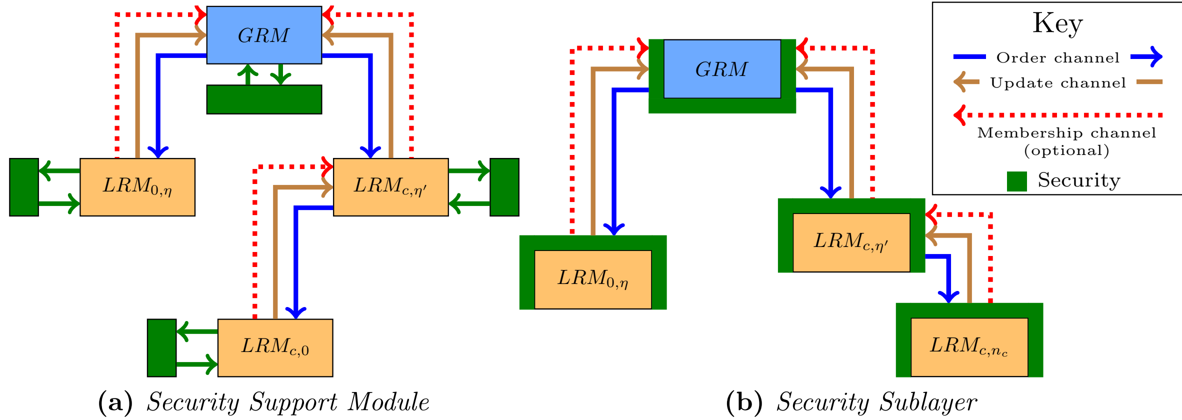
Membership Channel

We need only to ensure that the messages are authentic and unchanged for the Membership channel because the heartbeat messages usually do not hold any confidential data. Hence, we need to provide Security Level 1 (integrity and authenticity) on this channel by default. Nevertheless, the system designers can increase it to Level 2 or disable security via the resource management configuration file (Listing C.L21).

Table V.T2 summarizes the default security levels used in the resource management framework for the communication channels.

Table V.T2: *Default Security Levels for Resource Management Communication Channels*

Channel	Security Level
Update	2
Order	2
Membership	1



(a) *Security Support Module* **(b)** *Security Sublayer*

Figure V.F5: *Implementation of Resource Management Security Services*

V.4.4 Implementation of Security Services

There are two options to implement the security services:

1. Implementation as a support module (Figure V.F5a)
2. Implementation as a layer between the resource managers and the underlying hardware or software (hypervisor or OS) (Figure V.F5b)

The two implementations are different only in terms of the handling of the communication. In an implementation that contains the security as a separate support module, the resource managers must handle the secure communication, i.e., each resource manager acts as a source and destination for the messages. A resource manager must first send a message to the security module. The security module performs encryption and adds security envelop around the message before sending it back to the resource manager. The resource manager must now send this secure message on a resource management channel. Similarly, the resource managers must send every received message to the security module to avail the security services. The security module performs the required checks and decryption on the message, and if successful, it sends the essential data back to the resource manager. If the checks fail, it indicates the failure to the resource manager. In such a case, the resource manager discards the message.

In an implementation containing security as a sublayer, the sublayer handles secure communication, i.e., the sublayer acts as the source and destination of the messages. A resource manager needs only to send the messages to the security sublayer and

indicate the resource management channel type for the message. The security sublayer performs encryption and adds security envelop around the message before sending it to the appropriate resource management channel. It receives messages from all channels and performs the reverse actions; if successful, it sends the essential data back to the resource manager; otherwise, it discards the message. At the end, both approaches ensure end-to-end security for resource management communication. However, the main benefit of the second option is the direct communication between security sublayers and transparency for the resource managers. Hence, we chose this approach in the implementation for the avionics use case (Chapter VI).

Selection of Security Algorithms

Based on suggestion from the security experts at the University of Siegen, we decided to implement two options for algorithms in the security library of the resource management framework (Section III.9):

1. ChaCha20-Poly1305 [197, 198]
2. CLEFIA [199] (in Offset CodeBook (OCB) operation mode)

Both these options are symmetric key algorithms of different types for use in the security sublayer. Koller [200] explains the reasoning behind selection of symmetric key algorithms for our resource management framework from a security viewpoint. He also explains both algorithms in detail and the required key management. We do not cover these points here as they are out of the scope of this dissertation.

From a resource management viewpoint, we need the algorithms to have low overheads and applicability to all the system nodes. ChaCha20 is a low overhead stream cipher and a derivative of an algorithm that is part of the eSTREAM portfolio [201]. CLEFIA is a standardized lightweight block cipher algorithm. It is optimized for use in embedded systems and has a low CPU, and memory overhead [202]. Both algorithms have the same end goal of securing the resource management messages. In simple terms, a block cipher is a symmetric key algorithm that encrypts and decrypts data in fixed-size blocks, while a stream cipher does the same bit-by-bit. Stream ciphers are less resource-demanding and have faster processing speed than block ciphers. However, this does not necessarily mean that stream ciphers are better suited for resource management. Most trusted ciphers are those that are widely used in many applications and tested by cryptographers worldwide. As a result, the vulnerabilities of these ciphers are known and often fixed. The main advantage of block ciphers lies here. Block ciphers are used nearly everywhere, while stream ciphers are used only in few applications. Hence, block ciphers are more trusted.

We also considered other standardized algorithms but did not use them due to higher overheads or usage complexity. For example, PRESENT algorithm [199] requires bit-oriented permutations that have high overhead in software implementation [203]. Compared to a software implementation of AES-128-GCM [204], both the selected algorithms are faster when we do not consider dedicated hardware extensions for security [205, 206]; for example, the ChaCha20-Poly1305 construction is faster by a factor of 3, and PRESENT is faster by a factor of 2.5. On the other hand, when we consider

hardware extensions, AES-128-GCM is undoubtedly faster [205]. Similarly, it is possible to have a faster implementation of PRESENT in hardware consisting of bit-permutations with simple wiring. However, since our system structure consists of COTS heterogeneous hardware nodes, using a dedicated hardware extension or hardware implementation is not desirable as all nodes may not support the extension.

Depending on the permissible overheads and trust required in the security algorithm, a system designer can choose one of the two implemented algorithms via the resource management configuration file (Listing C.L21). Moreover, we designed the security sublayer to be modular, i.e., a system designer has an option to plug in a new security sublayer algorithm if the two implemented ones do not suffice.

In the avionics use case (Chapter VI), we provide a concrete implementation for the secure resource management communication channels. Moreover, we experimentally evaluate the overheads for sending and receiving resource management messages over different channels while using both the implemented algorithms (one at a time) in the security sublayer.

Limitations of Security Sublayer

Using security only at a sublayer level is not sufficient for all use cases. As explained by Koller [200], we require additional security strengthening measures at cluster-level, i.e., in Layer 2 devices, such as network interface cards and switches. He proposed Media Access Control security (MACsec) layer 2 protocol for the cluster-level.

We consider that the attackers do not have physical access to the hardware platform. If the attackers get physical access to the platform, they can potentially read stored keys and masquerade as resource managers. Furthermore, attackers can potentially read and manipulate all incoming and outgoing messages if attackers install malicious software between the resource management and the security sublayer. The security sublayer is not helpful if malicious code is present in the binary files of the resource managers, i.e., from the system development phase or added as part of a resource management update.

V.5 Using Existing Protocols For Resource Management Communication

There exists a variety of open-source protocols and services, such as Message Queue Telemetry Transport (MQTT) and Open Platform Communications Unified Architecture (OPC UA), for communication in distributed systems. Apart from using the resource management communication with security sublayer as explained in the previous sections, we can use these existing protocols, especially in Linux-based use cases.

V.5.1 MQTT and OPC UA

In this section, we provide an overview about using two popular existing protocols, MQTT and OPC UA, for resource management communication, taking into consideration requirements such as membership, security, and reliable message delivery. MQTT is the

most popular communication protocol for IoT, while OPC UA is gaining popularity in industries. MQTT is a lightweight protocol designed for low-resource and low-power nodes in low-bandwidth and high latency networks. On the other hand, OPC UA is a platform-independent standard for industrial automation to unify communication between the devices of different manufacturers.

OPC UA supports a request/response, and a publish/subscribe communication model, while MQTT uses only the publish/subscribe communication model. In the request/response model, a client node requests data or services, and a server node responds by providing the data or service. Such a model is useful when data is needed on demand.

In publish/subscribe model, a central node (called a broker) receives and distributes the data. The clients are nodes that interact with the broker, either by publishing messages or subscribing to a topic. i.e., when a client node has new data, it publishes the data to the broker node. Topics are the routes where the broker publishes data. Thus, client nodes that subscribe to a topic automatically receive data from the broker when new data is available. The broker moves the data from publishers to subscribers without storing it. Thus, the publish/subscribe model allows for mass data distribution efficiently and reduces network traffic by up to half compared to the request/response model. Publish/subscribe is useful when data is not updated continuously (i.e., we transmit discrete or abstracted values with well-defined intervals). Data does not have to be updated on demand, and already published data can be used.

MQTT allows publisher nodes to send messages to a broker. The broker, in turn, forwards the messages to the subscriber nodes. MQTT messages can have a variety of data formats for the payload, such as JSON, XML, encrypted binary, or Base64. However, the target client should have the ability to read the payload data. The MQTT clients and broker exchange messages over TCP connection (unsecure) or an encrypted Transport Layer Security (TLS) connection (secure).

MQTT *keepalive* ensures a broker and client connection is still open (for a predefined time interval in seconds) and that both of them are aware of being connected. As far as client and broker exchange messages frequently and do not exceed the keepalive interval, an extra heartbeat message is not required for membership purposes.

MQTT offers three possible QoS levels:

- *At most once delivery* – The message is sent only once, and lost if the subscriber node is not available.
- *At least once* – The publisher node temporarily stores the message and resends it if and until an acknowledgment is not received from the broker within a specified amount of time. However, a subscriber node may receive the same message more than once.
- *Exactly once* – The subscriber node receives the message exactly once. However, this form of message delivery has higher overheads due to more handshakes.

An OPC UA client may communicate with multiple servers, and a server can communicate with multiple clients. Moreover, a server acts as a client for communication with other servers. OPC UA allows on-demand access permission-based read or write of

data. It also allows nodes to subscribe to data and be notified when data changes based on specific criteria. OPC UA servers define the services provided by them to clients, possibility to be dynamically discovered by clients and data-types defined in by protocol, such as bool, Int32, ByteString, and XmlElement.

OPC UA implements security mechanisms for authenticating clients and servers and checking the integrity of communications. Moreover, it provides security (confidentiality) at the transport level by encryption and signing messages.

OPC UA uses heartbeat for ensuring the liveness of the client and the server. To ensure reliable delivery of messages, a OPC UA server must stores collected data and events in a non-volatile memory until the client confirms reception (method *SetSubscriptionDurable* defined in OPC 10000-5 [207]).

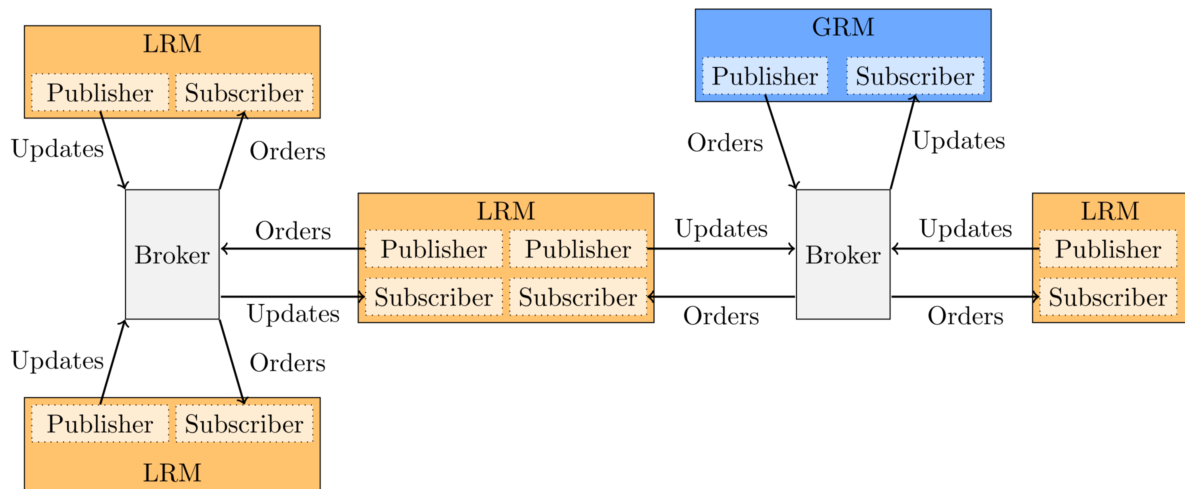
Table V.T3 summarizes the relevant properties of these protocols.

Table V.T3: *Existing Communication Protocols for Resource Management*

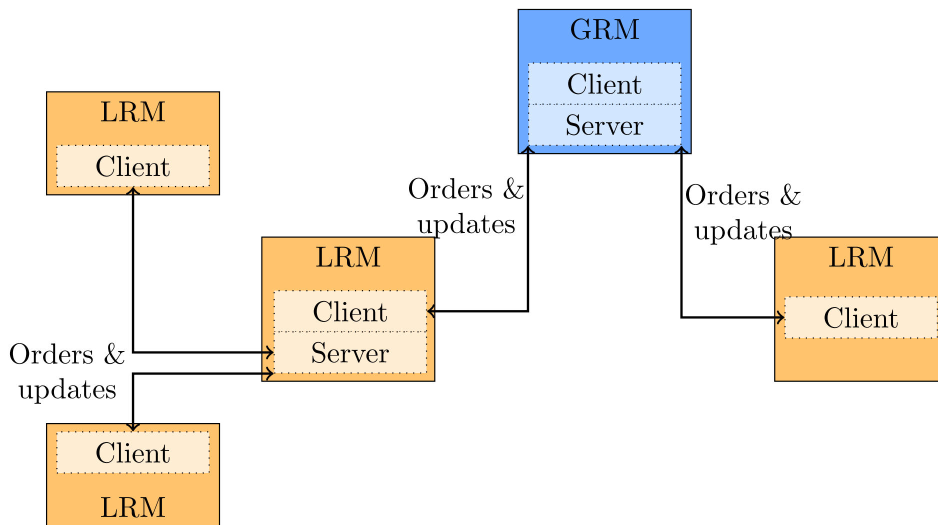
Property	OPC UA[208]	MQTT[209]
Security	Yes	Yes (with TLS version)
Membership	Bidirectional Heartbeat	Limited via MQTT Keep Alive
Communication model	Request/ Response or Publish/ Subscribe	Publish/ Subscribe
Reliable delivery	Yes	Yes
Existing Implementation	C, C++, Java, Python	C, C++, Java, Python

Resource Management Communication with MQTT

MQTT broker(s) running on node(s) facilitates communication between the LRMs and the GRM. In a flat resource management architecture, one broker is sufficient. On the other hand, the hierarchical architecture (Figure V.F6a) requires multiple strategically placed brokers (for example, one for each layer of the hierarchy) so that it is still possible to tackle heterogeneous nodes, complex system structures, and ensure scalability. If only one broker is present to deal with all levels of the hierarchy, then it may defeat the purpose of the hierarchical architecture by acting as a bottleneck. We can implement the broker on the same node with a LRM or a GRM or on a completely separate node to improve performance. The higher domain resource managers (e.g., GRM) subscribe to updates from the (lower domain) LRMs and publish orders. Contrarily, the (lower domain) resource managers subscribe to orders from the (higher domain) resource manager and publish updates.



(a) *Resource Management Communication with MQTT*



(b) *Resource Management Communication with OPC UA*

Figure V.F6: *Resource Management Communication using Existing Communication Protocols*

Resource Management Communication with OPC UA

An approach similar to MQTT is possible for OPC UA as well. However, we can simplify the implementation as shown in Figure V.F6b.

Discussion

Previous work (such as [210] and [211]) have examined the overhead of using MQTT and OPC UA. As expected, MQTT (with TLS) has much lower CPU overhead, memory overhead, and network bandwidth consumption as compared to OPC UA. MQTT requires data field coding, and its content is use case dependent. OPC UA has much wider use. It is a full architecture where the communication protocol is only one part of the architecture. OPC UA provides additional features such as node discovery, data structures, methods, and two different communication models to access nodes of a distributed system with services that allow reading, writing, and calling methods of the OPC UA servers.

We suggest using MQTT for resource management communication in conditions with low-resource and low-power nodes using low-bandwidth and high latency networks. Moreover, MQTT give more flexibility for use in specific applications. On the other hand, OPC UA based resource management communication will benefit industrial environments, where the unification of industrial devices from a different manufacturer is essential, and overhead or network bandwidth consumption is not so critical as compared to the advantages of OPC UA. We leave the actual experimental evaluation of resource management communication via OPC UA and MQTT to future work.

V.6 Distributed Decision-Making for Global Resource Management

In the previous section, we took the first step towards resource management security by securing the communication between the LRMs and the GRM. In addition, we proposed to add fault tolerance for the LRMs against cores failures by replicating LRMs on multiple cores of a node and executing them synchronously (Section IV.9.1). Moreover, it is possible to use NMR with the replicated LRMs. Furthermore, LRMs send heartbeat messages to the GRM on the membership channels so that GRM can detect if an LRM's node is dead.

Until now, we considered the GRM as a single central entity that is present on a dedicated or shared node in the system as shown in Figure V.F2 of Section V.1. This single central entity participates in resource management to enable system-wide adaptability in situations such as resource failures. Such a GRM is easier to implement and has a low resource overhead. Besides, resource management makes consistent global decisions without the risk of double-spending system-wide resources as the GRM is a single entity controlling the entire system.

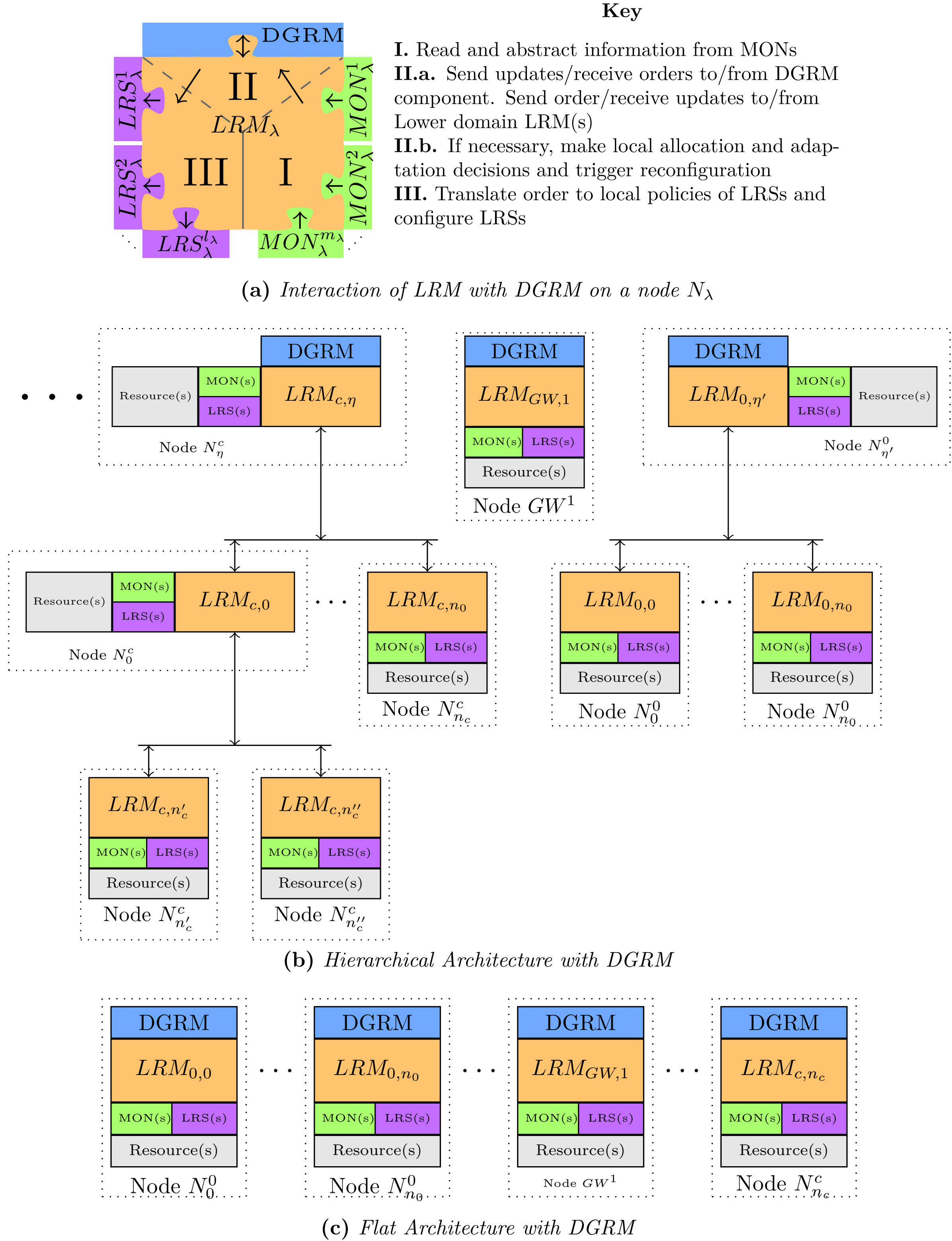
Nevertheless, a central GRM has limited scalability as it becomes a bottleneck for resource management when it needs to keep up with increasing LRMs in the system. Moreover, resource management with central GRM has a single point of failure. Suppose

the GRM fails or the node running the GRM fails (or loses network connectivity to the system). Then, the LRMs alone can only make node-level resource management decisions without the possibility to leverage system-wide resources resulting in limited adaptability in the system. Furthermore, any security flaws or faults in the GRM can compromise global resource management. A faulty or compromised GRM can make erroneous or fraudulent global reconfiguration decisions. It can also communicate inconsistent decisions to the LRMs. These faults and flaws can impact the execution of real-time applications. An attacker can also manipulate the stored system information or current (abstract) status of the resources (or nodes) that the GRM has gathered so far from the LRMs. Furthermore, as discussed in the previous section, if the attackers get physical access to the platform, they can potentially read stored keys and masquerade as a GRM. The attackers can also conceivably read and manipulate all incoming and outgoing messages if attackers install malicious software between the GRM and the security sublayer.

This section proposes extending the concepts discussed so far to make global resource management safe and secure by using distributed global decisions instead of centralized decisions. To do so, we eradicate the central GRM and add a new resource management component, the Distributed Global Resource Manager (DGRM) component, on all nodes with LRMs in the highest level of the hierarchy. Thus, such a node N_λ will resource management components $DGRM_\lambda$ and LRM_λ (as shown in Figure V.F7a). The corresponding LRMs function almost the same as before. The only difference is that instead of sending updates (including reconfiguration requests) to the central GRM, each LRM_λ sends updates to the $DGRM_\lambda$ on the same Node (N_λ). Instead of the central GRM making the global decision on its own, the DGRM components on all these nodes work together for making global decisions based on distributed coordination. A similar concept applies to the flat resource management architecture. As a result, we have two new resource management architectures, as shown in Figures V.F7b and V.F7c, in addition to those explained in Chapter III.

V.6.1 Distributed Global Resource Manager (DGRM)

DGRM is a distributed component that participates in making global resource management decisions. It receives abstracted monitoring updates from the LRM on the same node. Based on these updates, the DGRMs on all nodes together maintain an abstracted global view of the entire system. Henceforth, we will refer to the abstracted global view stored in the GRM as the system state for simplicity. As before, the global resource management decisions to mitigate the issues that a local reconfiguration alone cannot resolve. In such cases, DGRMs work together in a distributed manner to find node(s) in the system that can reallocate their resources to host the affected applications and meet their resource demands. The DGRMs update their stored system state to reflect this change and send reconfiguration orders to the LRMs in the subsystem affected by the change. In turn, the LRMs translate these orders to the node's local scheduling policy and apply them via their LRSs. It is important to note that global resource management does not perform resource scheduling itself; it only manages the allocation of resources



(a) *Interaction of LRM with DGRM on a node N_λ*

(b) *Hierarchical Architecture with DGRM*

(c) *Flat Architecture with DGRM*

Figure V.F7: *Distributed Global Resource Manager (DGRM)*

in the system as a basis. The LRMs are responsible for actually scheduling (via LRSs) the resources as per the global orders.

As we use DGRMs instead of the central GRM, there is neither a single point of failure nor a bottleneck in the resource management anymore. However, we must ensure that DGRMs are safe and secure themselves. Therefore, we identified the following challenges and requirements concerning safety and security for building the DGRMs:

Requirement R1 The global resource management maintains a distributed system state via the DGRMs. The stored system state should satisfy the following:

R1.1 The stored system state should be consistent across all DGRMs, i.e., no DGRM should have a different system state view different from others at any point in time.

R1.2 It should not be lost upon failure(s) in one or few DGRMs. A reconfiguration in any domain of the system should not result in the loss of the system state.

R1.3 The system state should be Byzantine fault-tolerant to include the possibility that some DGRMs are malicious. Thus, in such a system, the DGRMs must reach a consensus on the system state, and each DGRM must know when the quorum is achieved.

R1.4 The system state must be readable or modifiable only by legitimate DGRMs authorized by the system designers. This requirement safeguards sensitive system information from malicious attackers and prevents attackers from masquerading as a DGRM or manipulating the stored system state.

Requirement R2 The communication between the LRMs and DGRMs for sending updates and reconfiguration orders should be secure. As before, the communication requires the following:

R2.1 integrity, to protect exchanged messages from being manipulated

R2.2 authenticity, to ensure that the messages originated from legitimate LRMs and DGRMs

R2.3 confidentiality, as there is sensitive system information in the messages

Requirement R3 Since the DGRMs make decisions with distributed coordination, we need to ensure that they do not double-spend (over-allocate) resources.

Requirement R4 Instead of the LRMs, now the DGRMs require a membership protocol among themselves to detect if all nodes are responsive. When a node is non-responsive, the global resource management must take further action to ensure continuity of services.

Requirement R5 The system needs to log all global resource management decisions to trace any errors that may occur. However, the log should not be readable or modifiable by an attacker.

V.7 Blockchain-Based Safe and Secure Distributed Global Resource Management

This section describes the implementation of our global resource management approach with distributed global resource management decision-making. The implementation ensures that we meet the safety and security requirements highlighted in the previous section.

The DGRMs do not perform the scheduling themselves; they only manage the allocation of the system resources based on abstracted system state. Based on the global resource management decisions received and translated (to local scheduling policy) by the LRMs, the LRSs do the actual scheduling to ensure that the real-time applications running on the node meet their resource demands. As a result of this decoupling, we can implement the global resource management using a blockchain (Section II.21) without impacting the currently executing real-time applications.

As explained in Section II.21, there are two types of blockchains – public or private blockchain . Both are decentralized approaches to maintain a shared ledger of digitally signed transactions. However, public blockchains have low transaction speeds, high costs, and high energy consumption. On the other hand, private blockchains are permissioned, and only participants pre-approved by a system designer can join the blockchain or initiate adding a new entry to the distributed ledger. As a result, they have fast transaction speeds, low costs for each transaction, and low energy consumption.

We choose a private (permissioned) blockchain called the Hyperledger Sawtooth [138] for our implementation. Section II.21 provides an introduction to the Sawtooth platform. Sawtooth has a modular design that helps us to separate the core blockchain platform from resource management. As a result, we can use smart contracts to specify the global resource management logic without knowing the underlying design of the blockchain platform. Moreover, Sawtooth supports Practical Byzantine Fault Tolerance (PBFT) algorithm for consensus; hence it is suitable for our requirements. Other open-source private blockchain platforms do not have support for Byzantine fault-tolerant algorithms. Sawtooth minimizes security and safety risks as it only allows a fixed set and semantics of transactions via transaction families. A transaction family is a set of operations or transaction types allowed on the distributed ledger. With transaction families, a system designer can select the level of versatility and risk suitable for the system. Each transaction family consists of a Client (CLI), a data model to store the data in the distributed ledger, and a smart contract (also called a Transaction Processor (TP)). As explained in Section II.21, smart contracts are small programs written in a programming language such as C++. Smart contracts execute on the blockchain and facilitate the implementation of use case-specific functions.

V.7.1 RM Transaction Family

We implemented a new Sawtooth transaction family called the *Resource Manager (RM) transaction family* for implementing the DGRMs and the LRMs. As shown in Figure V.F8a, the RM transaction family consists of the following:

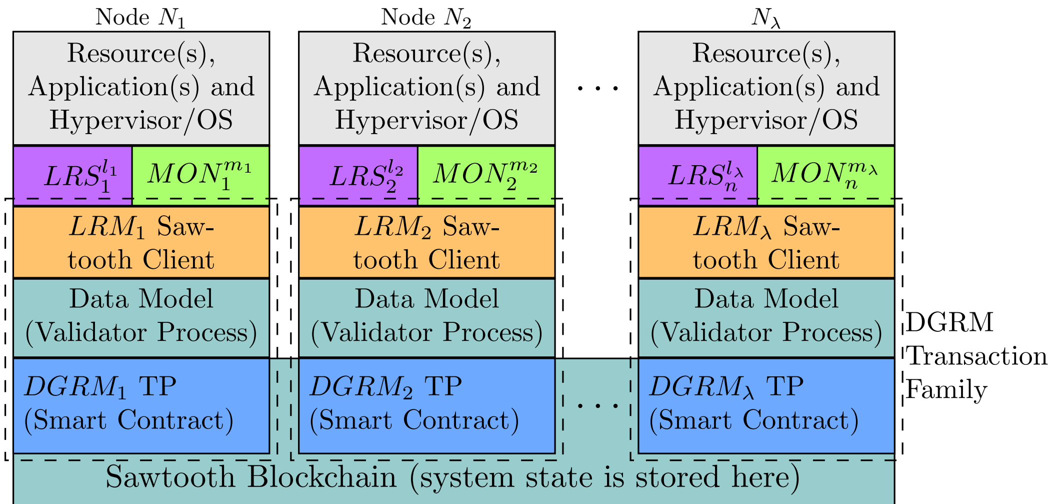
1. An LRM CLI to handle the local resource management logic.
2. A data model to record the system state in the blockchain.
3. A DGRM smart contract (TP) that runs in the blockchain and handles the global resource management logic for distributed decision-making.

Sawtooth views a smart contract as a state machine. Smart contracts take transactions (from a distributed log) and the distributed ledger's current data state as input and write back a new data state to the distributed ledger. A Sawtooth component called the *validator* ensures that the same transaction results in the same transition and the same resulting data state for all participants. This view of smart contracts matches our perspective of the global resource management discussed in this chapter. Thus, we chose smart contracts of the Sawtooth platform to implement the DGRM logic and associated functions. Note that some parts of the logic implemented in the DGRM transaction family are dependent on the use case-specific resource management requirements. An example use case scenario is presented in Chapter VII to give a demonstration of the RM transaction family.

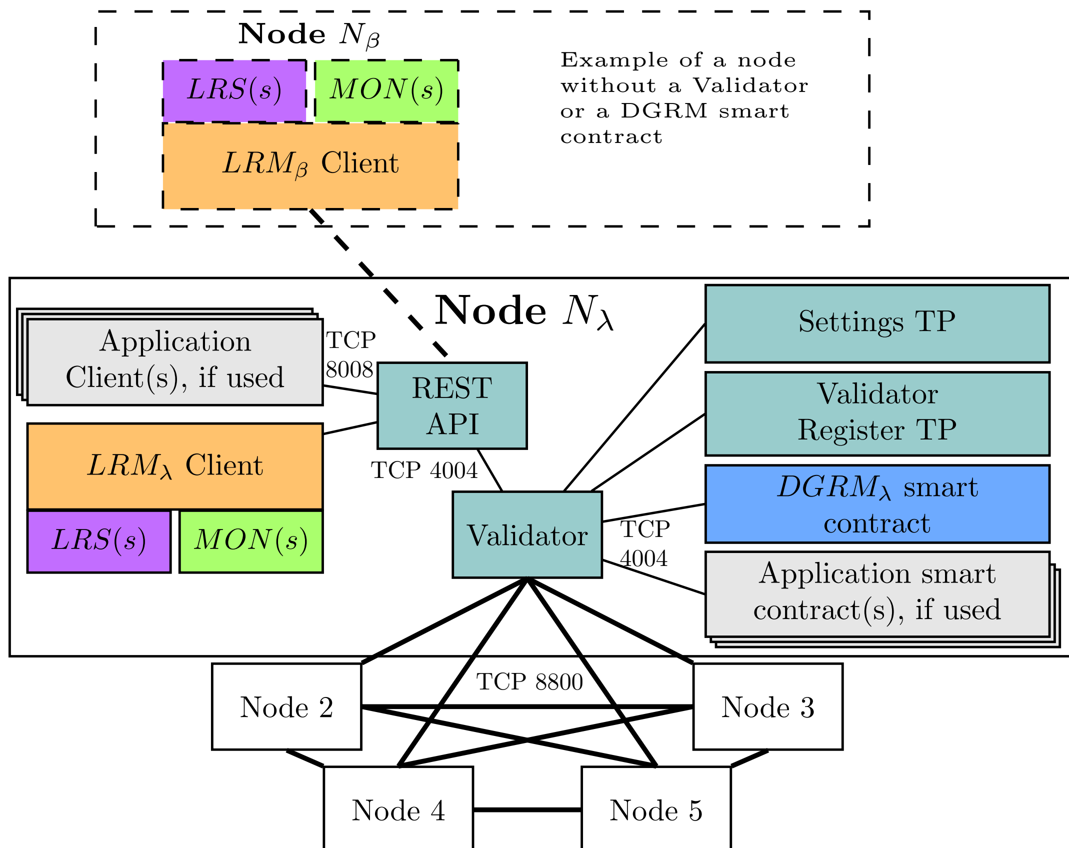
The following subsections explain how we satisfy the requirements from Section V.6.1 using the Sawtooth blockchain platform and the RM transaction family. Figure V.F8b shows the Sawtooth relevant components and how they interact with the RM transaction family on the node, N^{λ}, in a system containing five nodes with DGRMs and LRMs.

Requirement R1.1 - R1.3

The DGRMs store the system state in the Sawtooth blockchain . Sawtooth blockchain is a ledger distributed to all (potentially untrusted) nodes. There is no central administrator or centralized database that could potentially leak transaction patterns or other confidential information. The distributed ledger is demonstrably identical on all nodes, i.e., all nodes in the system have the same information. Furthermore, since the ledger is available on all nodes, it is not lost if a node fails or leaves the system. As a result, all DGRMs have a consistent view of the stored system state. The system state is not lost if a few nodes fail. The ledger is also immutable as Sawtooth uses block hashes to detect and prevent attempts to alter the history. Any changes to the global system-state stored in the ledger can only be performed via transactions and pre-defined transaction families. Applying transactions involves submitting them to the validator via the *REST API* provided by Sawtooth. A crucial function of the validator is to achieve consensus among all DGRMs on the ordering of transactions. The validator also stores the resulting state in the distributed ledger after processing each transaction. The Sawtooth platform provides different consensus algorithms as plugins such as PBFT algorithm. We use the PBFT algorithm plugin for our implementation so that the stored system state is Byzantine fault-tolerant. Taking all these properties of the Sawtooth blockchain into account, we fulfill requirements R1.1 to R1.3. Note that the PBFT algorithm requires at least four participants to reach a consensus. Hence, our implementation requires at least four Sawtooth enabled nodes with DGRMs.



(a) *RM Transaction Family Components*



(b) *Integration of RM Transaction Family with Sawtooth*

Figure V.F8: *RM Transaction Family for Sawtooth*

Requirement R1.4

The Sawtooth network design helps us solve the challenges of permissioned (private) networks. The Sawtooth network only grants permissions to modify and read the system state stored in the blockchain to legitimate DGRMs based on public-key cryptography. These DGRMs must be authorized by a system designer during the design phase. Sawtooth provides secure P2P communication over TCP to submit transactions and messages. It provides security by requiring all transactions to be signed by known identities (DGRMs in our case). It is possible to control who can connect to the network and sync the current system state stored in the ledger, participate in the consensus process, and submit DGRM transactions. Sawtooth also provides some inbuilt smart contracts, such as the Settings TP and the Identity TP, that let us store and modify Sawtooth settings. For example, these settings include who can participate in the consensus or the minimum number of votes needed to accept a proposal. Besides, we use the Settings TP to store settings needed by the DGRM smart contract at runtime. The settings are stored on the blockchain itself, and hence, they are immune to manipulation by an attacker and not lost if a few components fail. Considering all these points, an attacker cannot read, manipulate the system state or make global decisions. Thus, the Requirement R1.4 is met.

Requirement R2

The LRMs still use the conceptual resource management update channel from Section V.2. Hence, it is possible to use the security sublayer that we developed in Section V.4.1 to send secure messages for global resource management. However, as seen in Figure V.F8b, the DGRMs TP and LRM CLI communicate via the validator and the secure Sawtooth network. Similarly, the messages exchanged by the DGRMs for achieving consensus to store the system also pass through the secure Sawtooth network. As a result, a result we do not need to use the sublayer. The sawtooth network already provides security equivalent to security Level 2 from Section V.4.2. As a result, Requirements R2.1 to R2.3 are fulfilled.

Requirement R3

All modifications to the system state stored in the blockchain take place as transactions. Sawtooth ensures that all the transactions are processed by the DGRM smart contract according to their order. The validator ensures the total ordering of every transaction. The distributed ledger state is incremental per transaction execution only. So it is easy to enforce rules in the DGRM smart contract to ensure that the resources are not double spent, thus meeting Requirement R3.

Requirement R4

A list of known peers (other nodes running a validator) is available to each validator. The validators maintain a list of active peers based on heartbeats and received consensus messages from other nodes. LRM_λ on a node N_λ can obtain this from the validator

of N_λ list and determine if other nodes are reachable; if not, it informs the $DGRM_\lambda$ smart contract about the failure, which can, in turn, trigger a global reconfiguration. Consequently, Requirement R4 is satisfied.

Requirement R5

Sawtooth stores records of all transactions in a distributed log. The DGRMs or external applications authorized by the system designer can access the distributed log. Since all transactions are written in a distributed log, Requirement R5 is automatically fulfilled.

V.7.2 RM Transaction

As earlier, the MONs send monitoring information from resources and applications to LRM CLIs regularly. Upon long-term changes in availability or demands of local resources, the LRMs generate an update channel message. However, instead of sending the messages to the security sublayer, the LRMs now package messages as Sawtooth transactions. We will refer to these transactions as Update transactions.

An Update transaction consists of a header signed by an LRM CLI and a payload containing the abstracted update (including reconfiguration request). The header information includes the name and version of the DGRM smart contract and input/output addresses to read/write from the blockchain. The validator securely receives the Update transaction from the LRM CLI via the REST API. In general, a node with an LRM CLI does not necessarily need a validator and DGRM smart contract to be present. For example, Node N_β shown in Figure V.F8b has no validator or DGRM smart contract present on the same node. However, it can still send the transaction request to the validator of another node (as authorized by the system designer).

The validator applies a cryptographic hash function to the content of the received transaction. Then, the validator uses its private keys to sign the generated hash values, and the other nodes use this signature to validate the authenticity and integrity of the transactions. Next, the validator broadcasts the transactions to the validators of other nodes in the Sawtooth network. The validators of nodes receiving this transaction broadcast a (signed) consensus message if the received transactions pass authenticity and integrity check; otherwise, the validators discard the transaction. Based on the PBFT algorithm, the transaction is committed to the distributed log present on the blockchain when $3f + 1$ consensus messages are available in a system with f acceptable faulty nodes. The consensus mechanism also ensures the total ordering of every update transaction generated by the Coordinator CLIs. Once the transaction is committed to the distributed log, the validators (on all nodes) send the payload to the DGRM smart contract of their node for further processing.

DGRM smart contracts on all nodes are identical. They enforce checks on the payload data to ensure it has the correct data format and passes the data sanity checks. For example, if an update claims more resource availability than physically present on the node generating the update, the check will fail. If one of the checks fails, the transaction is considered faulty and not processed further. Therefore, the validator of that node does not generate a consensus message. If the transaction payload passes the check,

the DGRM smart contract of those nodes process the transaction further as per the implemented global resource management logic (use-case specific) and generate a new system state as an output. The new system state reflects the (abstracted) updates sent by the LRMs and may contain reconfiguration orders for one or more LRM CLIs. The new global system-state is sent to the validator on each node to write it in the blockchain. As per the PBFT algorithm, if validators of $3f + 1$ nodes agree on the generated system state, then the system state is written to the blockchain. We use the Sawtooth event subscriptions mechanism for subscribing the LRM CLIs to receive notifications about any change in the system state related to their subsystem. Those LRM CLIs that receive such a notification apply any orders present in the system state to their nodes via LRSs.

We provide fault-tolerance to the LRM CLIs via replication as explained in the previous chapter. If a Sawtooth validator breaks, the corresponding LRM CLI cannot submit transactions to it. In such a case, the LRM CLI submits them to a validator on another node (offline defined order of node selection). It can continue functioning without an active validator on the same node similar to Node N_β in Figure V.F8b. Note that the PBFT algorithm requires at least four validators to reach a consensus.

In Chapter VII, we present an example scenario with a concrete implementation of the RM transaction family including the DGRM smart contract. Moreover, we perform experiments to evaluate the RM transaction family and obtain the observed maximum delay for global reconfiguration (including distributed global resource management decision-making).

V.8 Chapter Summary

In this chapter, we introduced the Global Resource Manager (GRM) of our resource management architecture. The GRM provides various services, such as gathering updates from LRMs, global reconfiguration decision-making, obtaining or computing a new global configuration, sending/receiving orders/updates to/from the LRMs, and managing an external input. Conceptually, one GRM exists in the system, although distribution is possible for fault tolerance and scalability. Therefore, the GRM can be realized either by a single node or a set of nodes.

We proposed three conceptual communication channels between the central GRM and the LRMs (and between higher and lower domain LRMs) to exchange information – Update, Order and Membership channels. In addition, we provided an overview about using two popular existing protocols, MQTT and OPC UA, for resource management communication, taking into consideration requirements such as membership, security, and reliable message delivery.

Unsecured resource management has several weak spots that an attacker can exploit. Therefore, we[1] analyzed the resource management communication, LRM, and GRM from a security viewpoint. We proposed security services for resource management communication to prevent various security attacks. In addition, we proposed three different security levels (Levels 0, 1, and 2) for resource management communication and

[1]Based on inputs from the security experts at the University of Siegen

two options to implement the security services – implementation as a support module or implementation as a layer between the resource managers and the underlying hardware or software (hypervisor or OS). Moreover, we[1] discussed two options for security algorithms – ChaCha20-Poly1305 and CLEFIA (in Offset CodeBook (OCB) operation mode).

Finally, in this chapter, we discussed several limitations of a single central global resource manager. We proposed extending the concepts to make global resource management safe and secure using distributed global decisions instead of centralized decisions. To do so, we eradicated the central GRM and added a new resource management component, the DGRM component, on all nodes with LRMs in the highest level of the hierarchy. Furthermore, we identified the challenges and requirements concerning safety and security for designing the DGRMs. We choose a private (permissioned) blockchain called the Hyperledger Sawtooth [138] for the implementation as it helps us to meet the challenges and requirements concerning safety and security. We designed a new Sawtooth transaction family called the Resource Manager (RM)-transaction family for implementing the DGRMs and the LRMs of our resource management framework.

VI

Avionics Use Case: Resource Management for Distributed Mixed-Critical System (MCS)

"Flying isn't dangerous. Crashing is what's dangerous."

– Unknown

The following avionics use case was developed during the DREAMS project. The avionics use case system consists of several multicore processors and Multi-Processor System on Chips (MPSoCs) connected via a Time-Triggered (TT) network. The system hosts three safety-critical avionics applications and multiple instances of a best-effort application. We use the avionics use case to evaluate the proposed resource management framework in two scenarios:

1. **Permanent core failure:** As a result of intensive integration of on-chip devices, permanent core failures can occur due to phenomena such as wear-out and infant mortality [173]. When a core fails, the resource management needs to redeploy the application partition(s) executing on the failed core in one of the following ways within the maximum allowed unavailability time as per offline defined schedules: redeploy to another core of the same node (local reconfiguration) or redeploy to another node in the system (global reconfiguration).

   Our evaluation focuses on the fault tolerance capacity of the resource management against core failures and the time taken for global reconfiguration on a realistic platform (Section VI.5.1).

2. **Temporal overload condition (potential deadline overrun):** A temporal overload can occur due to over-utilized resources in the nominal mode. Resource management can ensure the timing constraints for the critical applications by suspending the best effort applications based on potential deadline overrun monitoring (local adaptation).

   Our evaluation focuses on the efficiency of the resource management adaptation to improve the system utilization while ensuring the critical applications meet their timing requirements (Section VI.5.2).

Finally, we present an experimental evaluation for the time taken by the secure resource management communication (with single central Global Resource Manager (GRM)) in the avionics use case using all three security levels and both security algorithms presented in Chapter V.

VI.1 Avionics Use Case

The avionics use case consists of three safety-critical applications that are examples of next-generation applications for use in future avionics systems [212]:

1. Flight Management System (FMS) [213] - It is responsible for services that provide in-flight path management based inputs such as pre-defined flight plans, aircraft position (determined by sensor data), and pilot directives. This application has stringent real-time requirements and has a safety criticality level of DAL B.

2. Display Management System (DMS) - It is responsible for managing the information displayed on the cockpit panel and sending the input provided by the pilot to the applications, such as the Flight Management System (FMS), via the network. Figure VI.F1 shows an example of such a display. The criticality level of Display Management System (DMS) is the same as the highest criticality application receiving input from the DMS. In our case, the DMS provides inputs to the FMS. Hence, it has the same criticality level as the FMS.

Figure VI.F1: *Example of a Flight Display Panel*

3. Sensor Data Provider (SDP) - It is responsible for collecting the sensor data, such as the GPS position, and forwarding it to the FMS as packets via the network. This application has stringent real-time requirements and has a safety criticality level of DAL A. Since the use case does not use actual physical components that move, the Sensor Data Provider (SDP) values do not change. Instead, the use case feeds the trajectory computations performed by the FMS back to the SDP for

avoiding this problem. The SDP, in turn, uses this input to generate new data and sends it to the FMS. As a result, in this use case, the communication is not only in the direction of SDP to FMS, but also the reverse direction.

As best-effort application, we consider a Quality of Service (QoS)-enabled MPEG-2 video server [214]. A similar application was used in the demonstration of the Matrix [21], and ACTORS project [22] resource management framework. The application provides three QoS levels: HIGH/MEDIUM/LOW. At the HIGH QoS level, the application process all MPEG-2 frames. However, at MEDIUM and LOW QoS levels, the application selects only those frames which provide the best possible picture quality while using only the available amount of resources for the corresponding service level.

In this use case, a cockpit panel (communicating with DMS) is treated as best-effort tasks and simulated on separate x86 processors running Linux. Figure VI.F2 illustrates the applications in the avionics system and the interactions among them.

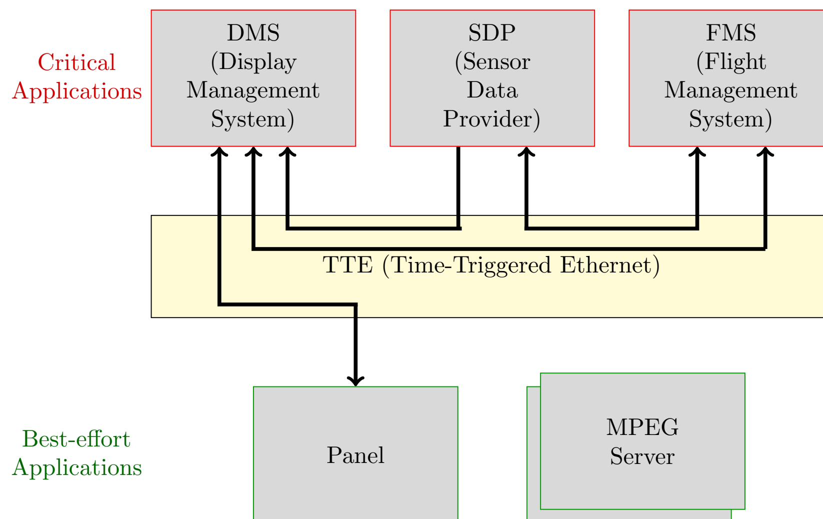


Figure VI.F2: *Applications of the Avionics Use Case*

Table VI.T1: *Avionics Use Case Application Parameters*

Application	Criticality Level (DAL)	Maximum Unavailability (ms)	Number of Tasks (Periodic,Aperiodic)
FMS	B	600	26 (10,16)
DMS	B	600	5 (4,1)
SDP	A	1000	7 (6,1)
MPEG Server	E	∞	Best-effort
Panel	E	∞	Best-effort

Table VI.T1 summarizes the application parameters, such as the criticality level (DAL), maximum unavailability, and the number of tasks for each application. Maximum

unavailability corresponds to the maximum suspension time for the application due to (resource) failures. Further information about the avionics applications is not provided here due to confidentiality issues.

VI.2 Use Case Demonstrator

The demonstrator consists of two different types of hardware platforms:

- *NXP QorIQ T4240 multicore processor* board (introduced in Section II.4.1) with a TT-Ethernet (TTE) PCI-Express card.
- *DREAMS Harmonized platform (DHP)*: This platform has a Xilinx ZC706 board with Zynq 7000 MPSoC. As explained in Section II.4.2, the Zynq 7000 MPSoC contains an ARM Cortex-A9 dual-core processor and an FPGA on the same die. The DHP implements a Spidergon TT Network-on-Chip (NoC) [29] (and the required Network Interfaces (NIs)), Microblaze softcore processors and a TTE controller in the Field Programmable Gate Array (FPGA). The TT NoC connects the Cortex-A9 and the remaining components on the FPGA with each other. Note that we do not use the Microblaze processors for this demonstrator.

In the demonstrator, three boards – two T4240 and one DHP, are connected via a TTE network (Section II.13) explains TTE with an example), as shown in Figure VI.F3. The demonstrator hosts the three safety-critical avionics applications and multiple instances of the best-effort MPEG server application. An x86 PC, connected via regular Ethernet, simulates the best-effort panels. However, henceforth, we will not refer to the x86 PC as it is only present to simulate the panels and does not require resource management. Each node runs the XtratuM hypervisor (introduced in Section II.17.4) that supports ARINC 653 scheduling (explained in Section II.13.1) of partitions[1] and ensures strong temporal and spatial partitioning complaint with Integrated Modular Avionics (IMA) and other avionics standards [33, 215]. FentISS and Valencia Polytechnic University (UPV) extended XtratuM to provide interfaces for applications (including resource managers) to access the TTE network based on an offline computed TTE schedule.

VI.3 Resource Management Implementation on the Demonstrator

We (together with Thales R&T and ONERA) implemented resource management services on top of the T4240 and DHP nodes of the avionics demonstrator. The following subsections provide the resource management components that we used in the avionics demonstrator and explain the implementation choices for each component where required. Figure VI.F4a and VI.F4b illustrate the resource management components on the two T4240 boards and the DHP.

[1] XtratuM uses the term Virtual Machine (VM) and partition interchangeably.

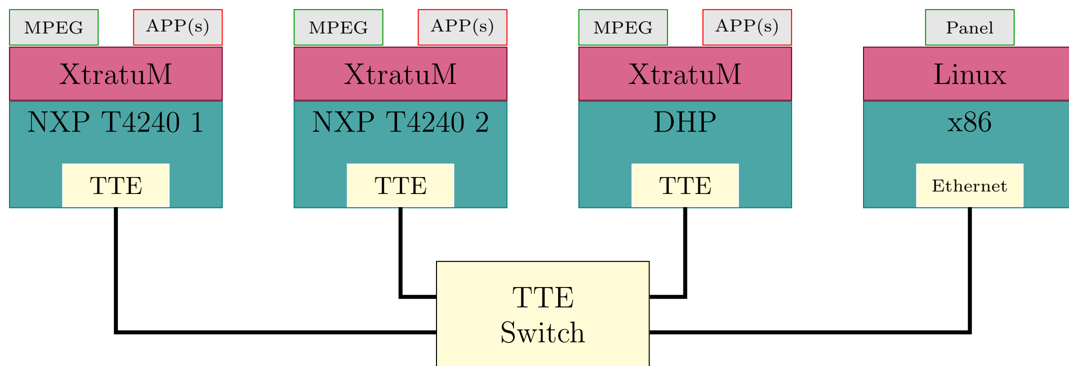


Figure VI.F3: *Avionics Use Case Demonstrator*

VI.3.1 Local Resource Schedulers (LRSs)

There are two types of LRSs present on each node:

1. **LRS for Critical Partitions** (Section IV.7.1) to execute tasks of a critical application in a partition.

2. **LRS for Online Reconfiguration** (Section IV.7.2) to support the Local Resource Manager (LRM) of a node.

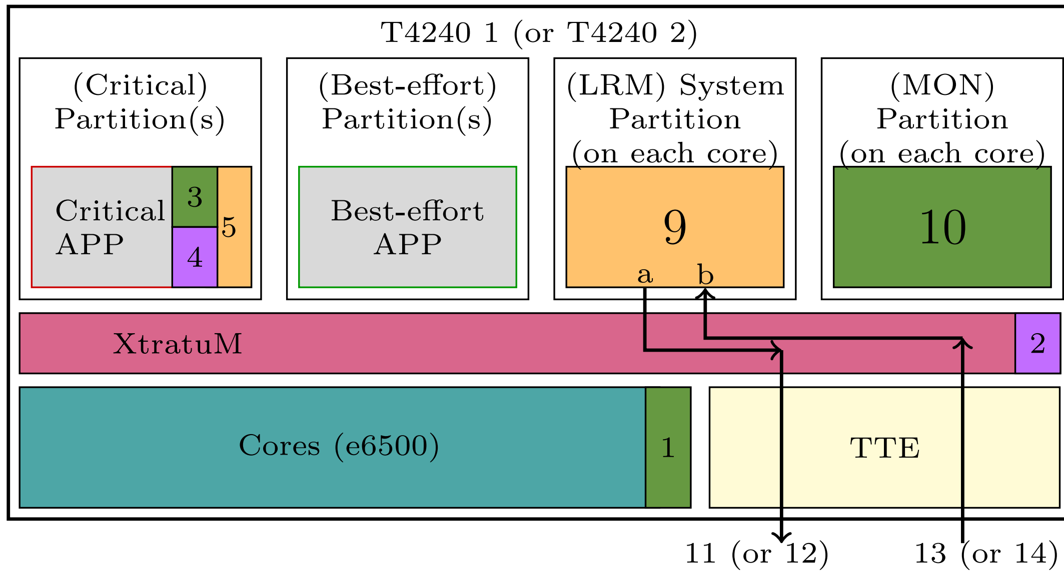
VI.3.2 Local Resource Monitor (MONs)

There are three types of MONs present on each node:

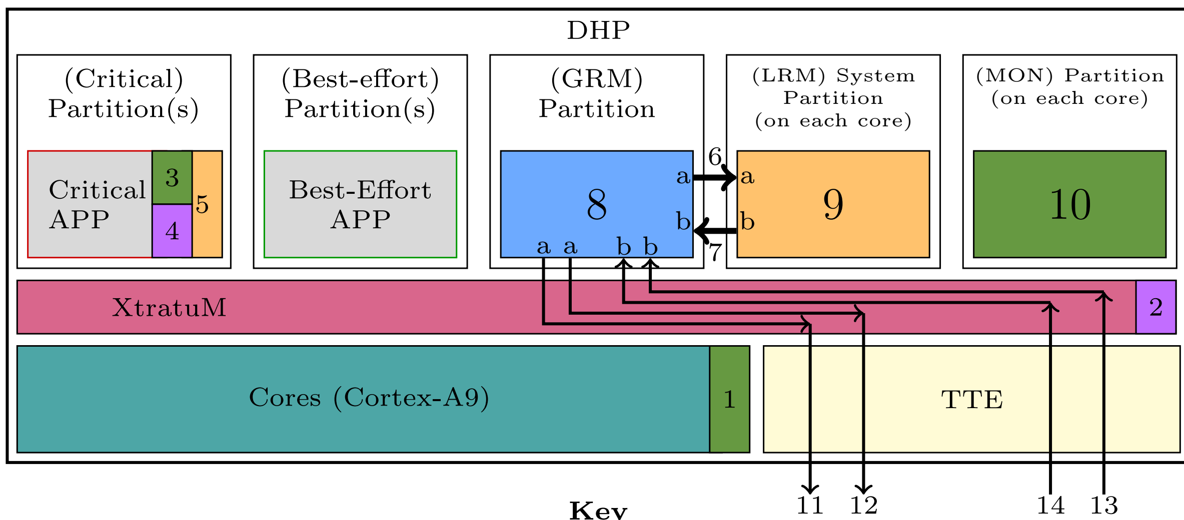
1. **Core failure MONs** (Section IV.4.3): We implemented instances of core MON as asynchronously executing XtratuM partitions on each core of a node. We provide the schedule for executing core failure MON partitions on a node via the offline-defined configuration.

2. **Potenital deadline overrun MONs** (Section IV.4.4) for critical application partitions: In Section IV.4.4, we discussed two options to place instances of this MON for use with the critical applications. From these two options, we chose to add deadline overrun MON instances between the critical application tasks in a partition as it avoids task instrumentation and helps to keep the monitoring overheads to a minimum.

3. **Hardware MONs** to support deadline overrun MONs (Section IV.9.2): We execute hardware MONs before and after each task of an application. Moreover, we use hardware MONs to determine the overheads of the resource management itself by executing this MONs before and after each resource management component (not shown in Figure VI.F4 for clarity). Since this MON consists only a few lines of assembly code, the monitoring overhead (probe effect) is negligible.



(a) *DHP*



Key

1.	Hardware MON
2.	LRS for reconfiguration
3.	Potential Deadline Overrun MON (+ Hardware MON)
4.	LRS for critical application partition
5.	Application Domain LRM for deadline overrun management
6.	Order channel (XtratuM sampling channel - DHP only)
7.	Update channel (XtratuM queuing channel - DHP only)
8.	GRM (DHP Only)
9.	LRM in virtualization domain (synchronous execution on each core)
10.	Core Failure MON (asynchronous execution on each core)
11.,12.	Order channels (TT VLs) to T4240 1 (and T4240 2)
13.,14.	Update channels (TT VLs) from T4240 1 (and T4240 2)
a.,b.	Sampling ports, Queuing ports

(b) *NXP T4240*

Figure VI.F4: *Resource Management Components in the Avionics Demonstrator*

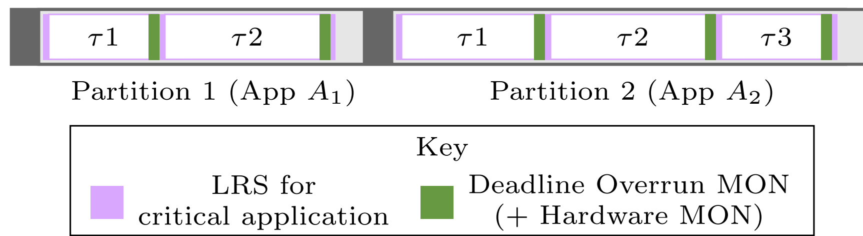


Figure VI.F5: *Deadline Overrun MON in Avionics*

VI.3.3 LRMs

There are two LRMs present on each node as follows:

1. **LRM in virtualization domain**: This LRM provides local reconfiguration upon core failure (Section IV.9.1). We implemented instances of this LRM as synchronously executing XtratuM partitions on each core of a node at the end of every Major Frame (MaF). As explained in Section IV.9.1, we execute LRM instances synchronously on each core to ensure we do not lose local resource management capabilities upon core failures. LRM instances are executed at end of every MaF. Figure VI.F6 illustrates this as a reminder. We provide the schedule for the execution of the LRM partitions on each node via the offline-defined configuration. Section VI.4 explains the use case specific local reconfiguration strategy.

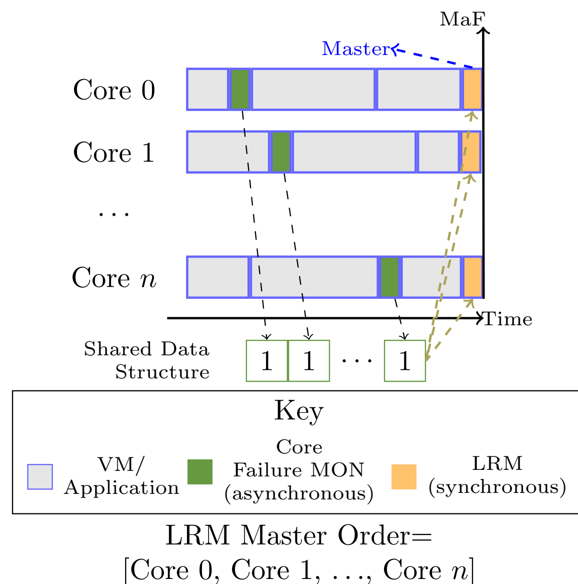


Figure VI.F6: *LRM for Core Failure Management in Avionics*

2. **LRM for deadline overrun management:** This LRM ensures that the critical applications meet their timing constraints while improving the QoS of non-critical

applications (Section IV.9.2).

The LRM requires each node to be over-utilized as per Equations IV.4, IV.5 and IV.6 from Section IV.9.2. Moreover, the same section discussed three options (with their advantages and disadvantages) to implement this LRM. We selected the third option, i.e., executing the LRM in the *application domain* (LRM instances executing directly after deadline overrun MONs instances). We chose this option because the adaptation by the LRM is almost immediate since the LRM executes directly after the MON ($t_{LRM} = 0$ in Equation IV.1). Moreover, this approach has a very low performance impact. The only restriction with this approach was that the underlying hypervisor must allow starting or stopping best effort partitions from within a critical application partition. Thanks to the XtratuM extensions by FentISS and UPV, we can perform the required actions from within a critical application partition. Figure VI.F7 illustrates the positioning of these LRM instances as a reminder.

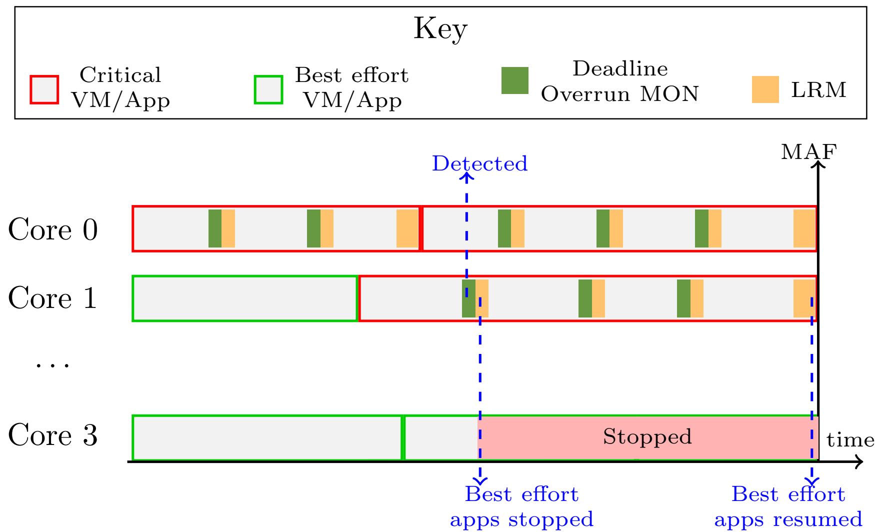


Figure VI.F7: *Potential Deadline Overrun Management in Avionics*

To reduce the impact of deadline overrun management on the QoS of the best-effort applications and increase the utilization of the node's resources, Thales R&T further implemented the second solution described in Section IV.9.2. However, we don't consider it in this dissertation.

VI.3.4 GRM

To keep the global resource management overheads low, we chose to implement a central GRM on the DHP. The GRM facilitates recovery upon core failures when virtualization domain LRMs cannot recover on their own. We implemented the GRM as an XtratuM partition. The GRM partition does not need any extra privileges because it only decides the reconfiguration but does not apply it. The LRMs on each node apply the actual

reconfiguration via their LRSs. We provide the schedule for executing the GRM partition on the DHP via the offline-defined configuration. Section VI.4 explains the use case specific global reconfiguration strategy.

Table VI.T2 summarizes the implementation of the resource management components discussed in the previous subsections.

Table VI.T2: *Resource Management Component Implementation Summary*

Component	Location	Function	Description
LRS for critical partitions	Critical application partition	Execute critical application tasks	Section IV.7.1
LRS for reconfiguration	Virtualization domain LRM partition	Plan (mode) changes for XtratuM	Section IV.7.2
Core failure MON	Asynchronous execution as XtratuM partition on each core	Core failure detection	Section IV.4.3
Potential deadline overrun MON	Execution between tasks of a critical application	detection of potential deadline overrun in critical applications	Section IV.4.4
LRM in virtualization domain	Synchronous execution as XtratuM system partition on each core towards end of MaF	Core failure management on a node	Section IV.9.1
LRM in application domain	Execution immediately after deadline overrun MON	Deadline overrun management in critical partition	Section IV.9.2
GRM	Execution as XtratuM partition on the DHP	Global core failure management	Section V.3

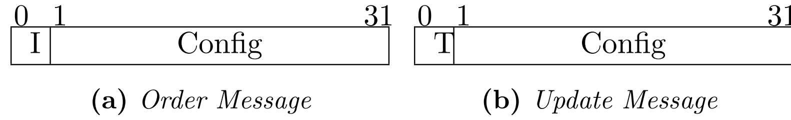
VI.3.5 Resource Management Communication

We implemented two resource management communication channels from Section V.2:

1. An **order channel** between the GRM (source) and each LRM (destination). The GRM uses these channels to send reconfiguration orders to the LRMs. A message is only sent on a order channel if there is a new reconfiguration order for the destination LRM.

2. An **update channel** between the GRM (destination) and each LRM (source). Each LRM sends periodic (every MaF) updates, including reconfiguration requests, to the GRM on the update channels.

As explained in Section V.2.3, we do not require a membership channel as the LRMs send periodic updates to the GRM.



(a) *Order Message* **(b)** *Update Message*

Figure VI.F8: *Resource Management Communication Message Formats*

The messages sent on update and order channels are 32-bit wide. Figure VI.F8 shows the message format. The order message consists of the following fields:

1. Immediate (I): 1-bit field to indicate if the destination LRM must apply the new offline schedule immediately (0) or deferred (1).

2. Configuration (Config): 31-bit wide field with the new offline schedule for the destination LRM to apply on its node.

The Update message consists of the following fields:

1. Type (T): 1-bit field to indicate if the source LRM is sending a simple update message (0) or a reconfiguration request (1).

2. Configuration (Config): 31-bit wide field with current configuration that the source node is executing.

Table VI.T3: *Resource Management Communication Implementation*

Channel	XtratuM Port	TTE Virtual Link (VL) Type	Source	Destination
Update	Queuing	TT	LRM	GRM
Order	Sampling	TT	GRM	LRM

To ensure timely delivery of resource management messages and bounded global resource reconfiguration time, we implement these channels as TT VLs on the TTE network. Furthermore, we provide the schedule for these VLs via the offline-defined TTE schedule. Since XtratuM implements the ARINC 653 standard, each resource manager can send messages on the TT VLs via sample or queuing ports (introduced in II.13.1). As explained in Section V.2, the GRM needs all updates from the LRMs. Thus, we use a queuing port for the Update channel. Contrarily, the LRMs require only the latest order from the GRM. Thus, we use a sampling port for the Order channel. Table VI.T3 summarizes the implementation of the communication channels. Note that both GRM and LRM are present on the DHP, In this case, we do not use TTE VLs for communication between the GRM and the LRM on the DHP. The GRM communicates with this LRM via XtratuM sampling and queuing channels. Figure VI.F4 (on Page 148) illustrates the communication channels for resource managers on the two T4240 boards and the DHP.

We implemented all three Security Levels – 0, 1, and 2, and both the security algorithms from Section V.4.1. Each secure resource management message (Level 1 and 2) is sent as a secure frame consisting of three parts as follows:

1. The first part is the header which is not encrypted but authenticated by the third part of the message. In security domain, the header is also called as Additional Authentication Data (AAD). The header includes:
   - length of the header (*aad length*),
   - length of the cipher text (*ciphertext.length*),
   - *nonce* that includes a time-varying parameter,
   - XtratuM source and the destination partition IDs (*sourceID* and *DestID*) for the channel,
   - XtratuM port descriptor (*Portdesc*),
   - XtratuM port type (*Ptype*) - Sampling or Queuing port,
   - *flag* field with variable length. We omit the flags currently, i.e., flags size is 0 Bytes. However, it is possible to use them in future extensions.
2. The second part of the frame is a payload containing the encrypted (cipher) text for security Level 2 messages (*chiper.text*) or unencrypted (plain) text for security Level 1 messages (*plain.text*). The payload can have variable length. ´ We use the update and order message format shown in Figure VI.F8. Hence, message size is 4 Bytes. However, it is possible to send messages up to 32 bytes in future extensions.
3. The third part, *Message Authentication Code (MAC)*, secures the first two parts.

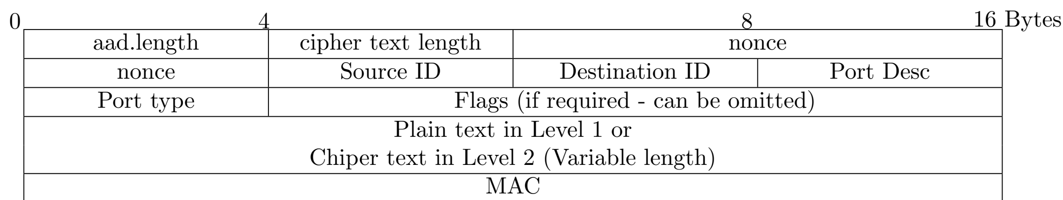


Figure VI.F9: *Secure Message Frame*

Security Level 2 increases the message size at least by 416 bit over Level 0: 288 bit for the header and 128 bit for the MAC. Figure VI.F9 shows an example of a Level 2 frame. The MAC in the third part authenticates both the first part (header) and the second part (plain text) again. A Level 1 frame is similar to Level 3 in size. However, there is plain text instead of the cipher text in the second part of the frame. Finally, since Level 0 has no security mechanism, the header only contains plain text length and port information. The second part of the frame is the plain text itself. We do not need the third part of the message (MAC) at all. For both security levels, the maximum size is 232 bytes as the size of the *cipher text length* field is 32 bit. Table VI.T4 shows

the increase in size for the order and update messages in the current implementation. Listing D.L1 in Appendix D contains a pseudo-code for the functions that send orders and receive updates in the GRM. Similar functions exist in LRMs to receive orders and send updates.

Table VI.T4: *Resource Management Communication Message Sizes*

Message Type	Original Message Size	Secure Message Size (Level 1 & 2)	
	Bytes	Bytes	Increase
Update or order	4	56	1400%

VI.4 Reconfiguration Strategy for Core Failure

When a core MON instance detects a core failure, the resource management must take recovery action. As explained in the scenario in Section V.3, a LRM initiates the recovery. If there are not enough resources at the node-level to tolerate the failure, then the LRM requests a global reconfiguration from GRM. In this demonstrator, we implemented a recovery procedure based on mode changes (Section II.16) applied by the LRMs via their LRSs. This entails that a set of possible configurations are computed offline with a varying number of cores on a node. A reconfiguration involves moving from one configuration to another via mode changes. The transition between the configurations must be safe. Since we cannot present the strategy using names and parameters of the avionics applications due to confidentiality issues, we explain it with a generalized application model based on the requirements of the avionics applications.

VI.4.1 Application Model

The avionics use case hosts a series of applications $A = \{A_1, A_2, \ldots, A_a\}$. The applications have one of the following two types:

1. Safety critical applications A_c consisting of r_c critical tasks. Each task $\tau_{t,c}$ of A_c has a unique identifier t ranging from 1 to r_c. Thus, we define a critical application A_c as $\{\tau_{t,c} = (C_{t,c}, AET_{t,c}, T_{t,c})\}_{1\leq t\leq r_c}$, where $C_{t,c}$, $AET_{t,c}$, and $T_{t,c}$ are the Worst-case Execution Time (WCET), Average Execution Time (AET) and Period. All tasks are non-preemptible. The system must meet the timing constraints of every critical application.

2. Best-effort (non-critical) applications, which need to achieve a minimum QoS. We define a best-effort application A_j as (U_j, AU_j), where U_j and AU_j are the worst case utilization and average utilization of A_j. In the avionics use case, all best-effort applications are composed of a singe task.

We execute applications as XtratuM partitions. A partition can run in one or more slots (Minor Frame (MiFs)) (Section II.13.1). A slot S_s is defined as $(start(S_s), end(S_s), C_s)$, where $start(S_s)$ and $end(S_s)$ are start and end time of S_s and C_s is the core ID of S_s. Several tasks of an application can execute in a single slot. We must ensure that all tasks of the critical application A_c belonging to the slot S_s fit completely within the slot, i.e,

$$\sum_{\forall \tau_{k,c} \in S_s} C_{k,c} \leq End(S_s) - Start(S_s)$$

An allocation *alloc* is defined as the mapping of applications $\in A$ to one of the three platforms (the DHP and the two T4240). Let $alloc^0$ be the initial allocation of applications to nodes defined by the system designer.

VI.4.2 Local Configuration

Each node has at least one offline defined (ARINC 653 style) schedule, called a local configuration[2]. We define a local configuration LC as consisting of the following elements:

1. MaF with length $|MaF|$. We select a MaF such that the period of every critical application task is either a integer multiple or a factor of the MaF period, i.e., $|MaF| \bmod T_{t,c} = 0$ or $T_{t,c} \bmod |MaF| = 0$ for all tasks of all critical applications.

2. A set of slots distributed over the processors cores and $|MaF|$. If $|MaF| < T_{t,c}$, then the task $\tau_{t,c}$ of a critical application A_c must not be executed every MaF.

3. A mapping between tasks (jobs[3]) of allocated critical applications and the available slots. Since jobs of tasks are unrolled over $|MaF|$, the offline scheduler maps each job $J_{t,k}$ of task τ_t ($\in A_i$) to a slot S_s of a MaF. The offline scheduler also provides the order in which the jobs must be executed in the slot. Each slot only contains jobs of tasks belonging to the same critical application.

4. A mapping between allocated best-effort applications and the available (remaining) slots. The offline scheduler maps each best-effort application A_j to one or more slots $S_{j,1}, S_{j,2}, \ldots, S_{j,s'}$.

We determine both the partition slots and the schedule of tasks inside a slot using an existing offline tool called Xoncrete [216] that is provided with XtratuM. Let LC^0_{DHP}, LC^0_{T42401}, and LC^0_{T42402} be the initial local configuration of the DHP and the two T4240 boards determined using Xoncrete. XtratuM executes these initial configurations upon booting.

[2]Configurations are also commonly referred to as modes. Hypervisors often refer to them as plans.
[3]As a reminder, each instant of a task activation with a specific input data is called a Job.

VI.4.3 Global Configuration

The global configuration GC consists of all the possible combination of local configurations executed by the nodes in the system. GC^0 is the initial global configuration, i.e, $GC^0 = \langle LC^0_{DHP}, LC^0_{T42401}, LC^0_{T42402} \rangle$.

VI.4.4 Reconfiguration Requirements

When we consider core failures, each node can have multiple local configurations, each considering the reduced number of slots available for executing applications (since slots are fixedly distributed over the cores). The number of available slots depends on the number and ID of working cores. XtratuM can store all the configurations as hypervisor plans. We pass all the configurations to XtratuM via the resource management configuration file as illustrated in Listing C.L12 of Section IV.6.1. A node's virtualization domain LRM stores pointers to the local configurations stored by XtratuM.

Multiple global configurations exist as a result of multiple local configurations. The GRM stores the global configurations (via reference to the local configurations) and keeps track of the local configurations via the update messages from the virtualization domain LRMs of the nodes.

There are two important requirements for switching between configurations:

1. As far as resource availability allows, critical applications must be locally reconfigured upon core failure, i.e., the priority of critical applications is greater than the priority of best-effort applications, as discussed in the example scenario of Section V.3. Thus, reconfiguration should first allocate available slots to critical applications on the failed core(s) before allocating them to best-effort applications.

2. All applications that need redeployment must be moved entirely to a different node, i.e., at any point in time, all tasks of an application must run on the same node. Thus, a local configuration with a reduced number of cores should not allocate slots to applications (after considering the first requirement) if they cannot execute entirely under the new core availability.

The virtualization domain LRM sends the currently active local configuration to the GRM via a periodic update message (every MaF). In this case, the LRM sets the field T of the update message format (Figure VI.F8b) to 0 and field *Config* to the current configuration. Naturally, if a LRM switches between local configurations, it sends the new configuration with the next periodic update message.

During the MaF, the LRM collects a list of all failed cores. Then, based on the new core availability, it decides on a possible (local) configuration. If the LRM is unable to host all applications via the new configuration, it sends a reconfiguration request together with the newly selected configuration by setting the type of the update message (field T) to 1. The other field is used as earlier. When the GRM receives the reconfiguration request, it can decide whether to perform a global reconfiguration.

We pass the application set A, the initial global configuration GC^0, all possible core failure combinations to consider, and the requirements stated above as input to a tool

called GREC [171] that was created by ONERA during the DREAMS project. GREC determines local and global reconfiguration graphs based on a heuristic.

VI.4.5 Local Reconfiguration Graphs

A local reconfiguration graph is defined by $\langle L_n, \longrightarrow_n, \dashrightarrow_n, LC_n^0 \rangle$, where

- L_n is the set of all local configurations of a node N_n.
- LC_n^0 is the initial configuration of node N_n.
- $\longrightarrow_n \subseteq L_n \times L_n$ represents the set of local transitions between configurations of node N_n upon failure of a core, i.e, local reconfiguration decisions.
- $\dashrightarrow_n \subseteq L_n \times L_n$ represents the set of transitions between configurations of node N_n due to orders from the GRM upon failures in other nodes ($\neq N_n$), i.e., global reconfiguration decisions.

Let the three nodes – DHP, T4240 1 and T4240 2, be represented by N_0,N_1, and N_2. Then, $\langle L_0, \longrightarrow_0, \dashrightarrow_0, LC_0^0 \rangle$ represents the local reconfiguration graph for the DHP. Similar graphs exist for the other nodes.

VI.4.6 Global Reconfiguration Graph

The global reconfiguration graph is defined by $\langle G, \longrightarrow, \dashrightarrow, GC^0 \rangle$, where

- G is the product of all local configurations from each node, i.e., $G = L_0 \times L_1 \times L_2$.
- $GC^0 = \langle LC_0^0, LC_1^0, LC_2^0 \rangle$
- $\longrightarrow \subseteq G \times G$ represents set of transition between configurations on a node according to the order of the GRM, i.e, global reconfiguration decisions.

  $$(\langle LC_0, LC_1, LC_2 \rangle, \langle LC_0', LC_1', LC_2' \rangle) \in \longrightarrow$$

  such that

  $$\begin{aligned} \exists i \in \{0,1,2\}, (LC_i, LC_i') \in \dashrightarrow_i \wedge \forall i \neq j, LC_j = LC_j' \\ \wedge \exists j, \exists LC_j''(LC_j'', LC_j') \in \longrightarrow_j \end{aligned}$$

  A global decision transition $\longrightarrow$ from $\langle LC_0, LC_1, LC_2 \rangle$ to $\langle LC_0', LC_1', LC_2' \rangle$ can only occur in the Global reconfiguration graph, if there exists at least one node N_i with a transition $\dashrightarrow_i$ from LC_i to LC_i' (corresponding to a global reconfiguration decision in N_i's graph). For all other nodes, N_j, in the system (i.e., $N_j \neq N_i$), there are two possibilities:

  - No transitions occurred, i.e., $LC_j = LC_j'$, or
  - There is a transition $\longrightarrow$ from LC_j'' to LC_j', i.e., a local reconfiguration decision (that does not require global reconfiguration).

- $\dashrightarrow \subseteq G \times G$ represents the set of transitions between configurations made by a node's LRM and received by the GRM in an update message. $\dashrightarrow$ represents a unique local transition.

$$(\langle LC_0, LC_1, LC_2\rangle, \langle LC'_0, LC'_1, LC'_2\rangle) \in \dashrightarrow$$

such that

$$\exists i \in \{0,1,2\}, (LC_i, LC'_i) \in \longrightarrow_i \wedge \forall i \neq j, LC_j = LC'_j$$

i.e., a local decision transition $\dashrightarrow$ from $\langle LC_0, LC_1, LC_2\rangle$ to $\langle LC'_0, LC'_1, LC'_2\rangle$ can only occur in the Global reconfiguration graph, if there exists at least one node N_i with a transition $\longrightarrow_i$ from LC_i to LC'_i (corresponding to a unique local reconfiguration decision without need of global reconfiguration). For all other nodes N_j in the system ($N_j \neq N_i$), no transitions occurred, i.e., $LC_j = LC'_j$.

Durrieu and Pagetti [171] explain the generation of these graphs by GREC in detail. Since the resource management supports multiple core failures, GREC generates symmetric and complete local and global configuration graphs, i.e., for any given combination of failed cores, irrespective of the order of failure, the same configuration must be reached. This implies that the number of failure states in a graph is equal to the number of possible cores failure combinations minus one (the one state where all cores have failed is not meaningful), i.e., $2^{num_cores} - 1$.

VI.4.7 Example of Reconfiguration Graphs

Let us consider an example with $A = \{A_1, A_2, A_3, A_4\}$. To simplify the example, let us assume the N_0 (DHP) cores cannot fail and no applications can be assigned to it. Hence, we do not need to consider N_0 for reconfiguration graphs. LRMs execute on each node, while the GRM executes on N_0 as explained in Figure VI.F4 (on Page 148). To simplify the example further, we consider N_1 and N_2 (T4240s) only have four cores each (instead of 12 cores). Thus, the total number of possible failure states for N_1 and N_2 are $2^4 - 1 = 15$ each. Figure VI.F10 shows these 15 failure states for N_1 (on the left). In addition, there are configurations that occur due to global decisions by the GRM (indicated by $\dashrightarrow$), as a result of core failure on N_2. In the graph, there is only one such configuration (LC_g^{init}), on the right side in the figure. For the graph to be complete, we have all possible configurations from LC_g^{init} onward as well. To make the representation of the graphs simpler, we draw the local reconfiguration graph of N_1 (Figure VI.F10) in a condensed format shown in Figure VI.F11a (note that both graphs are essentially the same, but illustrated differently). Similarly, Figure VI.F11b shows the local reconfiguration graph of node N_2 in a condensed format. The resource management framework takes the local reconfiguration graphs from GREC (via the resource management configuration file) as input and automatically adds it as static C-array to the code of the corresponding LRM. Listing D.L2 in Appendix D illustrates the pseudo-code of this array generated by the resource management framework for the example in Figure VI.F10.

Each local reconfiguration graph has 15 (left) + 15 (right) = 30 states. The global

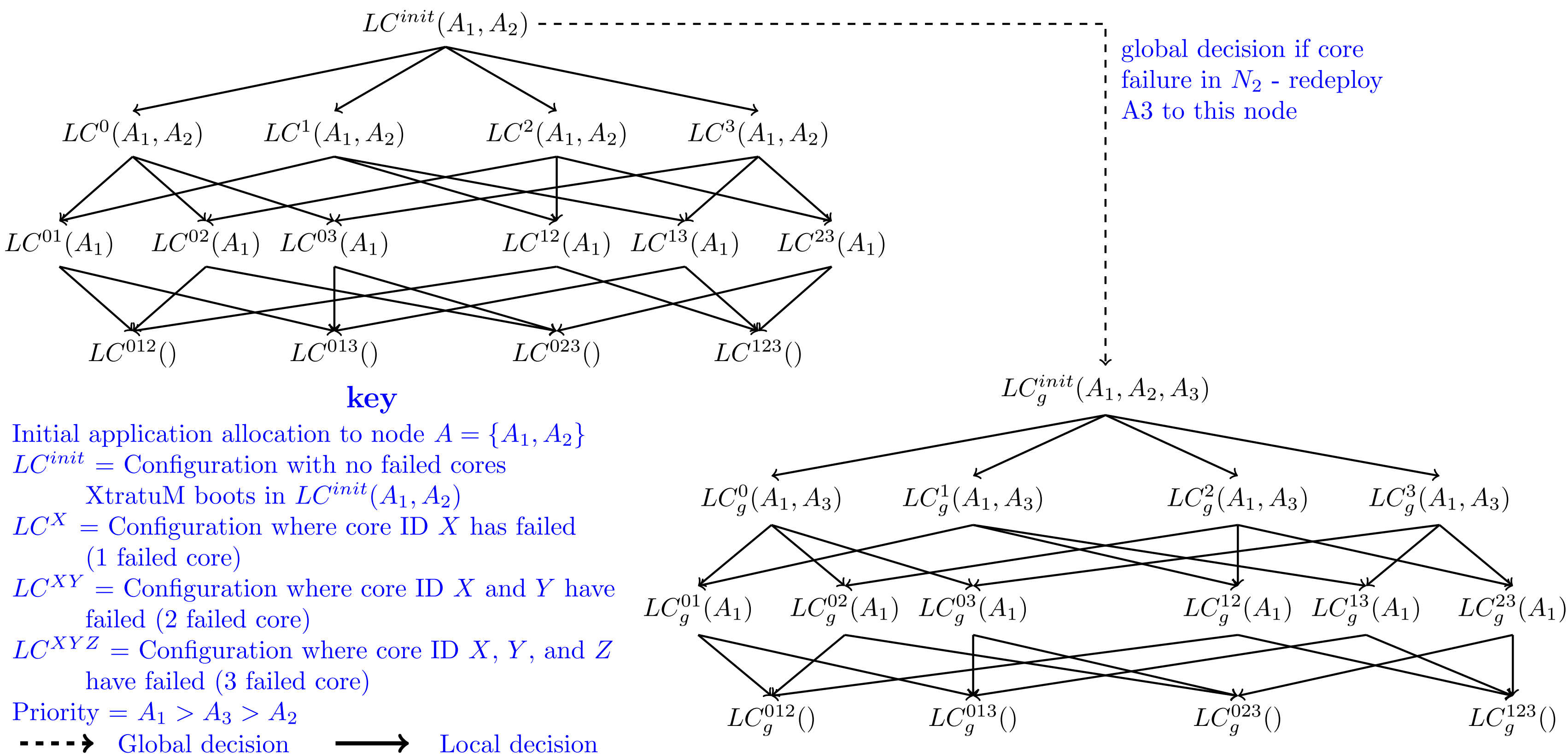
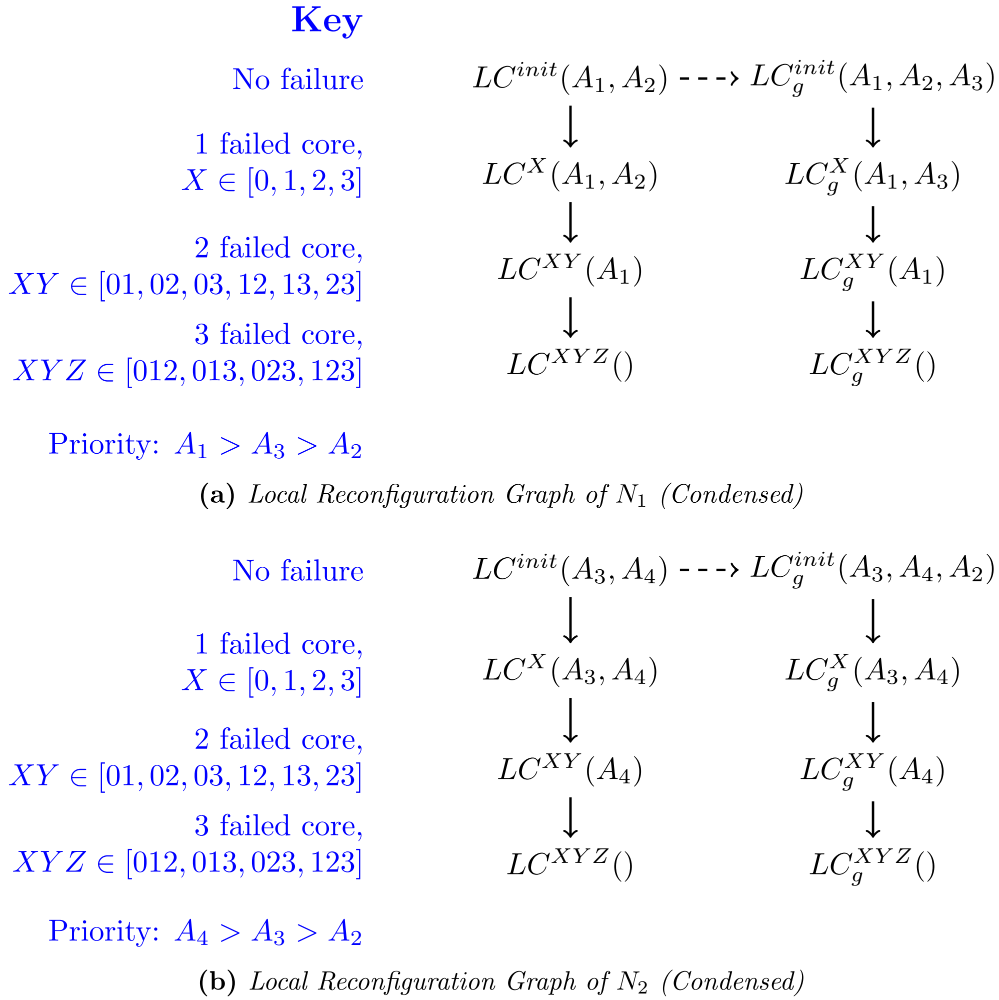


Figure VI.F10: *Example of Local Configuration Graph (Expanded) of Node N_1 (4 cores)*



(a) *Local Reconfiguration Graph of N_1 (Condensed)*

(b) *Local Reconfiguration Graph of N_2 (Condensed)*

Figure VI.F11: *Examples of Local Reconfiguration Graphs (Condensed)*

reconfiguration graph in this scenario can have up to $30 \times 30 = 900$ states in total. However, here we have fewer states as some local reconfigurations also result in a transition in global reconfiguration graphs as shown in Figure VI.F12a. Note that in this figure only $\longrightarrow$ indicate global reconfiguration decisions, while $\dashrightarrow$ are as a result of local reconfiguration by LRM of N_1 and N_2. The GRM implementation only needs to take care of $\longrightarrow$. Since, in the actual use case, the number of $\longrightarrow$ can vary from very low to very high depending on the possible local reconfigurations, we cannot use a static structure to store the global reconfigurations. Instead, the resource management framework takes the global reconfiguration graph from GREC (via the resource management configuration file) as input and automatically generates a C-function. Listing D.L3 in Appendix D illustrates the pseudo-code of this function automatically generated by the resource management framework for example in Figure VI.F12.

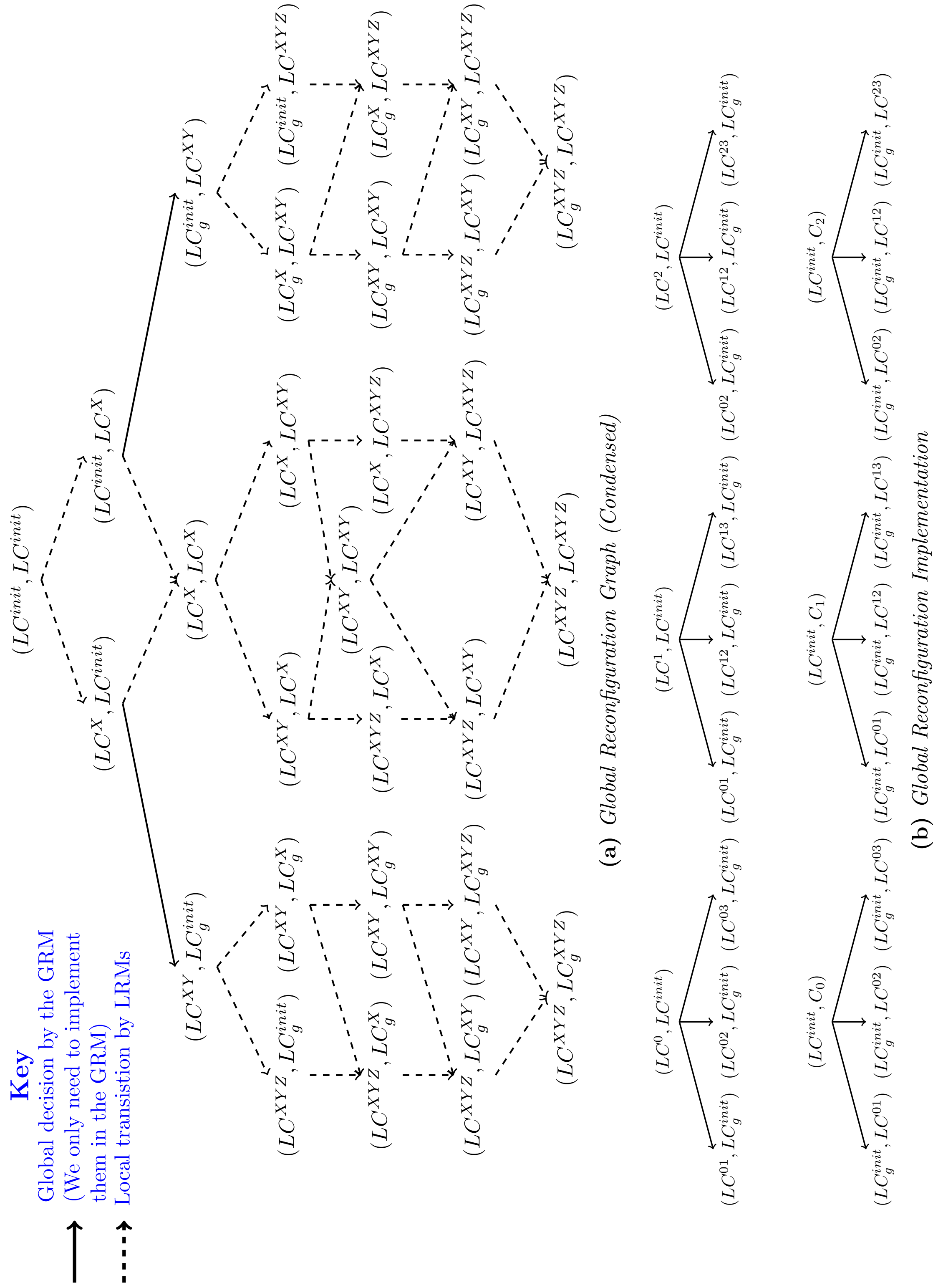


Figure VI.F12: *Example of the Global Reconfiguration Graph*

VI.4.8 Network reconfiguration

At the network level, we need to reconfigure the network components according to the reconfiguration decision of the resource management. Applications communicate to other applications potentially over the network by means of VLs. At runtime, the reconfiguration of an application on a different node requires the redirection of VLs. The computation complexity involved in the construction of new network schedules for TTE at runtime is far beyond the reasonable reaction time required for reconfiguration. Several approaches exist to construct online schedules based on heuristics. However, they are not deterministic and cannot guarantee the feasibility of schedules. Thus, we rely on pre-computed network schedules. For each possible configuration that requires a change in the communication scheme (reallocation of source or destination TTE ports to different nodes), we require an offline defined alternative TTE VL. There are two ways to use the alternative VLs:

1. We can pre-define a *super schedule* containing all VLs based on all possible locations for each application resource. A super schedule is a single schedule combining all possible network schedules required for all global reconfiguration. When a global reconfiguration occurs, it reflects at the network level as a switch from the old VL to a new one, which is already present in the network schedule but not in use until now. The reaction time of such a switch between VLs is negligible since the communication continues from the next scheduled slot of the newly activated VL. Since the GRM never activates two different global configurations simultaneously, the schedule remains conflict-free.

2. We can activate an entirely new pre-defined network schedule in one or several nodes and switches upon global reconfiguration. However, this approach requires the adaptation of internal tables and resetting of internal states that adds a considerable latency to the reconfiguration.

In the avionics use case, we selected the super schedule approach. We were able to create such a super schedule as a result of the modification carried out by TTTech to their offline network scheduling tool, TTPlan, during the DREAMS project.

The three tools (Xoncrete, GREC, and TTPlan) required by the resource management framework for reconfiguration were integrated into a toolchain created during the DREAMS project. This toolchain generated all the user-defined configuration files required for resource management in the avionics use case (Figure VI.F13).

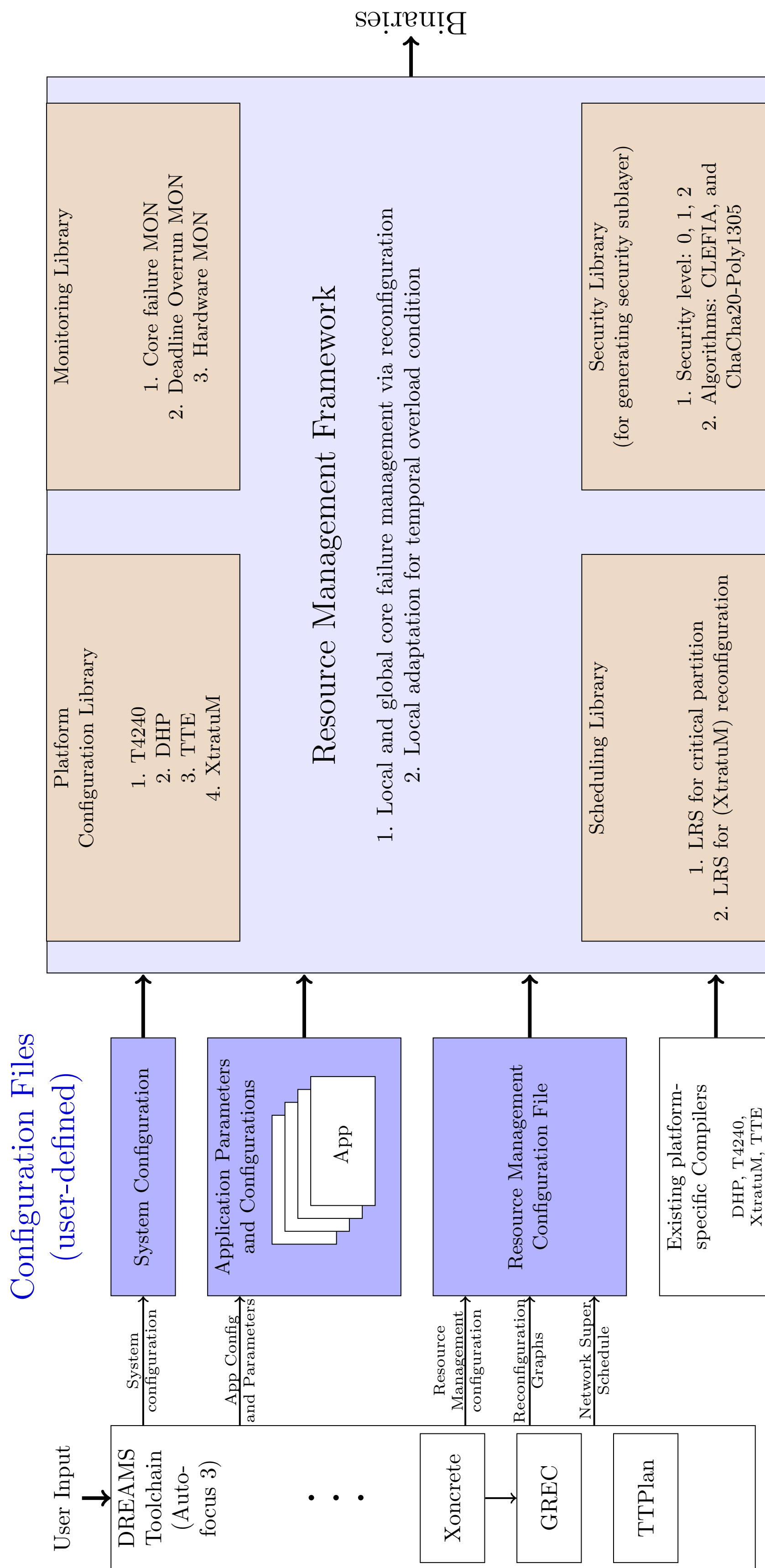


Figure VI.F13: *Configuration of Resource Management Components in Avionics Use Case*

VI.5 Resource Management Evaluation

We set up resource management for the evaluation using the user-defined configuration files generated via the DREAMS toolchain. Examples of such configuration files are presented in Listing A.L3 of Appendix A. We performed the evaluation for determining global reconfiguration delay and validate the local and global core failure management by the GRM and the LRMs. We also determined the overheads of the resource management components involved in the core failure management. Next, an evaluation for temporal overload management by an LRM is presented. Finally, we performed an experimental evaluation to determine the overheads for secure resource management communication.

VI.5.1 Evaluation of Local and Global Core Failure Management

We successfully validated the resource management in the avionics use case by simulating core failures in a T4240 node. For reducing the executable file generation and the fault injection demonstration, we used the scenario from the example in Section VI.4.7 where only four cores (Core 0 to 3) were activated on each T4240 node. The LRMs execute on both nodes synchronously as XtratuM partitions towards the end of each MaF as described earlier. The GRM executes as a XtratuM partition on the DHP. In the initial configuration (MaF 0, Plan 0), all three avionics applications (SDP, FMS, DMS) are hosted on a T4240 node and MPEG server is hosted on the DHP.

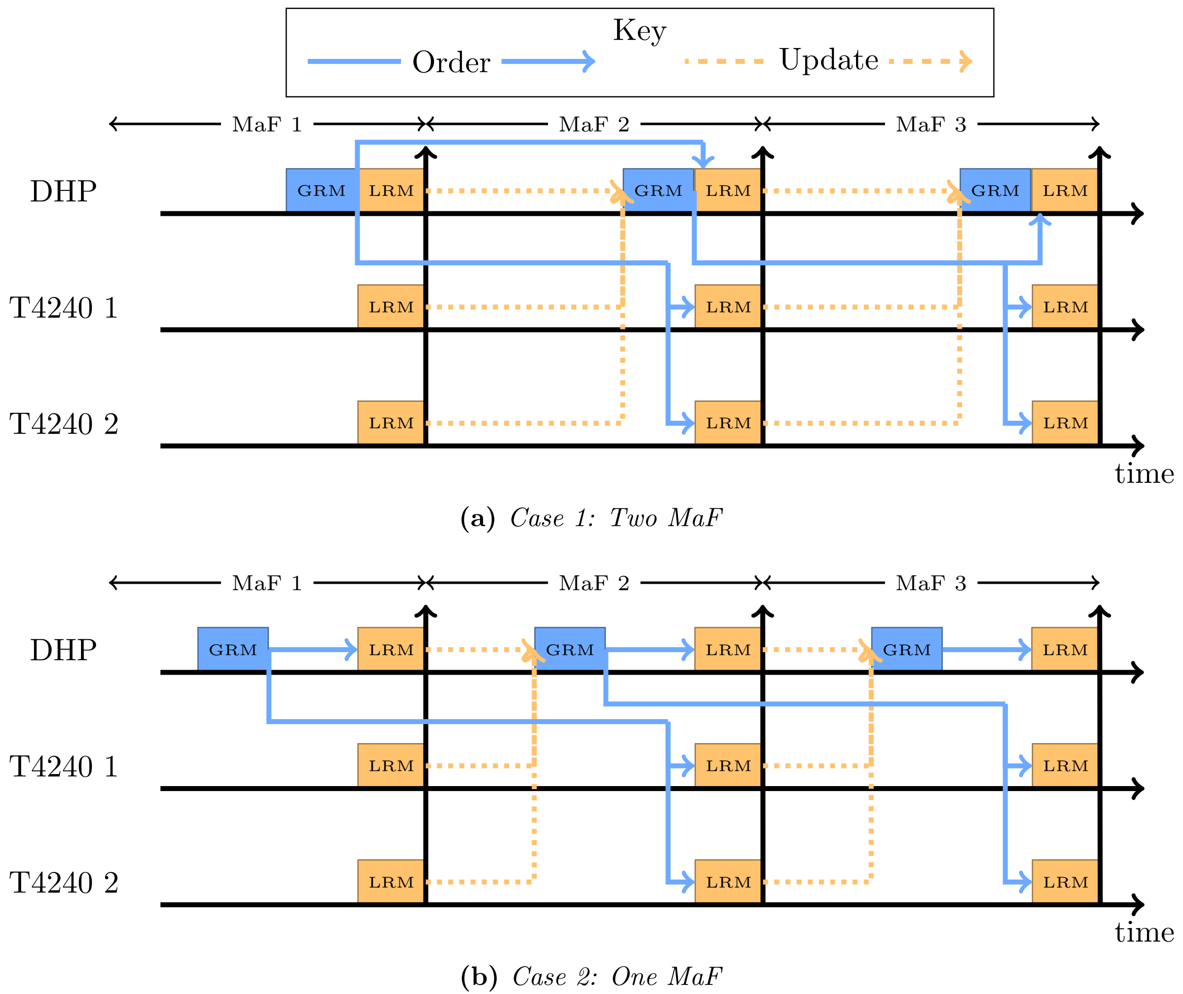
Global Reconfiguration Delay

We observed that the overall delay for global reconfiguration depends on the allocation of the GRM slots on the DHP in relation to the slots of LRMs on all the nodes. Furthermore, the delay also depends on the network schedule. As a reminder, update and order messages are sent over TT VLs. Thus, we can guarantee when the message reaches the destination by planning VLs according to requirements. Moreover, the avionics use case requires that reconfigurations only take place at end of the MaF. Therefore, the order message immediate field (I) is always set to deferred (0) by the GRM. Hence, nodes change their configuration according to the GRM orders only at the end of a MaF.

Case	GRM Slot location	TT VL period		Reconfiguration delay[4]
		Update channel	Order channel	
1.	Occurs directly before the slots of LRMs	$\leq \lvert MaF\rvert - Start(GRMSlot)$	$\leq \lvert MaF\rvert$	2 MaF (Figure VI.F14a)
2.	Occurs betwen the slots of LRMs (in different MaF)	$\leq \lvert MaF\rvert - Start(GRMSlot)$	$\leq End(GRMSlot) - Start(LRMSlot)$	1 MaF (Figure VI.F14b)

Table VI.T5: *Global Reconfiguration Delay*

[4]After core failure detection by an LRM



(a) *Case 1: Two MaF*

(b) *Case 2: One MaF*

Figure VI.F14: *Global Reconfiguration Delay*

There are two cases with different global reconfiguration delays as shown in Figure VI.F14. In the first case (Figure VI.F14a), when an LRM instance detects a core failure (at end of MaF 1 in the figure) and requires a global reconfiguration, it sends an update message to the GRM. The GRM instance of the next MaF (MaF 2 in the figure) receives this update. This GRM instance will, in turn, send orders to the instance(s) of the LRM(s) belonging to the following MaF (MaF 3 in the figure). The LRM belonging to the following MaF applies the reconfiguration according to the GRM order. As the reconfiguration takes place at the end of MaF, the total delay for global reconfiguration is two MaFs after a LRM detected a core failure. The main benefit here is that resource management communication has to occur less often, requiring less network bandwidth. However, the drawback is a longer reconfiguration delay.

In the second case (Figure VI.F14b), when an LRM instance detects a core failure (MaF 1 in the figure) and requires a global reconfiguration, it sends an update message. The GRM instance of the following MaF (MaF 2 in the figure) receives this update. This GRM instance will, in turn, send orders to the instance(s) of the LRM(s) belonging to the same MaF. Then, the LRM(s) will apply the reconfiguration according to the

order from the GRM at end of the MaF. As the reconfiguration takes place at the end of MaF, the total delay for global reconfiguration is one MaFs after a LRM detected a core failure. The disadvantage here is that resource management communication has to occur more often, requiring more network bandwidth. However, the advantage is a shorter reconfiguration delay.

In both cases, the delay for global reconfiguration is deterministic. Since the network bandwidth was not an issue in the avionics use case, we selected the second option as the reconfiguration delay is shorter (1 MaF).

Experimental Validation

Table VI.T6 shows an example where core 3, core 2 and core 1 of a T4240 (T4240 1) fail in MaF 5, MaF 6, and MaF 7 respectively. We repeated the experiment 1000 times, but with different order and time of core failure. We simulated core failure in three out of four T4240 node cores in each case.

The evaluation confirmed that every core failure is correctly detected by the virtualization domain LRM via the core failure MON. As expected, the LRM selected a new configuration from the local reconfiguration graph (Plan 2, Plan 5, and Plan 12 on failure of Core 3, Core 2, and Core 1 respectively in Table VI.T6). The LRM informed the GRM upon local reconfiguration with update messages. Until the failure of two cores (core 3 and core 2 in the table), the LRM always found a local configuration that can accommodate all four applications on the same node. Upon failure of the third core (core 1 in the table), the LRM found a configuration that hosted the three critical applications; for example, in Table VI.T6, core 0 hosts SDP, FMS, and DMS. The LRM correctly informed the GRM about the local reconfiguration and requested for a global reconfiguration to host the MPEG server on another node. As expected, the GRM found a new global configuration for the DHP that hosted MPEG server and informed the virtualization domain LRM of the DHP via an order message. Each time we observed that the DHP configuration changed at end of the MaF according to the GRM order.

Resource Management Overheads

The GRM, LRMs and core failure MON introduce run time overhead. Each of them requires one slot to execute. During the slots for GRM, LRMs and MONs, we do not allocate any application partition slots in parallel to avoid interference. GRM and LRM overheads are not only caused by the resource management logic but also the updates and orders communication. In addition, the LRM must reconfigure the system (via the LRS for reconfiguration). The MONs have the least overhead. We measured the overheads using hardware MONs (Performance Monitor Counter (PMC) configured for counting CPU clock cycles). Table VI.T7 presents the maximum observed overheads for the components on the DHP and the T4240. The overhead of the LRM includes the time consumed by the LRS for reconfiguration as the LRS is called inside the LRM. As the GRM only executes on the DHP, there is no GRM overhead on the T4240.

MaF	DHP			T4240 1					Description
	Config-uration	Core ID[5]		Config-uration	Core ID[5]				
		0	1		0	1	2	3	
0	0	✓	✓	0	✓	✓	✓	✓	Initialization configuration; Move to next configuration after initialization completion
1-4	1	✓	✓	1	✓	✓	✓	✓	Nominal configuration - SDP, FMS, DMS, MPEG server are on T4240
5	1	✓	✓	1	✓	✓	✓	✗	Core 4 failure detected; Local reconfiguration at end of MaF and update to GRM
6	1	✓	✓	2	✓	✓	✗	✗	Core 3 failure detected; Local reconfiguration at end of MaF and update to GRM
7	1	✓	✓	5	✓	✗	✗	✗	Core 2 failure detected; Local reconfiguration at end of MaF and reconfiguration request to GRM
8	1	✓	✓	12	✓	✗	✗	✗	GRM received reconfiguration request. GRM searches for new global configuration. GRM sends order to involved LRM (new configuration hosts the MPEG Server on the DHP). LRM on DHP receives order and applies order at end of the MaF.
9	4	✓	✓	12	✓	✗	✗	✗	New configuration on DHP as a result of global reconfiguration

Table VI.T6: *Example of Local and Global Reconfiguration in Avionics Use Case*
[5]✗= failed core, ✓= active core

Component	MON	LRM	GRM
DHP	$80\mu s$	$1.9ms$	$1.5ms$
T4240	$20\mu s$	$900\mu s$	-

Table VI.T7: *Resource Management Overhead*

VI.5.2 Evaluation of Local Temporal Overload Management

Thales R&T evaluated local temporal overload (potential deadline overrun) management due to confidentiality issues in providing the complete code of the avionics applications. We summarize the evaluation here. The evaluation was done on a T4240 using only four cores. The three critical avionics application [6] (A_1, A_2, A_3) are allocated to core 0. Four stressing benchmarks were added as non-critical applications: Write sequential (ws), write random (wr), read sequential (rs), and read random (rr). Partitions of the non-critical applications were assigned $8MB$ array (>Last Level Cache (LLC) size). Each non-critical application accessed memory with 64bytes steps (=LLC line size). This ensured cache misses and stressed the memory.

Three deployment scenarios were considered:

1. *Isolation*: Non-critical applications are deployed on core 1 to 3 and allocated in time windows that do not coincide with critical application slots.

2. *Interference*: Non-critical applications are deployed on all four cores. Non-critical applications are allocated on core 0 in time windows that do not coincide with critical application slots. On core 1 to 3, non-critical applications use the complete MaF.

3. *Deadline overrun*: This scenario is the same as the second one, but with potential deadline overrun MONs and application domain LRMs to manage temporal overload.

Table VI.T8 shows the slot deadlines (D) per critical application. To compute the slot deadline (D) of an application in the isolation scenario, the worst case observed execution time of all tasks (from any run) are added. For example, suppose a critical application A_i has three tasks, $\tau_{1,i}, \tau_{2,i}$, and $\tau_{3,i}$. Worst case observed execution time $C_{1,i}$ for $\tau_{1,i}$ occurs in the first run, $C_{2,i}^{iso}$ for $\tau_{2,i}^{iso}$ occurs in the tenth run and $C_{3,i}^{iso}$ for $\tau_{3,i}$ occurs in the fiftieth run. In this case, the resulting deadline $D_i^{iso} = C_{1,i}^{iso} + C_{2,i}^{iso} + C_{3,i}^{iso}$. This D_i^{iso} is considered as 100%. Values are presented as percentages as actual values cannot be provided due to confidentiality issue.

For the interference scenario, the isolation deadline is multiplied by a slowdown factor to get the new deadline. The slowdown factor is the worst-case observed slowdown of any task of any critical application divided by the worst-case observed time for the same task in an isolation scenario. For example, suppose a task, $\tau_{1,j}$ of critical application A_j suffered the worst slow down from all tasks of all critical applications in the interference scenario with the worst-case observed execution time $C_{1,j}^{int}$. The same task has a worst-case observed execution time $C_{1,j}^{iso}$ in the isolation scenario. In this case, the slowdown factor is calculated as $S = C_{1,j}^{int}/C_{1,j}^{iso}$ and the slot deadline as $D_j^{int} = S \times D_j^{iso}$. The same value of S is used for all applications. The actual value of S was determined to be 4.036. Table VI.T8 presents the values as percentage of the slot deadline in isolation.

For the deadline overrun scenario, the slot deadline (D) of an application is the sum of the worst-case observed execution time of all application tasks in isolation, plus the

[6] Application names anonymized due to confidentiality issue

Application	Benchmark	Isolation (%)			Interference (%)			Deadline Overrun (%)		
		D	C	MET	D	C	MET	D	C	MET
A_1	ws	100	85.6	79.7	403.6	342	238.1	247.5	104.1	99.6
	wr		86	79.7		175.1	165.2		96.7	90.4
	rs		97.6	96.6		104.1	102.8		107.2	106
	rr		97.7	96.7		104.5	103.5		107.2	106.1
A_2	ws	100	39.6	37.9	403.6	79.2	69.1	240.3	75.3	68.5
	wr		39.8	37.9		58.7	55.9		58.5	55.6
	rs		51	50.9		55.3	54.8		58.5	57.9
	rr		51.1	50.9		55.2	54.9		58.2	57.9
A_3	ws	100	93.6	93	403.6	152.2	139.9	258.9	98.5	97.3
	wr		93.7	93		113	111.9		98.3	95.5
	rs		97.2	97.1		102.1	102		98.7	98.3
	rr		97.1	97.1		102.5	102.2		98.4	98.3

Table VI.T8: *Evaluation of Local Temporal Overload Management*

largest of the application's tasks worst observed execution time minus the execution time of the same task in the isolation scenario. For example, suppose a critical application A_k has an isolation deadline D_k^{iso}. Let $C_{1,k}^{do}$ be the largest observed execution time of a task from all tasks $\in A_k$ in the deadline overrun scenario. The slot deadline of A_k in this scenario is $D_k^{do} = D_k^{iso} + C_{1,k}^{do} - C_{1,k}^{iso}$. Table VI.T8 presents the values in percentage of the slot deadline in isolation.

As observed in the table, the local resource management in the deadline overrun scenario helped to reduce slot deadline significantly for all three critical applications $(247.5\%, 240.3\%,$ and $258.9\%)$ compared to the interference scenario $(403.6\%, 403.6\%,$ and $403.6\%)$.

In addition, Table VI.T8 also presents the observed worst-case execution time (C) and median execution time (MET) as percentage of the application deadlines when running in the isolation scenario. The values are per deployment scenario and considering non-critical applications with one of the four stressing benchmarks at a time. The most significant reductions are observed in C and MET of A_1 (with ws benchmark) in the deadline overrun scenario (104.1% and 99.6%) as compared to the same in interference scenario (342% and 238.1%). In some situations of the deadline overrun scenario (e.g., A_1 and A_2 values for rs and rr), the C and MET values are slightly higher than the interference scenario. However, the slowdown is tiny compared to the speedup in other situations, and the main advantage lies in having smaller slot deadlines. In all runs of the deadline overrun scenario, the local resource management ensured that C remains below the slot deadline, i.e., the maximum observed execution time (C) in the table is less than the deadline (D).

Table VI.T9 shows the performance obtained by the four non-critical applications (each running a different benchmark) in deadline overrun scenario compared to the

Benchmark	ws	wr	rs	rr
performance	87%	93%	99%	92%

Table VI.T9: *Performance of Non-Critical Applications in Deadline Overrun Scenario*

performance in the interference scenario. The values in the table are calculated as:

$$\frac{I^{do} - I^{iso}}{I^{int} - I^{iso}} \times 100$$

where I^{do}, I^{iso}, and I^{int} are the amount of memory access iterations by the non-critical applications in a fixed total number of MaF in the deadline overrun, isolation, and interference scenarios.

Thales R&T evaluated the solution that they proposed for improving QoS of non-critical applications to reduce the impact of deadline overrun management by the LRM (second solution in Section IV.9.2). The evaluation can be found in [7].

VI.5.3 Evaluation of Secure Resource Management Communication

In this section, we present the experimental evaluation of the time taken by the GRM to send orders (SO) and receive updates (RU), and the LRMs to receive orders (RO) and send updates (SU). We need to consider two important instances of resource management communication for the evaluation:

1. Communication between GRM and virtualization domain LRM on the DHP (see Figure VI.F4 on Page 148 for reference): The GRM and the LRM read/write messages to/from the XtratuM sampling and queuing ports and incur an overhead in the process. The message is available to the GRM and LRM as soon as the GRM or LRM partition time slot occurs. Hence, the only real overhead is sending/receiving messages to/from the XtratuM ports.

2. Communication between GRM on the DHP and the virtualization domain LRMs on two T4240 (see Figure VI.F4 on Page 148 for reference): The GRM and the LRMs read/write messages to/from the XtratuM sampling and queuing ports (for TTE) and incur an overhead in the process. Once the message is written to the ports, it is sent to the destination node as a time-triggered message based on an offline-defined network schedule. The time for delivery of message via the TTE depends on the TT VL period. The only unknown overhead is sending/receiving messages to/from the XtratuM TTE ports.

Thus, we evaluated the time for sending/receiving messages to/from the XtratuM (TTE) ports. All messages pass through the security sublayer regardless of which security level is selected. Although security Level 0 does not provide any security services, it still adds some headers to the messages as explained in Section VI.3.5. Security levels 1 and 2 have a message size of 56 bytes.

Message Type	Security Level	Security Algorithm	
		ChaCha20-Poly1305 (ns)	CLEFIA (ns)
GRM - Receive Update (sampling port)	0	37.97	
	1	167.45	1564.68
	2	216.01	2295.86
LRM - Receive Order (queuing port)	0	42.3	
	1	163.19	1551.29
	2	216.50	2264.88
LRM - Send Update (sampling port)	0	37.25	
	1	165.79	1562.14
	2	223.93	2282.22
GRM - Send Order (Queuing port)	0	29.19	
	1	135.97	1534.49
	2	187.29	2255.97

Table VI.T10: *DHP – Maximum Observed Time for Resource Management Communication*

Message Type	Security Level	Security Algorithm	
		ChaCha20-Poly1305 (ns)	CLEFIA (ns)
GRM - Receive Update (sampling port)	0	13.73	
	1	54.82	1029.93
	2	72.21	1583.21
LRM - Receive Order (queuing port)	0	21.32	
	1	61.32	1052.44
	2	78.46	1511.99
LRM - Send Update (sampling port)	0	13.04	
	1	51.01	933.58
	2	67.32	1630.08
GRM - Send Order (Queuing port)	0	13.30	
	1	52.07	1029.32
	2	68.68	1557.50

Table VI.T11: *T4240 – Maximum Observed Time for Resource Management Communication*

We used hardware MONs (PMC set up for counting clock cycles) on both DHP and T4240 to measure the time for reading/writing to/from the XtratuM (TTE) ports accounting for all three security levels (Level 0, 1, and 2) and both security algorithms (CLEFIA and Chacha20-Poly1305). Table VI.T10 and Table VI.T11 show the observed maximum time (in nanoseconds) needed for sending and receiving messages (including security sublayer overhead) for 1000 runs each on the DHP and a T4240. These values are useful in determining the worst-case execution time of the GRM and the LRMs.

In Figure VI.F15, we further break down the observed average values for DHP and T4240 into two components: security sublayer overhead and overhead for reading/writing XtratuM ports. The evaluation shows that the time needed for reading/writing update messages is higher than for order messages. This difference arises from using different port types for order (sampling) and update (queuing) channels.

Overhead for security level 0 is minimal as the messages only pass through the security sublayer without further processing for security purposes. However, security levels 1 and 2 have a more considerable overhead for reading/writing messages due to their larger message size. Moreover, security level 2 has more overhead than security level 1 as it adds confidentiality on top of authenticity and integrity.

Furthermore, the evaluation also indicates that the ChaCha20-Poly1305 algorithm is much faster than the CLEFIA algorithm (in the OCB mode of operation). This is because the stream cipher design of ChaCha20 allows for faster encryption and decryption. However, stream ciphers, such as ChaCha20, are generally used only in few applications and, thus, not closely scrutinized regarding security. Contrarily, block ciphers, such as CLEFIA, are heavily used in many applications, and thus, are more trusted. In the end, the trade-off between using ChaCha20-Poly1305 and CLEFIA lies in speed versus trust in the algorithm.

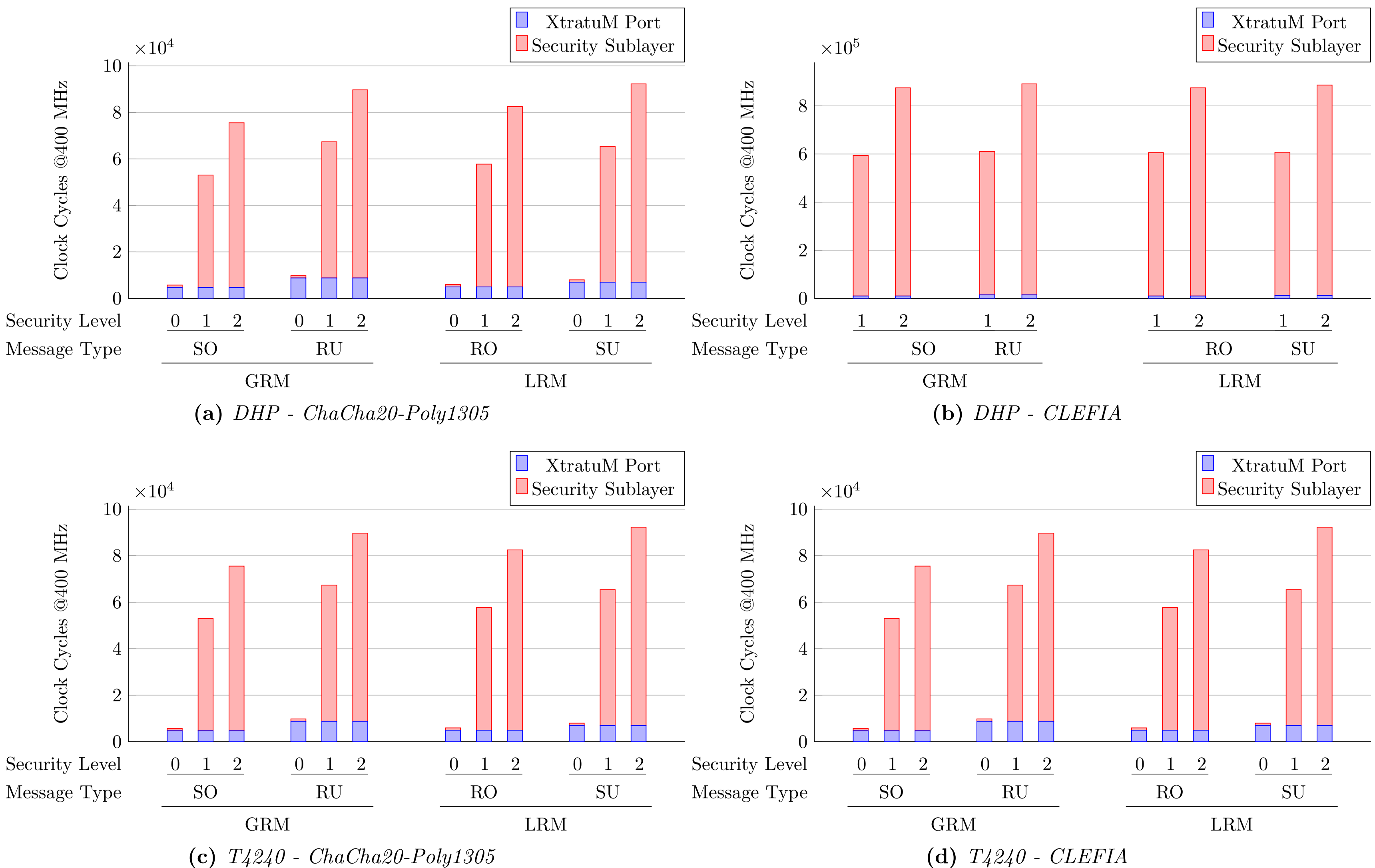


Figure VI.F15: *Resource Management Communication Overheads – DHP and T4240*

VII

Railway Use Case: Real-Time Cloud via Resource Management

"Today, companies have to radically revolutionize themselves every few years just to stay relevant. That's because technology and internet have transformed the business landscape forever."

– Nolan Bushnell

In this chapter, we present an existing real-time safety-critical railway use case from the EU SECREDAS project [154] and explore virtualization technologies and cloud computing for migrating this use case from dedicated hardware solutions. We examine existing virtualization technologies (hypervisors) for deploying a (private) Cloud on Commercial-Off-The-Shelf (COTS) server hardware to run the use case while meeting stringent safety requirements. We base our migration on qualitative and quantitative benchmarking of the relevant hypervisors for specific railway requirements. Based on the insights gained, we provide suggestions using an existing hypervisor with new RT-cloud components to safely and securely run the railway use-case applications. The new components include a resource management layer for the cloud based on our resource management framework (Chapter III). Finally, we present an evaluation for the Time-Triggered (TT)-Local Resource Scheduler (LRS), initial experiments with Intel Memory Bandwidth Allocation (MBA) and the evaluation of the distributed global resource management.

VII.1 Railway Use Case

The railway domain has high reliability and availability requirements given by the CENELEC standards ([155, 156, 157]) to certify safety-critical railway applications. The system has a long lifespan ($\geq$ 25 years). Therefore, any change to the system should not compromise the safety requirements. In addition, railway components require redundancy to remove single points of failure. Multiple hardware components are required that can host replicas of the same applications for safe operation. These components require high availability and reliability to ensure continuous operation. However, building systems with high availability and reliability incurs high costs.

At present, railway domain applications are hosted on specialized dedicated hardware platforms using federated architectures. The main benefit of using this architecture is fault containment. However, with the increase in demand, many nodes are required to host more safety-critical applications. Thus, the use of federated architecture has led to an increase in Size, Weight and Power (SWaP), required wiring, and the associated costs in the railway domain. Moreover, when there is a failure in a system with specialized hardware, it is challenging to replace obsolete specialized platforms that are no longer manufactured. Such situations result in prolonged downtime. Systems with obsolete components begin to generate more failures which, in turn, require more time to remedy. Eventually, a situation can arise where components cannot be fixed, or there is enormous downtime. As a result, hardware obsolescence is a significant issue for the railway domain. Obsolescence is a reality that any safety-critical system will eventually encounter.

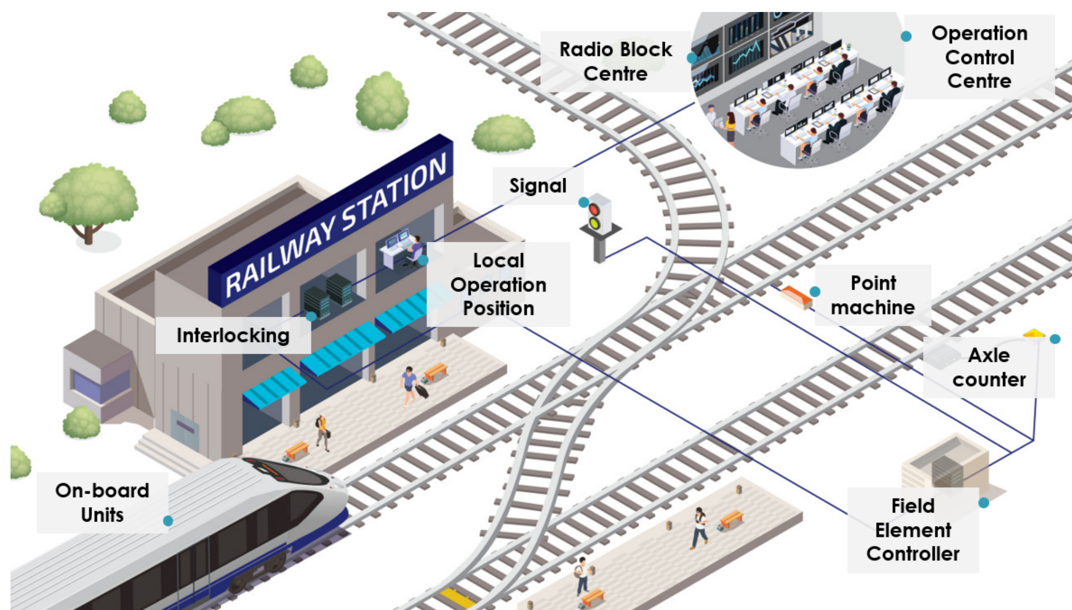


Figure VII.F1: *Layered railway domain [5]*

Figure VII.F1 illustrates the different layers and complexity involved in the railway domain. There exist numerous indoor and outdoor components. Indoor components control several interlocking simultaneously and operate the ever-growing railway networks. Radio Block Center (RBC) is an example of an indoor component that is responsible for the radio communication towards the European Train Control System (ETCS)-L2 operated trains. As a result of increasing high-speed lines and the growth of railway networks, such indoor components have widely increased recently. Such an increase has led to the demand for a more scalable architecture in the railway domain. However, relying on a limited amount of dedicated specialized hardware platforms hinders scalability.

Cloud computing principles provide a lucrative option for the railway domain to deal with the earlier mentioned issues. Clouds abstract the underlying hardware from applications by use of a virtualization technology. Clouds provide a virtualized equivalent of a computer system, i.e., a Virtual Machine (VM), to host applications. The applications

are unaware that they are running in a virtualized environment or know the underlying hardware resources. Thus, the reuse of existing applications without modifications is possible by directly executing them in a VM. As a result, clouds allow easier reuse of legacy railway applications and help overcome hardware obsolescence issues of the railway domain.

Cloud computing also provides high availability by creating clusters. The main idea is that a group of servers that appear as a single server to the applications provide continuous up-time. Thus, if a railway application is hosted on a cloud when one server is unavailable, the other servers can replace it and provide continuity of service.

In the current on-premise hardware setup used in the railway domain, it is essential to shut down the hardware to add new resources. However, when the hardware requirements increase in a cloud, it is possible to add resources without interrupting the existing applications seamlessly. Similarly, if the demand reduces, it is possible to downscale the system seamlessly. The railway domain can benefit from the cloud to support the increase in high-speed lines and the growth of railway networks over time. The cloud can ensure the required long railway application lifetime through continuous operation during updates. Moreover, it is possible to dynamically increase or decrease the available hardware resources to support temporary fluctuations in computing requirements; for example, to support increased requirements during peak hours.

Clouds help increase resource utilization and reduce SWaP and wiring costs by hosting multiple VMs concurrently on the same multicore node (hardware consolidation). In turn, hardware consolidation results in a reduction in maintenance cost and fewer spare parts. Clouds provide basic mechanisms for improved resource sharing and help to allocate resources efficiently among various VMs (via hypervisors). Moreover, clouds can partition some of the underlying hardware, such as CPU and memory space, and provide some form of isolation among the concurrently executing VMs. Such segregation can form the basis for permitting railway applications of various criticality levels and other non-critical applications in the cloud.

In general, hosting applications using cloud computing principles supports ease of re-usability, reconfiguration, and scalability, while providing higher availability, less need for maintenance, and reduction in running costs. Moreover, cloud computing helps organizations reduce their carbon footprint by letting them reduce resource over-provisioning.

Despite the advantages provided by clouds for the railway domain, there are some hindrances. Multiple VMs run on each node in a cloud and share the underlying node resources. Cloud virtualization environments (e.g., Xen or Kernel Virtual Machine (KVM) hypervisors) allow partitioning off some of these shared resources to each VM. For example, it is possible to use classic scheduling techniques (e.g., Round-robin, Earliest Deadline First (EDF)) to provide a constant amount of computing time to each VM or spatially divide memory space among VMs. However, most cloud hypervisors do not support partitioning other shared resources, such as memory bandwidth or network bandwidth.

As discussed in Section II.18, there are two types of cloud: public and private. In a public cloud, the control of the entire cloud lies with the cloud service provider, who do

not guarantee resource partitioning or precise resource allocation to VMs. However, to achieve (timing) predictability, railway applications require careful allocation of resources and isolation among VMs. As a result, the railway VMs cannot adhere to safety standards and certification requirements if hosted on a public cloud.

Contrarily, in a private cloud, complete control over the cloud resources is possible. We can adapt private clouds with our resource management techniques to partition/allocate crucial resources and ensure all railway applications meet their safety-assurance levels. Besides, our resource management can monitor the cloud to detect resource failures and non-conformant behavior of railway VMs. We envisage such private Real-time clouds (RT-clouds) in the near future.

Since virtualization is at the heart of cloud computing, the choice of a suitable virtualization technology is essential for ensuring the performance, predictability, availability, and safety required for the railway domain. Since virtualization is a new concept to the railway domain, limited research has examined cloud virtualization techniques for use in the railway domain. Resch [158] presented an initial step towards this goal. We take a step further by examining virtualization technologies and cloud computing on COTS server hardware for migrating an existing real-time safety-critical railway use case from dedicated hardware solutions.

VII.1.1 TAS Control Platform

Railway operators widely use indoor and outdoor computer-based railway control systems, such as electronic-interlocking systems, axle counters, and automatic onboard train control systems. The components are used in both the mainline and urban rail markets. Although these applications share similar stringent security and safety requirements, their performance assumptions, operating conditions, and cost requirements are different.

Thales Austria GmbH developed the TAS Control Platform [129]. It is a computing platform that meets these requirements and serves a common base with fault-tolerance and reliability services for railway applications to build on. Thales has certified TAS according to Safety Integrity Level (SIL) 4 requirements defined in the CENELEC standards ([155, 156, 157]). As illustrated in Figure VII.F2, TAS Control Platform functions on top of specialized hardware and contains different layers. The bottom layer is based on a Linux-based operating system that provides basic services such as file management, network protocols, and task scheduling. The top layer represents the functional aspects of the system: the railway-specific applications. Lastly, the middle layer provides the essential fault tolerance and reliability service required by the railway applications together with an implementation of an API for use by the applications. This layer acts as a bridge between the programming models and the operating system.

This use case requires us to move the indoor safety-critical railway applications from Figure VII.F1 (e.g., RBCs) to a private cloud running on servers consisting of 2nd Generation Intel Xeon Scalable processors [30]. Such a cloud-based approach using COTS hardware has multiple advantages over the current deployment of TAS control platform on dedicated hardware.

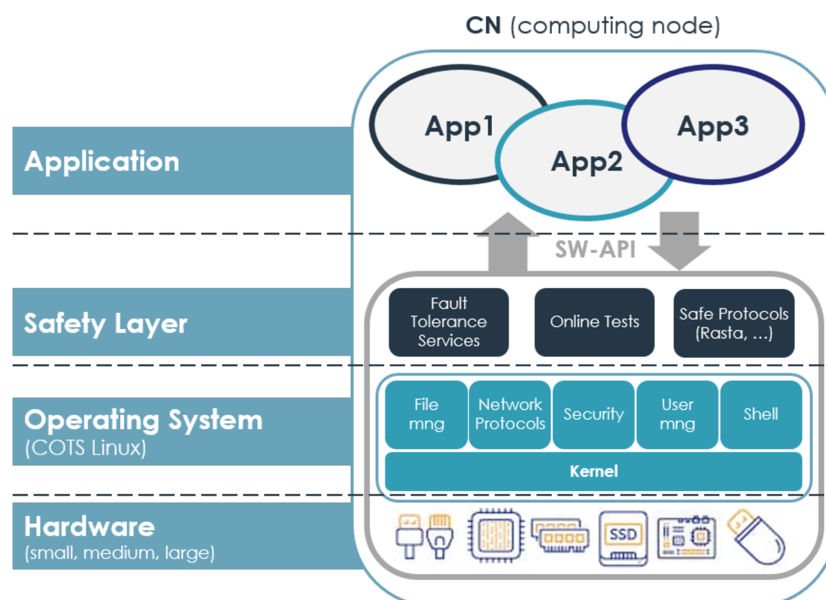


Figure VII.F2: *Layered Architecture of the* TAS Control Platform *[5]*

- This move will enable the railway industry to eradicate the dedicated hardware components near tracks.
- Virtualization will help to move away from custom hardware solutions by making the TAS control platform independent of the underlying hardware.
- Virtualization will also eliminate or reduce re-certification costs upon a change in the underlying hardware.
- The cloud will allow hardware consolidation by allowing multiple TAS control platform VMs on a single node, and thus, promote efficient utilization of resources.
- Cloud will improve scalability, ease of maintainability, and thus, lower operating costs while providing higher availability.

VII.1.2 TAS Control Platform in Cloud

Figure VII.F3 illustrates the architectural decisions made to run the TAS control platform in a cloud environment. We can identify two major components that are novel compared to the traditional approach of running the TAS Control Platform on dedicated hardware. Firstly, a hypervisor can abstract the COTS hardware from the virtualized safety-critical applications. In addition, the hypervisor can provide some degree of separation and isolation. Secondly, When running a TAS Control Platform on dedicated embedded devices, the network between these systems was dedicated and exclusive to the TAS Control Platform and the safety-critical applications it hosted. However, introducing virtualization and resource sharing makes the network interconnection between different TAS Control Platform instances a shared resource. Therefore, besides virtualizing the

TAS Control Platform, we also need to employ a network virtualization technology to separate the traffic from different virtual instances while still transporting it over the same shared physical medium. In this dissertation, we investigate suitable cloud hypervisor technologies to host the TAS control platform.

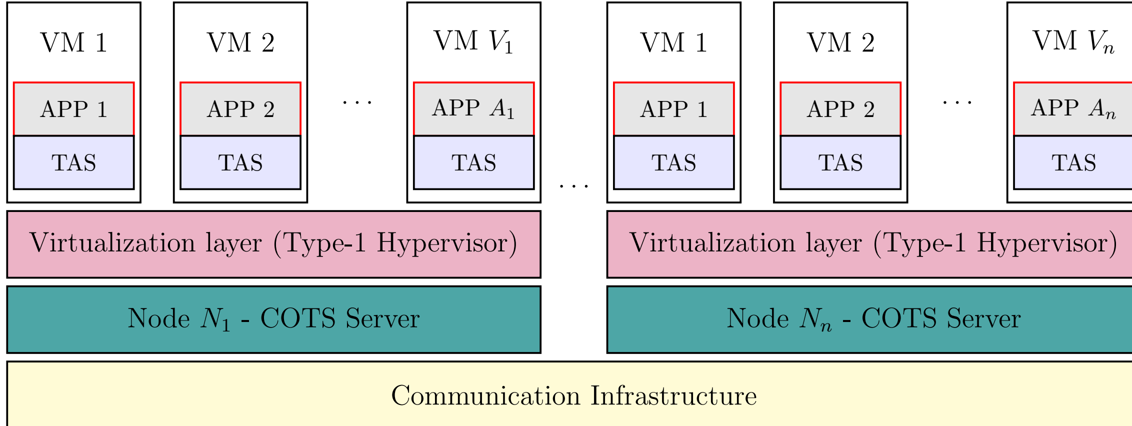


Figure VII.F3: *TAS Control Platform with Virtualization*

VII.1.3 Safety Requirements for TAS VMs

TAS control platform is already an industrially accepted solution and provides the redundancy, safety, and security measures required by the safety-critical railway applications. Therefore, our primary focus is to ensure that the hypervisor running a TAS Control Platform VM fulfills the requirements of the platform. Based on inputs from Thales Austria GmbH, we identified safety requirements for these VMs. They mainly concern encapsulation, predictability, safe connectivity, and real-time requirements.

- The VMs running safety-critical applications must be encapsulated with respect to safety. Errors in VMs should not trigger any failure in other VMs or compromise the node itself. We can achieve this by partitioning the different shared resources used by the VMs to ensure isolation.

- Multiple VMs running on the same hardware require predictable timing behavior (i.e., predictability despite multicore interference). They require some fixed minimum allocation of resources to the safety-critical VMs. For example, the virtualization layer must assign the VMs a minimum required time on the CPU and a fixed amount of cache and memory allocation, shared-bus/interconnect, memory bandwidth, and network bandwidth.

- Multiple safety-critical tasks within a VM require timely and correct execution. We require hierarchical scheduling to ensure predictability for these tasks. The first level of resource allocation allocates physical resources to each VM. The second level allocates the virtual resources assigned to a VM among the tasks running inside that VM.

- VMs must safely connect to off-site equipment (e.g., axle counters and signals). Another communication requirement is predictable timing and orderly message delivery between different VMs or among VMs and off-site equipment. VMs require a minimum guaranteed network bandwidth to achieve this.
- Predictable timing behavior for re-allocation of safety-critical applications because of occurrence of faults.
- The virtualization overhead should not hinder the real-time constraints of safety-critical applications.
- Monitoring is required to detect faults or non-conformant behavior of VMs.

The TAS Control Platform already contains several security hardening measures to fulfill these requirements. Thus, the introduced virtualization technology must not hamper any existing security requirements. Besides, the VMs running the TAS Control Platform require security encapsulation.

VII.2 RT-Cloud Component Selection – Hypervisor

Hypervisors have emerged as invaluable virtualization technologies for running VMs and driving innovation in a cloud environment. Since a hypervisor enables one host node to support multiple simultaneous VMs, hypervisors have become important cloud computing components. Hypervisors make VMs available to users across a virtual environment while still enabling the administrator to maintain control over cloud's resources, applications, and sensitive data. We are interested in hypervisors that are suitable for efficient execution on large clusters of COTS nodes. Cloud hypervisors must support a wide range of COTS server hardware, partitioning of various node resources, resource-efficient scheduling, dynamic reconfiguration (starting, stopping, and migrating VMs at run-time), and generic interfaces, such as libvirt API [217]. Ideally, they should also provide support for Intel Xeon-specific features to benefit from Intel Resource Director Technology (RDT) (e.g., memory bandwidth management, cache allocation, and monitoring technology) [102]. We considered four of the most widely used hypervisor technologies for cloud computing:

1. **Kernel Virtual Machine (KVM)** [110] is a virtualization module in the Linux kernel that allows the Linux kernel to function as a hypervisor. KVM requires a processor with hardware virtualization extensions (Section II.17.2), such as Intel VT [113]. KVM supports scheduling based on a combination of EDF and Constant Bandwidth Server (CBS). Efforts to further support real-time VMs under KVM are being made by use of PREEMT_RT real-time patch [193].
2. **Xen** [111] is an open-source (GPL) hypervisor. It is a type-1 hypervisor and supports guests with para-virtualization and full or hardware-assisted virtualization. Xen refers to guest VMs as domains. Every Xen installation contains a VM named Domain Zero (Dom0) that contains a privileged Linux-based guest Operating

System (OS) with drivers for the hardware and the tool stack to control the hypervisors and other VMs. The remaining VMs are called User Domains (DomUs). These VMs are not privileged and cannot control the hypervisor or other VMs.

RT-Xen [218] is an open-source platform that extends Xen to provide support for hierarchical scheduling with real-time scheduling algorithms such as the discarding periodic server (polling server) [219] and the (enhanced) sporadic server [220].

3. **VMware ESXi** [221] is a type-1 closed source hypervisor. It was leading the development of virtualization technologies on the x86 platform back in the early 2000s. However, there is no explicit real-time support.

4. **Microsoft Hyper-V** [222] is Microsoft's hardware virtualization product. It is a type-1 hypervisor with support for full virtualization. However, it is a closed source and not optimized for real-time performance.

VII.2.1 Qualitative Analysis

As a first step, we performed a qualitative analysis of the four hypervisor technologies. In Table VII.T1, we compare the basic features of interest among these hypervisors.

Xen and KVM are open-source hypervisors, while ESXi and Hyper-V are closed source. Open source hypervisors allow for a simplified business model and give high accessibility than closed source ones. As a result, they are more affordable and scalable. A business can increase the size of the cloud without fearing an increase in software budget. Moreover, vendor lock-in is a significant issue as providers of closed source hypervisors often require long-term contracts. For safety-critical domains like railways, it is paramount that the component integration follows strict safety requirement norms. Open source hypervisor can help make the component integration easier. Moreover, we can add new functionalities and resource management features as per the use case requirement without support from a hypervisor vendor. Therefore, we prefer an open-source hypervisor over the closed ones.

All four hypervisors support the x86 architecture, and thus, support servers based on 2nd Gen. Intel Xeon Scalable processors. All of them support full virtualization and hardware-assisted (full) virtualization (Section II.17.2), which is essential for using legacy railway applications based on the TAS control platform without modifying them. In general, hardware-assisted (full) virtualization has better performance than full virtualization. KVM and Xen additionally support para-virtualization (Section II.17.2). KVM supports para-virtualized I/O drivers for Linux in form of the VirtoIO drivers [115]. Xen supports para-virtualized drivers as well as para-virtualized OSs. Using para-virtualization support Xen and KVM can improve the performance of a guest OS as compared to full virtualization. However, using para-virtualized drivers with the railway use case requires adding these drivers to the TAS control platform.

A CPU model is a set of CPU features presented by the hypervisor to the guest. A host model presents the same hardware features of the host CPU to the guest; i.e., the virtual CPU model is the same as the physical CPU model. A predefined (or named)

Hypervisor properties	**KVM** [110]	**XEN** [111]	**VMware ESXi** [221]	**Hyper-V** [222]
Open Source	Yes	Yes	No	No
x86 hardware support	Yes	Yes	Yes	Yes
Para-Virtualization	I/O drivers (Virtio [115])	Yes	No	No
Full virtualization	Yes	Yes	Yes	Yes
Hardware assisted virtualization	Yes	Yes	Yes	Yes
CPU model	Host model, host pass-through, predefined		Host model	Host model, predefined
Cache model	Emulate, pass-through, disable			
Libvirt support	Yes, full support	Yes, Most functionalities	Limited	Limited

Table VII.T1: *Comparison of Virtualization Technologies for Cloud Computing*

CPU model presents a specific set of virtualized features to the guest that is modeled behind a particular CPU (not necessarily the host CPU). Predefined CPU models help shield the guest against various CPU hardware flaws. They also make live migration easier in a cloud with heterogeneous nodes. This is because predefined CPU model support makes it possible for nodes to emulate the same fixed CPU model irrespective of the host CPU heterogeneity. From the four hypervisors, KVM supports by far the most types of predefined CPU models. All hypervisors support three cache models – emulated, host pass- through, and disabled.

The railway use case requires the support of an open-source orchestration layer, i.e., a layer that provides a system designer the ability to allocate resources, monitor the cloud, and assist in recovery upon errors. Libvirt [217] is an open-source API and management tool for hypervisors that can be used as an orchestration layer in clouds. Libvirt has good support for KVM and Xen; unfortunately, it has limited support for the other two hypervisors.

In Table VII.T2, we compare the existing resource management features of the four hypervisors. Resource management features in hypervisors play a crucial role in ensuring that the system meets the requirements of safety-critical VMs.

KVM, Xen, and ESXi allow CPU hard pinning and setting CPU affinity for VMs. Hyper-V only allows setting CPU affinity. KVM and Xen allow allocating specific I/O threads. However, we could not find any information on the topic for ESXi and Hyper-V. All hypervisors allow allocation of global, per CPU Quota, memory space, and network bandwidth to the VMs. Moreover, they all provide an interface to access the core Performance Monitor Units (PMUs).

KVM and Xen provide support for some real-time scheduling policies, while ESXi and Hyper-V have no support for real-time scheduling. KVM supports round-robin,

First In First Out (FIFO), SCHED_DEADLINE (EDF + CBS). Further support for real-time exists in KVM in the form of PREEMPT_RT [193]. PREEMPT_RT enables full preemption of critical sections, interrupt handlers, and interrupt disabled code sequences. Xen supports scheduling policy based on real-time deferrable server [111]. RT-Xen [218] supports hierarchical scheduling with deferrable server, periodic server, discarding periodic server (polling server) [219], and the (enhanced) sporadic server [220].

All four hypervisors support Non-Uniform Memory Access (NUMA) node tuning. Often cloud nodes contain multiple processors and memory modules. The relative location of a memory module to a processor determines the memory access latency of a VM running on that processor. A VM has better performance if the data accessed by it is present in the memory module closet (local memory) to the processor. NUMA node tuning helps to migrate data on demand to memory that is local to the processors accessing that data.

All four hypervisors support some form of power-saving functionalities. Unchecked power consumption leads to a large amount of excess heat production. As a result, cooling costs also increase. Moreover, heating can lead to a reduction in the life span of the hardware. Although servers have unlimited access to electricity, power saving is required to keep the electric costs down. Hence, all hypervisors must have support for power-saving functionalities.

As discussed in Chapter IV, 2nd Gen. Intel Xeon processors have special hardware features to support monitoring and allocation of Last Level Cache (LLC) space and memory bandwidth. To get the most advantage of nodes consisting of these processors, hypervisor support for their features is essential. In newer versions of Xen and KVM it is possible to access these features. However, ESXi and Hyper-V do not provide any support. Moreover, since they are closed source, it is cumbersome for us to integrate these hardware features with the hypervisors.

Based on the qualitative analysis done so far, we concluded that KVM and Xen are better suited than ESXi or Hyer-V. ESXi and Hyer-V either miss or have limited support for essential features such as predefined CPU models, Libvirt, and real-time scheduling options. Moreover, they do not allow us to benefit from the Intel Xeon specific features. Finally, since they are closed source, we cannot add missing or new functionalities or resource management features as per requirement. Therefore, we narrowed down the selection of hypervisor technology between KVM and Xen. In terms of requirements, KVM and Xen equally suit our use case as per the qualitative analysis.

VII.3 Quantitative analysis

To further narrow down our choice, we performed a quantitative analysis of Xen and KVM by running benchmarks to compare their performance. Most previous works use benchmarks to compare the average performance of Xen and KVM for non-safety critical systems (e.g., [223] and [224]). The works that consider real-time systems and worst-case comparisons only present the Cyclictest benchmark [225] for a single core on non-server-grade hardware platforms (e.g., [226]). We used the benchmarks to evaluate

Hypervisor Resource Management Features	**KVM** [110]	**XEN** [111]	**VMware ESXi** [221]	**Hyper-V** [222]
CPU core allocation to a VM	Yes	Yes	Yes	Affinity, but no hard pinning
I/O Thread allocation per VM	Yes	Yes	No	No
Scheduler options	Complete Fair Scheduler (CFS), FIFO, and Round-Robbin [183], Real-time scheduler with PREEMPT-RT patch [193]	Credit, Credit2, Real-time Deferrable Server, and ARINC653 [111], Global/Partitioned EDF and RM, polling and sporadic server with RT-Xen [218]	Schedulers options only differ concerning security but not timing	Classic, Core and Root scheduler
Run-time re-configuration	VMs can be added/removed at runtime			
CPU core quota	Yes	Yes	Yes	Yes
Global CPU quota	Yes	Yes	Yes	Yes
Memory space isolation	Yes	Yes	Yes	Yes
Performance monitoring	Yes	Yes	Yes	Yes
NUMA node tuning	Yes	Yes	Yes	Yes
Network bandwidth allocation	Yes	Yes	Yes	Yes
Live Migration	Yes	Yes	Yes	Yes
Power management	Yes	Yes	Yes	Yes
Cache tune and monitoring	Yes, on Xeon	Yes, on Xeon	No	No
Memory bandwidth allocation	Yes, on Xeon	Yes, on Xeon	No	No

Table VII.T2: *Comparison of Existing Resource Management features in Virtualization Technologies*

the worst-case performance on COTS server-grade hardware with the multicore processor. Moreover, we ran benchmarks to observe the effect of multicore contention on the VMs.

Our experimental setup consists of a COTS Dell R640 server based on a single Intel 2nd Gen. Xeon 5218 processor (Section II.4) running at 2.30GHz. The processor has 16 dual-threaded CPU cores and 96GB memory (12 DDR4 Dynamic Random-Access Memory (DRAM) modules × 8GB). We disabled hardware multi-threading (logical processors) in the BIOS as it leads to an increase in shared resource contention. Thus, we only have the 16 physical CPU cores available for use (instead of 32 logical cores). Additionally, we disabled C-states and other hardware power-saving features in the BIOS to ensure that frequency scaling does not affect the system behavior.

We used KVM and Xen with the following specifications:

1. KVM with Ubuntu server and Linux Kernel version 5.4.87-rt48 SMP PREEMPT_RT as Host OS

2. Xen (version 4.11.4, Real-Time Deferrable Server - RTDS) with Ubuntu server 20.04.1 and Linux Kernel version 5.4.87 SMP Preempt as Dom0. We do not use PREEMPT_RT version in Dom0 as a previous work has already shown that PREEMPT_RT does not work well with Xen and incurs exceptionally high latency [226].

Both, KVM and Xen, used hardware assisted virtualization (Intel VTx [113]). In the case of KVM, we isolated cores 2 to 15 from SMP balancing and scheduler algorithms of the host OS (by using the isolcpus kernel parameter). The only way to move a guest VM onto or off an isolated CPU is by assigning CPU affinity. It ensures that KVM does not place unwanted VMs or host OS threads on these CPU cores. Similarly, in the case of Xen, we restricted Dom0 to physical CPU cores 0-1 and left CPU cores 2 to 15 for executing guest VMs.

We considered three scenarios:

1. *Scenario 0 – No virtualization + isolation:* In this scenario, an Ubuntu server 20.04.1 (Linux kernel 5.4.87-rt48 SMP Preemt_RT) runs directly on the hardware platform and employs no virtualization. We used this scenario to compare as a baseline and observe the overheads introduced by Xen and KVM in the other two scenarios. We executed the benchmarks alone (one at a time) on top of Linux and pinned them to CPU 2. We did not use any other interfering benchmarks.

2. *Scenario 1 – Virtualization + isolation:* One Guest VM with 1 virtual CPU (vCPU) pinned to physical CPU core 2. We allocated 4GB RAM to the VM. We ran the benchmarks alone (one at a time) in this VM for Xen and KVM. We did not use any other interfering benchmarks.

3. *Scenario 2 – Virtualization + interference:* 14 Guest VMs, each with 1 virtual CPU and 4GB RAM. We pinned the vCPU on a physical CPU core between core ID 2 to 15 (only one vCPU ran on a physical core). We did not place any restrictions on the amount of other shared resources (e.g., memory bandwidth and network bandwidth) used by the VMs. We ran the benchmarks (one at a time) in each VM

simultaneously for Xen and KVM. We aimed to observe the performance of the hypervisors under resource contention.

Table VII.T3 and VII.T4 summarizes the three scenarios.

Scenario	**Host**	**Number of VM(s)**	**Description**
0. No virtualization + isolation	Linux (isolcpus=2-15)	none	Benchmarks pinned to core 2
1. Virtualization + isolation	KVM (isolcpus=2-15) or Xen (Dom0=0-1)	1	VM executing benchmarks pinned to core 2
2. Virtualization + interference	KVM (isolcpus=2-15) or Xen (Dom0=0-1)	14	VMs executing benchmarks pinned to core 2-15 each (one VM per core)

Table VII.T3: *KVM vs. Xen - Summary of the Three Scenarios*

Scenario	Core ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	VM	No Virtualization															
	Benchmark			✓													
1	VM			✓													
	Benchmark			✓													
2	VMs			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Benchmark			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table VII.T4: *KVM vs. Xen - Three Scenarios*

Due to confidentiality issues, it was not possible to directly use TAS Control Platform as Guest OS to present the results of this evaluation. Instead, we ran the Ubuntu server (Linux Kernel version 5.4.87-rt48 SMP Preempt_RT) in all guest VMs. We chose Ubuntu Linux with PREEMPT_RT as the guest OS because it is similar to a TAS Control Platform and has out-of-the-box support for all the required benchmarks. In contrast to the TAS Control Platform, it is not optimized towards extreme low latency, and therefore, absolute numbers of the benchmarks are not representative of what we can achieve when using the TAS Control Platform. However, the motivation for the benchmarks is to compare KVM against Xen. The relative performance of both hypervisors remains independent of whether we use the TAS Control Platform or Ubuntu.

We used four different benchmarks to compare the hypervisors with respect to CPU, memory, and network performance. The railway applications of the use case do not rely

on the filesystem during their operational phase. Instead, they have all the necessary data pre-fetched to the memory. Thus, we did not use a benchmark to compare the performance of reading or writing from the filesystem. The following subsections explain the benchmarks and present the results of the benchmarks from the three scenarios.

Cyclictest Benchmark

Cyclictest [225] measures the latency of a (Linux-based) OS to a stimulus. Latency refers to the delay between the occurrence of an event to the time when the kernel handles the event. For example, the delay between the timer expiration and the kernel handling the task (thread) waiting for that timer. Such latency exists in all OSs. The causes of latency include scheduler overheads, delay due to high priority tasks, delay preempting the currently running task, and interrupt overhead. Cyclictest measures latency by calculating the time between when a timer expires and when the thread that sets the timer runs.

It is problematic to ensure that a safety-critical railway task meets its deadline if the guest OS experiences unpredictable latency due to the virtualization platform. Therefore, the latency must not exceed an arbitrary threshold in the worst case. System designers can determine this threshold based on the real-time parameters of the safety-critical railway tasks in the VM. Unfortunately, the exact value for this threshold is not provided due to confidentiality issues. However, we need the latency to be in a low microsecond value.

We ran the cyclictest benchmark to determine the worst-case observed latency in the VMs of our scenarios. Since executing an event from an idle state is usually spontaneous, the results of cyclictest would not be meaningful if we ran the benchmark only for a short period and without appropriate stress loads. We were interested in detecting the reaction to an asynchronous event independent of the time and code-path when the event occurs. Therefore, we ran additional CPU stressing benchmarks (Sysbench CPU) together with cyclictest to determine the worst-case latency reliably. We ran cyclictest for continuous 1×10^8 test cycles in each case. The base interval of the thread for cyclictest was $1000\mu s$.

As observed in the graph in Figure VII.F4, scenario 0 (no virtualization) had the lowest observed worst-case latency ($9\mu s$). The VM in Scenario 1 with KVM had the closest worst-case latency ($32\mu s$) to scenario 0. However, Scenario 1 with Xen had a much higher maximum observed latency ($377\mu s$) as compared to both scenario 0 and scenario 1 with KVM.

As expected, we did not observe considerable differences between the results of scenarios 1 and 2 for the same hypervisor as CPU cores are not shared by the VMs. The maximum observed values for scenario 2 with KVM and Xen were $45\mu s$ and $455\mu s$. Limited usage of shared resources, such as LLC and memory bandwidth, lead to the minor difference in the observed values between scenario 1 and 2.

We conclude that KVM introduced lower virtualization overhead as compared to Xen. Thus, if we use KVM to run the railway VMs, the safety-critical railway tasks of each

[1] Lower latency values are better

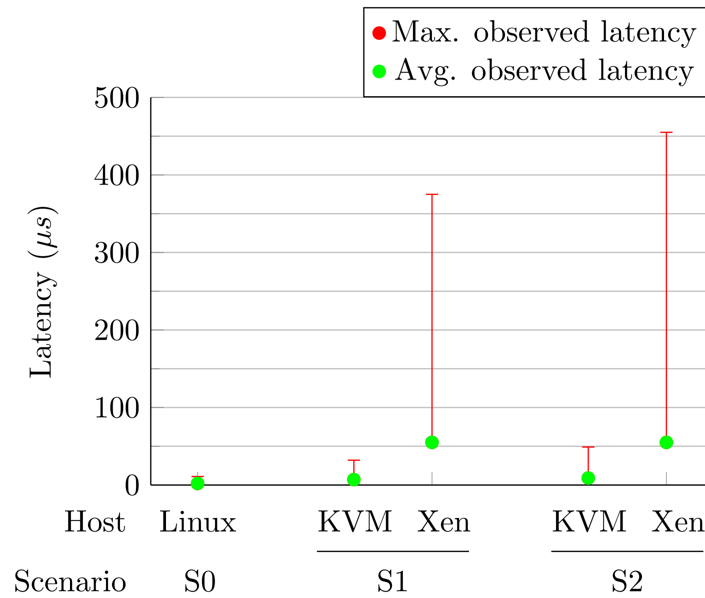


Figure VII.F4: *Cyclictest Bechmark*[1]

VM can meet their deadline if the earlier mention threshold is $> 45\mu s$ for this hardware architecture. In Xen, this threshold must be $> 455\mu s$. Hence, we prefer KVM over Xen for this scenario.

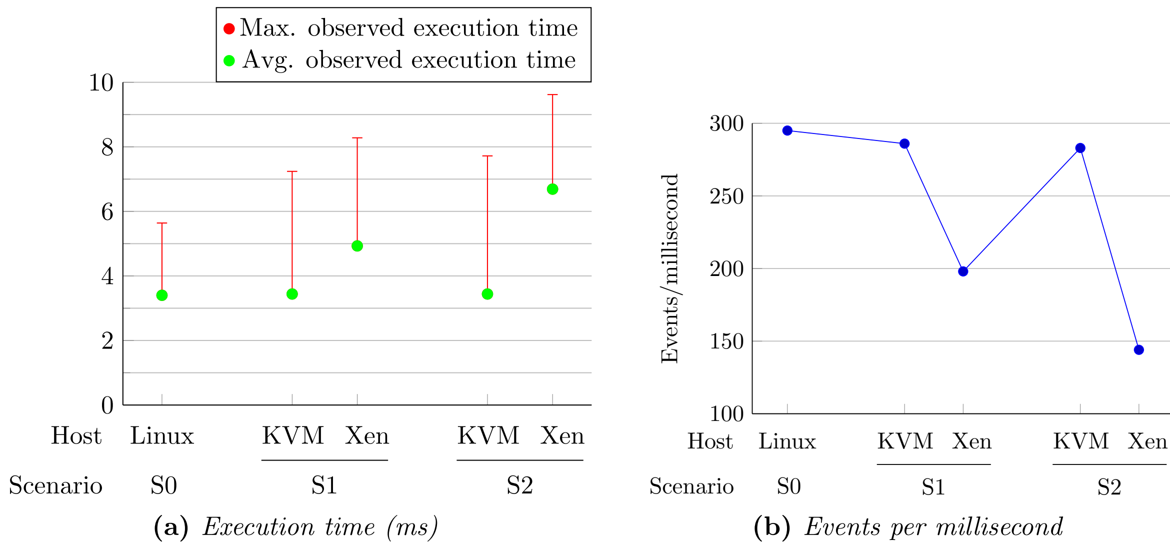
As a reminder, the experiments were performed with VMs running Ubuntu server (Linux Kernel v5.4.87-rt48 SMP Preempt_RT) and not TAS platform due to confidentiality issue. Therefore, the actual value for the TAS platform will be lower as it is geared towards low latency performance. However, the relative latency of both hypervisors remains similar.

Sysbench CPU Benchmark

We used Sysbench CPU benchmark [227] for benchmarking CPU capabilities in the three scenarios. It measures CPU performance by counting how many operations are performed within a given time or how long a benchmark takes to complete. It stresses the CPU by performing prime number verification. Based on the reduction of performance observed running this benchmark between Scenario 0 and the other two scenarios, we can estimate the impact on the worst case execution time of a safety-critical railway application (running in a VM) due to virtualization.

We ran this benchmark 1000 times for verifying prime numbers up to 20000. As observed in the graph in Figure VII.F5a, Scenario 0 (no virtualization) had the lowest observed worst-case execution time ($5.64ms$). The benchmark executing in the VM of Scenario 1 with KVM had worst-case execution time ($7.24ms$), which is the closest to scenario 0. In the case of Xen, the maximum observed execution time in scenario 1 ($8.28ms$) is higher than the former two cases.

As observed in the graph in Figure VII.F5, Scenario 0 (no virtualization) and scenario



(a) *Execution time (ms)* **(b)** *Events per millisecond*

Figure VII.F5: *Sysbench CPU Benchmark*[2]

1 with KVM performed almost the same number of events per milliseconds (295 and 286 events/ms, respectively). However, scenario 1 with Xen performed lower events per milliseconds (198 events/ms).

As expected, we did not observe considerable differences between the results of scenarios 1 and 2 for the same hypervisor as CPU cores are not shared by the VMs. The maximum observed execution times for scenario 2 with KVM and Xen were 7.72ms and 9.08ms (Figure VII.F5a), while the minimum observed events per milliseconds were 283 and 144 events/ms respectively (Figure VII.F5b). Limited usage of shared resources, such as LLC and memory, leading to the minor difference in the observed values between scenarios 1 and 2. Overall, scenario 2 with Xen had the worst performance impact.

Again, we conclude that KVM introduced lower virtualization overhead as compared to Xen. If we use KVM to run the railway VMs, the performance impact is much lower than compared to Xen. Hence, we prefer KVM over Xen for this scenario.

Sysbench Memory Benchmark

We used Sysbench Memory benchmark [227] for determining the memory read and write bandwidths in the three scenarios. This benchmark allocates a memory buffer and then reads or writes from it until the entire buffer has been read or written. The size of each read or write is 64 bits. Based on the reduction of performance observed running this benchmark between Scenario 0 and the other two scenarios, we can estimate the impact on memory read or write bandwidth for the railway VMs due to virtualization.

We ran the benchmark 1000 times to read or write 100GB from/to the RAM by a single thread. As observed in the graph in Figure VII.F6, Scenario 0 (no virtualization) and scenario 1 with KVM achieved similar read bandwidth (1.39$Gbps$ and 1.31$Gpbs$) and write bandwidth (1.30$Gbps$ and 1.22$Gpbs$). However, scenario 1 with Xen performed

[2]Lower execution time values and higher events per millisecond values are better.

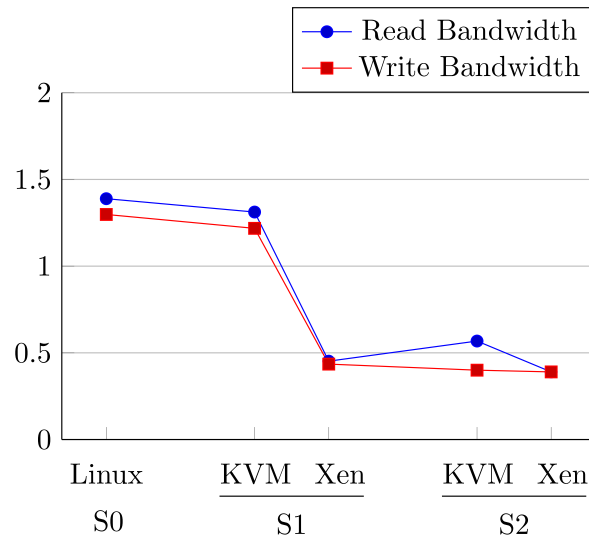


Figure VII.F6: *Sysbench Memory Read and Write Benchmark*[3]

poorly (read – 0.452*Gbps* and write – 0.44*Gpbs*). We experimented with different settings for Xen. However, none of the settings provided any change in the available bandwidth for the VMs.

Since the memory bandwidth is shared by the VMs in scenario 2, we observed a considerable reduction in this scenario with KVM (read – 0.568*Gbps* and write – 0.40*Gpbs*). On the other hand, Scenario 2 with Xen (read – 0.39*Gbps* and write – 0.39*Gpbs*) performed similarly to Scenario 1 and incurred only a slight reduction in the bandwidths. However, the performance of scenario 1 with Xen was already low. Overall, scenario 2 with Xen had the worst performance impact.

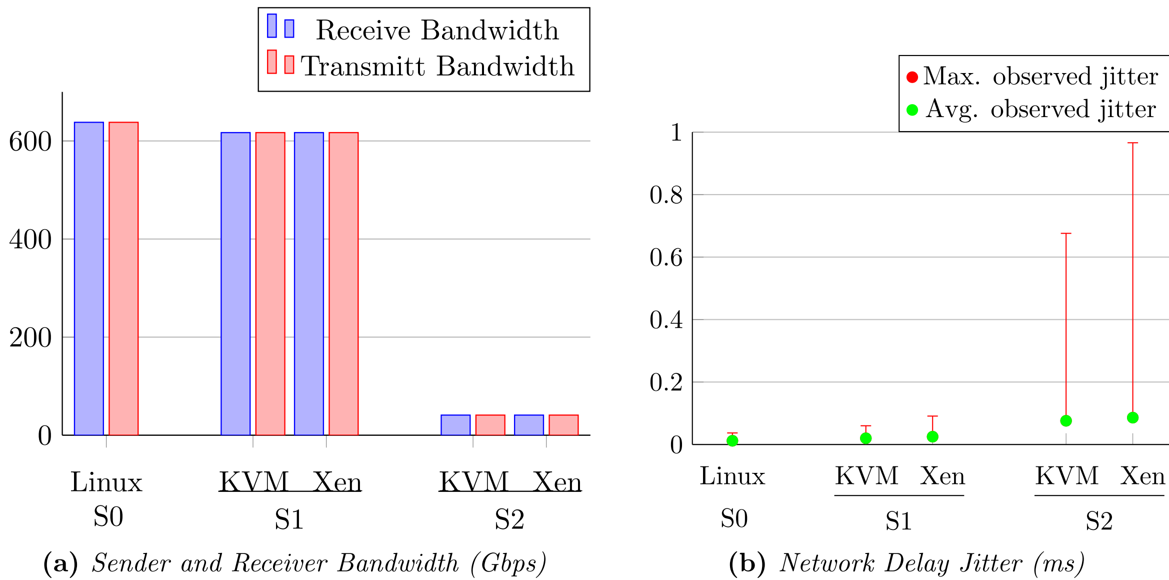
Again, we conclude that KVM introduced lower virtualization overhead as compared to Xen. Thus, if we use KVM to run the railway VMs, the VMs can potentially obtain a higher memory bandwidth. Hence, we prefer KVM over Xen for this scenario.

iPerf3 Benchmark

We used iPerf3 [228] to benchmark the achievable bandwidth and delay jitter on IP networks. The benchmark supports various parameters related to timing, buffers, and protocols (such as TCP and UDP).

We hosted the iperf3 server on a separate computer connected via a gigabit router. We ran this benchmark in the three scenarios. Firstly, we used TCP protocol with a window size of 128KB for 1000 seconds (total transmitted data: 71.8GB) to measure the sender/receiver network bandwidth. Secondly, we use the UDP protocol with a buffer size of 1460B for 1000 seconds (total transmitted data: 125MB) to measure the delay jitter.

[3] Higher bandwidth values are better.



(a) *Sender and Receiver Bandwidth (Gbps)* **(b)** *Network Delay Jitter (ms)*

Figure VII.F7: *iPerf3 benchmark*[4]

Figure VII.F7 shows the results. We observed only a small reduction in sender and receiver bandwidth in Scenario 1 (KVM – 617$Gbps$ and Xen – 617$Gbps$ for both, sender and receiver) as compared to scenario 0 (638$Gbps$). Scenario 1 (KVM – 0.060ms and Xen – 0.091ms) has higher worst-case delay jitter as compared to scenario 0 (0.037ms).

Since the network bandwidth is shared by the VMs in scenario 2, we observed a drastic reduction in this scenario (KVM – 41$Gbps$ and Xen – 41$Gbps$). However, Xen (0.97ms) suffered from a higher worst-case jitter as compared to KVM (0.68ms). Thus, if we use KVM to run the railway VMs, the VMs will suffer from a lower delay jitter.

VII.4 Demonstration

Taking into account the qualitative and the quantitative analysis, Thales Austria GmbH selected KVM as the hypervisor of choice to host the safety-critical railway VMs.

For demonstrating the feasibility of using KVM, a test setup with a virtualized RBC application (a Triple Modular Redundancy (TMR) application with TAS Control Platform) was deployed by Thales in coordination with the Austrian federal railways (ÖBB) [152]. In an actual deployment scenario, an RBC retrieves track-side information from an interlocking. It communicates via a radio link to the On-Board Unit (OBU) of a train and provides the OBU with movement authorities, such as how fast the train can go and when the train needs to brake. Both the OBU and the interlocking are external components connected via a secure tunnel to the cloud.

The demo setup emulated the OBU, interlocking, and other required components by software simulations. The simulated components communicated with the RBC similar to the real components. They used railway-specific protocols with additional

[4] Higher bandwidth values and lower delay jitter values are better.

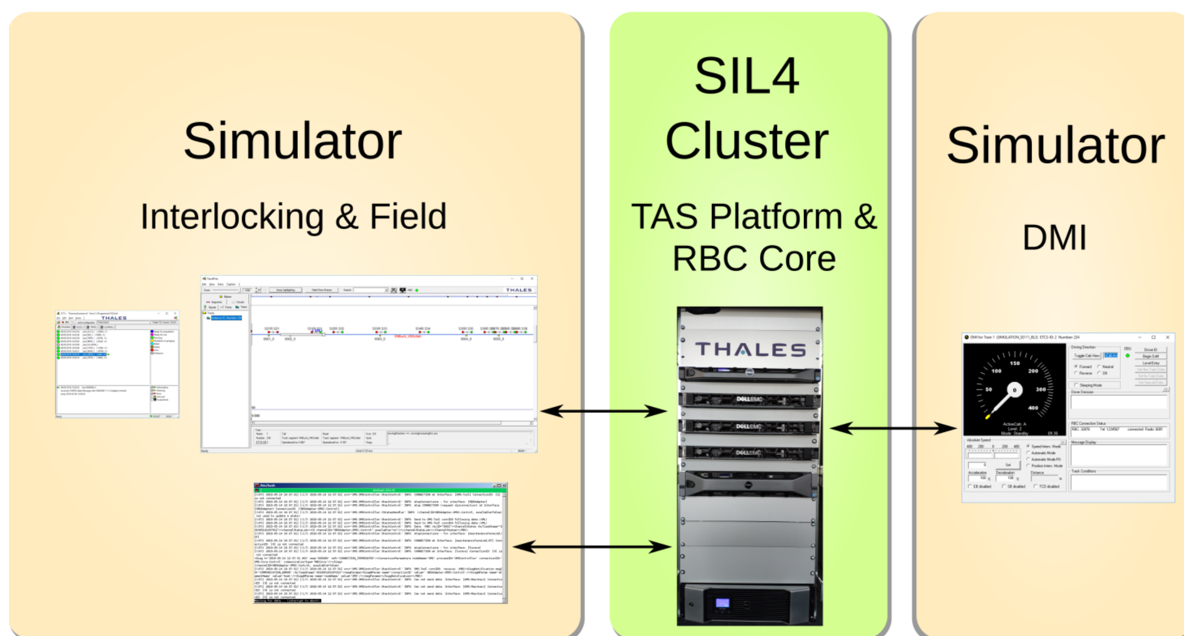


Figure VII.F8: *Test Setup for the RBC Application*

security protection and network isolation. Figure VII.F8 illustrates the test setup. The demonstration showed that a KVM-based private cloud can safely run a safety-critical RBC application.

This demonstration proved that our selected virtualization technology, KVM, can fulfill the operation of safety-critical TMR railway applications based on the TAS Control Platform in a private cloud.

VII.5 RT-Cloud Component Selection – Resource Management Layer

We propose a KVM-based private cloud containing three *zones*. A zone is a deployment area that is considered a single failure domain. We must deploy replica VMs of TMR applications, such as the TAS control platform considered in the railway use case, in different zones to increase availability, improve fault-tolerant VMs and help protect against unexpected failures.

As observed in scenario 2 in the quantitative analysis, when we executed multiple VMs in parallel, there was contention in the shared resources. Thus, KVM alone on a node is not adequate to meet all the requirements for a safety-critical use case such as the railway one described in this chapter. KVM neither supports partitioning and allocation of all the node resources among the VMs, nor does it have a view of cloud-wide resources. As explained in Section II.25, it is challenging to meet end-to-end constraints by considering each node, resource, or application individually. On the contrary, we require monitoring, control, and coordination of cloud nodes to ensure that the safety-critical VMs can meet end-to-end deadlines (even in the presence of faults), while non-critical VMs can achieve

the desired Quality of Service (QoS). Hence, we propose to use our resource management framework explained in the previous chapters with a private cloud.

For resource management purposes, each node N_λ contains a virtualization domain Local Resource Manager LRM_λ (Section IV.8) to manage the resources in coordination with KVM. Each Local Resource Manager (LRM) executes on top of KVM directly and has multiple LRSs and Local Resource Monitor (MONs). For ensuring adequate allocation of shared resources, we can use the LRSs proposed in Section IV.7 or use existing schedulers as LRSs. For example:

- A table-driven scheduler for cloud hypervisors, such as Tableau [229] or ARINC 653 scheduler [111]. Each safety-critical railway VM has quite a few services and subsequently various internal timeouts running. Suppose the hypervisor schedules the VMs at any time. In that case, jobs running in the VMs might occur at any arbitrary time without a fixed period in-between, albeit being overall restricted with their CPU usage. To exploit existing KVM schedulers for the TAS Control Platform would require reducing the internal functionality and aligning the "time slices" with the host, which breaks the boundary of the VMs for synchronization mechanism. A table-driven scheduler will ensure that the safety-critical VMs are provided CPU execution time as per their Worst-case Execution Time (WCET) precisely when required. Unfortunately, neither Tableau nor ARINC 653 scheduler have an existing implementation for KVM, our hypervisor of choice. Both of them have been implemented only for the Xen hypervisor. Due to the difference in the internal functioning of Xen and KVM, we could not use these existing solutions with KVM. Thus, we use our TT-LRS design (Section IV.7.4). Section VII.6.1 provides an evaluation and overheads for our TT-LRS.

- Intel Cache Allocation Technology (CAT) [102] and KVM memory allocation [110] to allocate dedicated LLC line(s) and memory space(s) respectively to safety-critical VM. The LLC line(s) and memory space(s) assigned to a VM are not accessible by unauthorized VM, thus ensuring isolation among VMs. A minimum allocation of LLC space helps reduce the pessimism in WCET estimation and contributes towards ensuring that the safety-critical VMs meet their deadlines. For interfacing with CAT, we use the LRS design presented in Section IV.7.3. We allocate memory regions to VMs by using the KVM API [230].

- Intel MBA [102] or MemGuard [88] to allocate memory bandwidth to VMs. As observed while running the memory benchmarks, there is a drastic reduction in the maximum available memory bandwidth when concurrently executing VMs access the memory. A minimum allocation of memory bandwidth helps reduce the pessimism in WCET estimation and contributes towards ensuring that the safety-critical VMs meet their deadlines. For interfacing with MBA, we use the LRS design presented in Section IV.7.3. In Section VII.7, we present some initial results from our evaluation of MBA to regulate the memory bandwidth.

- KVM network bandwidth allocation [110] to allocate average and peak inbound and outbound network bandwidth to safety-critical VM. We can allocate network

bandwidth to VMs by using the KVM API [230]. In the context of this dissertation, we do not use this feature and leave it for future work.

For monitoring resource and applications, we can use the MONs proposed in Section IV.4 or use existing solutions as MONs. For example:

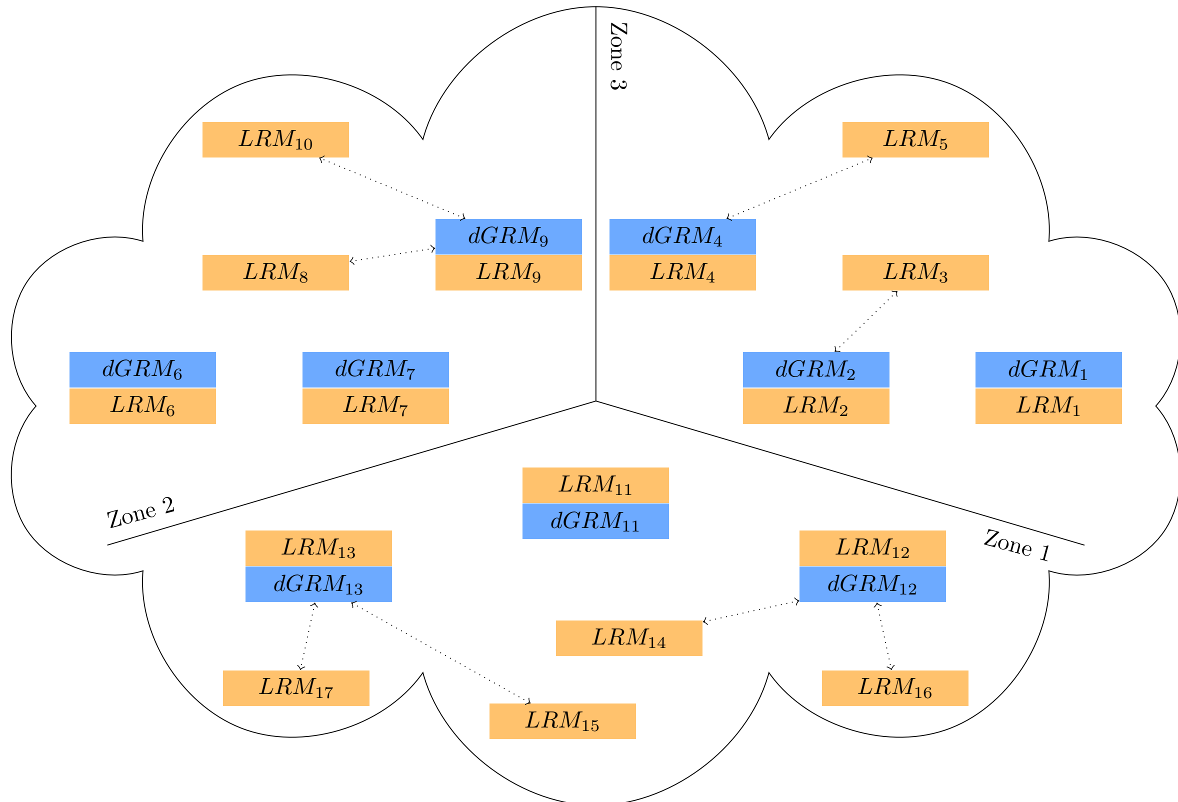
- The CPU usage monitor in KVM [110] to monitor the CPU availability. Based on the CPU availability, the LRM can decide on the scheduling of new VM on the node. The LRM can take action if a VM misbehaves and exceeds its assigned CPU budget.
- Intel Memory Bandwidth Monitoring (MBM) and Cache Monitoring Technology (CMT)[102] to check the memory-bandwidth and LLC usage by each VM. The LRM can take action if a VM exceeds its assigned memory bandwidth quota. For interfacing with MBM and CMT, we use the MON design presented in Section IV.7.3. A Network-bandwidth monitor, a KVM feature, has similar functionality as MBM for the network.
- Potential deadline overrun MON (Section IV.4.4) and core failure MON (Section IV.4.3).

The LRM of a node manages all the node's resources based on the monitoring information provided by the MONs. The LRM assigns the required resource to the VMs (via LRSs) to ensure that the safety-critical VMs meet their safety assurance levels and the non-critical VM can get the best possible QoS.

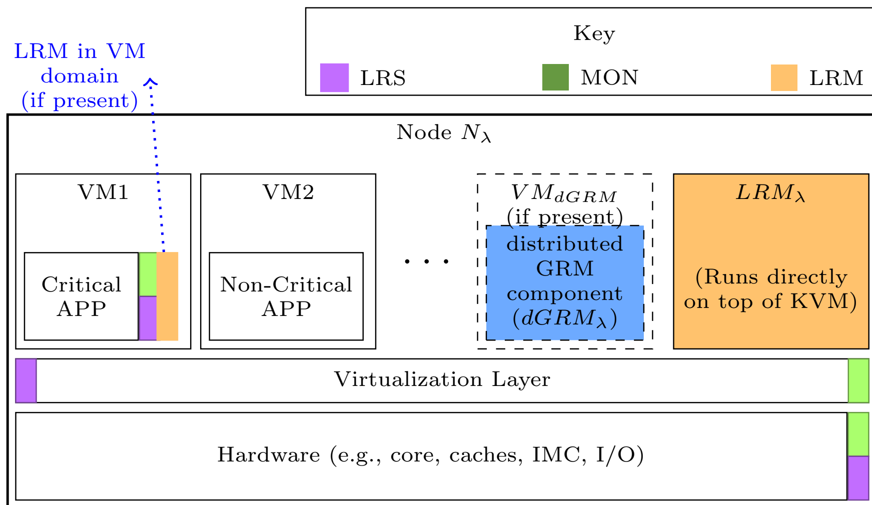
Similar to local and global resource management explained in Chapters IV and V, the resource management reconfigures the cloud upon resource failures or change in conditions at run-time. If node-level scheduling of a VM is not possible, the LRM coordinates with the global resource management to find nodes that can accommodate the VM. In addition, the global resource management ensures that no replica VMs of TMR safety-critical applications are placed in the same Zone.

In the avionics use case (Chapter VI), we used a single central Global Resource Manager (GRM) (IV.9.1) to provide reconfiguration for core failure management. As illustrated in Figure VII.F9, we use the distributed global resource management here for cloud-wide reconfiguration as it provides additional safety and security as explained in Section V.6. A distributed Global Resource Manager $DGRM_\lambda$ can be present on a node N_λ. However, all nodes do not need a Distributed Global Resource Manager (DGRM) present on them. As a reminder, we only need $3f + 1$ DGRMs to ensure Byzantine fault tolerance. The LRMs of nodes without DGRMs participate in global resource management via other nodes as explain in Section V.7.1.

In Section VII.8, we present the evaluation for reconfiguration in the cloud for core failure management using the DGRM (Section V.6).



(a) *DGRMs and LRMs in Cloud*



(b) *DGRM and LRM on a Cloud Node*

Figure VII.F9: *Resource Management in Cloud*

VII.6 TT Scheduling on Cloud Nodes

This section describes our CPU-and memory bandwidth-aware offline scheduler that generates a TT scheduling table for use with an LRM and a TT-LRS (Section IV.7.4). For simplicity of explanation, we assume each VM requires only one virtual CPU. We consider safety-critical VMs with a constant amount of memory accesses. Each VM is granted at most one memory access at a time, resulting in waiting times for accesses from other VMs. We assume that an ongoing memory access is not preemptible.

We consider global time whose progress is triggered by equidistant events, $\epsilon_0, \epsilon_1, \ldots, \epsilon_\infty$ [62]. We consider a sparse time base where the time duration separating two of these events, ϵ_s and ϵ_{s+1} respectively, is called a *Slot*, S_s [51]. Δt_{Slot} is the duration of each slot. Δt_{Slot} is constant and same for all slots. Slot $start(S_0) = \epsilon_0$, $end(S_0) = start(S_1) = \epsilon_1$ and so on. A multicore slot on a cloud node can run P VMs at a time, where P is the total number of CPU cores on the node. The sum of the reserved bandwidth for each of the VMs running in a single slot must not exceed the maximum allowed bandwidth of the node (BW_{max}). Each slot is represented by $S_s(\Delta t_{Slot}$, BW_S^p, M_s, $SBW_s)$ as shown in Table VII.T5.

A slot S_s is described by:	Units
CPU Allocation Δt_{Slot}	in time units, e.g., milliseconds
Memory Bandwidth allocation BW_s^p for each core $p \in \{0, 1, \ldots P-1\}$	in memory accesses per regulation interval of a slot
Mapping M_s of VMs (τ) allocated to S_s to CPU cores	M:$\tau \rightarrow p$
Spare (unused) Memory Bandwidth SBW_s	in memory accesses per regulation interval of a slot

Table VII.T5: *Summary of Slot*

We define a *regulation interval* as a slot-wide parameter that is small to enforce memory bandwidth BW^p effectively. In the most straightforward implementation, the regulation interval can be equal to the slot length. However, this would lead to very coarse memory bandwidth regulation. Although a small regulation interval is better for predictability, there is a practical limit on reducing the period due to interrupt and scheduling overhead; based on MemGuard [88], we set the regulation interval to $1ms$.

We define a VM τ_t with a tuple $\langle r_t, C_t^s, d, \phi_t, C_t^m \rangle$, where r_t is the start slot of τ_t, C_t^s is worst-case execution time of τ_t (in slots) when running in isolation and with no memory bandwidth restrictions, d_t is the absolute deadline of the τ_t (in slots), ϕ_t is the maximum number of memory accesses τ_t is allowed to issue per regulation interval, and C_t^m is the worst-case execution time of τ_t (in slots) when restricting the memory bandwidth of the τ_t to ϕ_t accesses per regulation interval. Note that C_t^m is dependent on ϕ_t; as the memory bandwidth of the task ϕ_t is increased, C_t^m will decrease. In other words, each value for ϕ_t will produce a different C_t^m for each τ_t. Moreover, [65] demonstrated that the latency for a single memory access operation increase significantly when the number

of active cores increases.

We created a heuristic algorithm to assign slots to τ_t before runtime, considering the points mentioned earlier. ϕ_t and its corresponding C_t^m are determined in such a way that the τ_t is schedulable while using the minimum possible bandwidth from BW_{max} on the node. Appendix E presents this heuristic. The result of the heuristic is a scheduling table with slots consisting of BW_s^p values, M_s, and SBW_s per slot S_s.

This scheduling table is stored by the LRM of a cloud node. The LRM only runs on a node's non-TT (housekeeping) core and executes before the start of each slot (Figure VII.F10). The LRM controls a TT-LRS (Section IV.7.4) on each core of the node and executes VMs in a slot S_s as per the mapping M_s in the scheduling table. At the start of a slot, the TT-LRS stops VMs currently executing and dispatches the new VMs as ordered by the LRM. The LRM also controls a LRS based on Memguard [88] which reserves memory bandwidth BW_s^p for core p during slot S_s. If a VM finishes execution early, the M_s is updated accordingly in future slots, and the BW_s^p value stored for that VM in future slots is freed (as a result, SBW increases for those slots).

If no TT VM is assigned to a core β in a slot S_ρ, the LRM lets the Linux scheduler execute any available non-TT VMs on core β only for duration of S_ρ. However, the LRM still restricts the memory bandwidth of these core β to ensure that it does not exceed the spare memory bandwidth SBW_ρ of S_ρ. If multiple such cores exist in a slot S_ρ, then SBW_ρ is divided among these cores proportional to the demand of the non-TT VMs executing on the respective core. This ensures efficient use of the node resources without impacting the TT VMs. Moreover, at end of a slot, the LRM of a node updates any change in SBW and M of future slots, and status of applications to the GRM for global resource management purposes.

An added advantage of this approach is that if an LRM has access to multiple scheduling tables for its node, it can easily switch between them upon a change in operational modes or a local/global reconfiguration without incurring any overhead. The LRM can change the scheduling table at the end of a slot if an immediate mode change is required or at the end of a hyperperiod if a deferred mode change is needed. The TT LRS remains unaware of the mode change as it only dispatches VMs every slot as per the directive of the LRM.

VII.6.1 TT-LRS Evaluation

As explained in Section IV.7.4, we designed a LRS to support scheduling of TT VMs in KVM (with PREEMPT_RT patch). KVM schedules VMs based on the Linux scheduler. The TT-LRS introduced a new scheduling policy in Linux ($SCHED_TT$). The new scheduling policy has the highest priority and sits on top of the hierarchy of (native) Linux scheduling classes: $TT > RT$ (Real-Time) $> CFS$ (Completely Fair Scheduling) $> IDLE$. If there are no runnable TT class VMs (or Linux processes), the Linux scheduler looks for runnable VM (or Linux processes) in each class in decreasing priority order.

Unlike the existing scheduling modules in Linux, the LRM (running on the housekeeping core) must explicitly enable the TT-LRS on a set of CPU cores at runtime (Figure

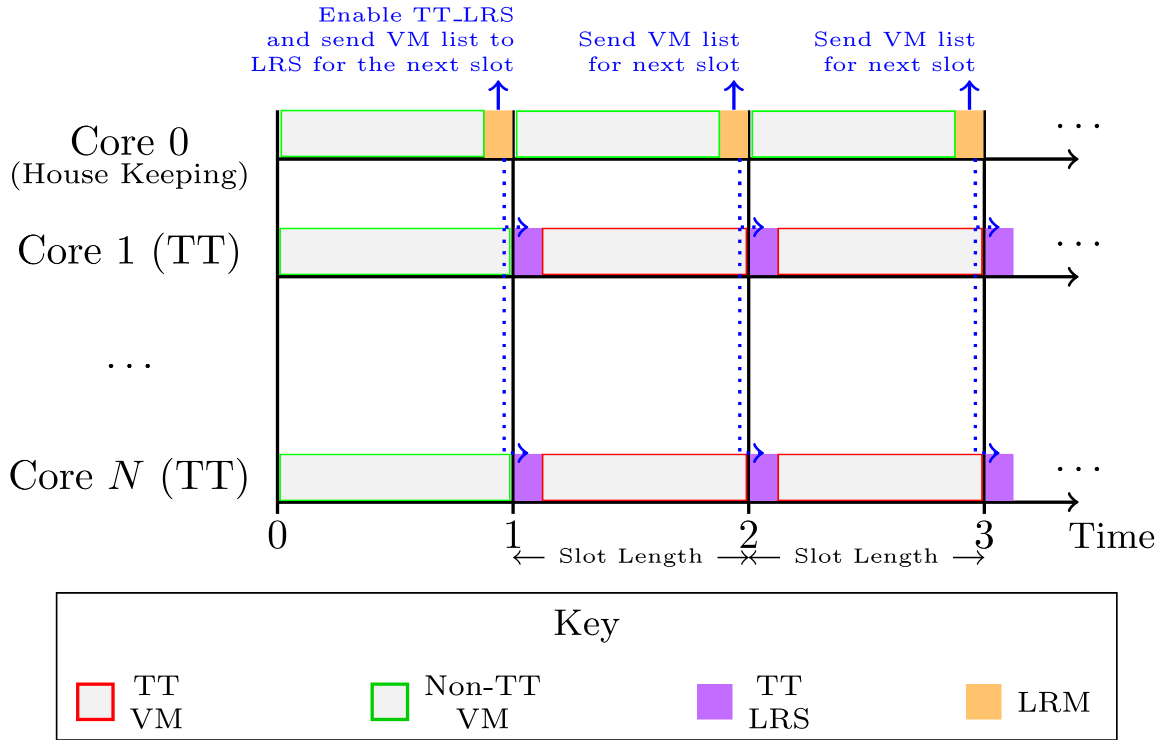


Figure VII.F10: *LRM and TT LRS*

VII.F10). Before start of a slot S_s, the LRM provides the TT-LRS on each core with a TT VMs (if available) as per the mapping M_s to run in S_s. The LRM itself runs as a real-time Linux task on a non-TT (housekeeping) core and has a period equal to one slot length (Δt_{Slot}). Hence, an LRM instance occurs once every slot.

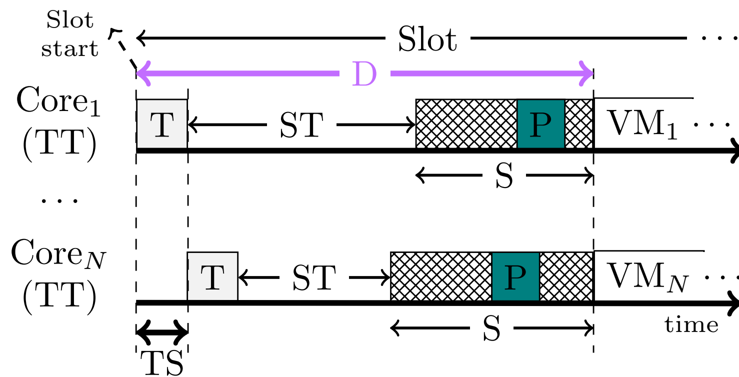


Figure VII.F11: *TT Dispatcher Overheads*

The experiments were performed on an Intel i6500 CPU (4 physical cores) running at 3.2 GHz with 8GB RAM. Core 0 was used as a housekeeping core, while core 1 to 3 were used to schedule the TT VMs. We measured the maximum observed overhead

VM Migration	T (Task tick)	TS (Tick skew)	ST (schedule trigger)	S (__schedule)	P(pick next task TT)	D (Total duration)
Disabled	0.44	0.88	0.75	0.94	0.35	1.74
Enabled	0.47	0.34	0.69	2.31	1.58	2.9

Table VII.T6: *Maximum Observed TT LRS Overhead (μs)*

for using the TT-LRS with migration between CPU cores enabled or disabled (via the offline scheduling table). The overhead includes the time taken for the complete Linux scheduler including the time for executing the TT-LRS. We denote the overall maximum observed overhead by D in Figure VII.F11 and Table VII.T6.

The Linux scheduler_tick() function is called regularly by a timer interrupt on each core. scheduler_tick() function calls the task_tick_TT() function of our TT-LRS (on each core). At start of each slot, the task_tick_TT() function checks if the LRM has assigned a new VM for its CPU core. If there is a new VM available, then it marks the core for rescheduling. We measure the scheduler_tick() function overhead including the time for task_tick_TT() function. We denote this overhead by T in Figure VII.F11 and Table VII.T6. In addition, we observed that the timer interrupt for the Linux scheduler_tick() does not occur synchronously on each core due to a small clock skew between the cores. We denote the overhead due to this skew by TS.

If a core is marked for rescheduling, the Linux kernel triggers the __schedule() function, the main entry point to the Linux task (VM) scheduler. We denote the time it takes for the Linux kernel to trigger the __schedule() function by ST in Figure VII.F11 and Table VII.T6. To decide which process to run next, __schedule() uses the pick_next_task_TT() function of our TT-LRS (on each core). pick_next_task_TT() returns the task_struct of the VM to be scheduled in the upcoming slot. In turn, __schedule() schedules this VM on the core. We denote the time taken to execute the __schedule() and the pick_next_task_TT() functions by S and P. If pick_next_task_TT() returns null on a core, i.e., LRM has assigned no TT-VM for that slot, then __schedule() checks for an available non-TT-VM (Linux process) and schedules it. This non-TT-VM can run until the end of the current slot.

Based on the global Slot-Shifting algorithm by Schorr and Fohler [231], we select the slot size as $10ms$. The overall maximum observed overhead to dispatch new VMs with migration enabled and disabled is $1.74\mu s$ and $2.9\mu s$ (0.017% and 0.029% of the slot size), which is considerably better than inbuilt Linux schedulers and slightly better than that reported for tableau scheduler with Xen [229]. Graphs in Figures VII.F13 and VII.F12 show the results in more detail.

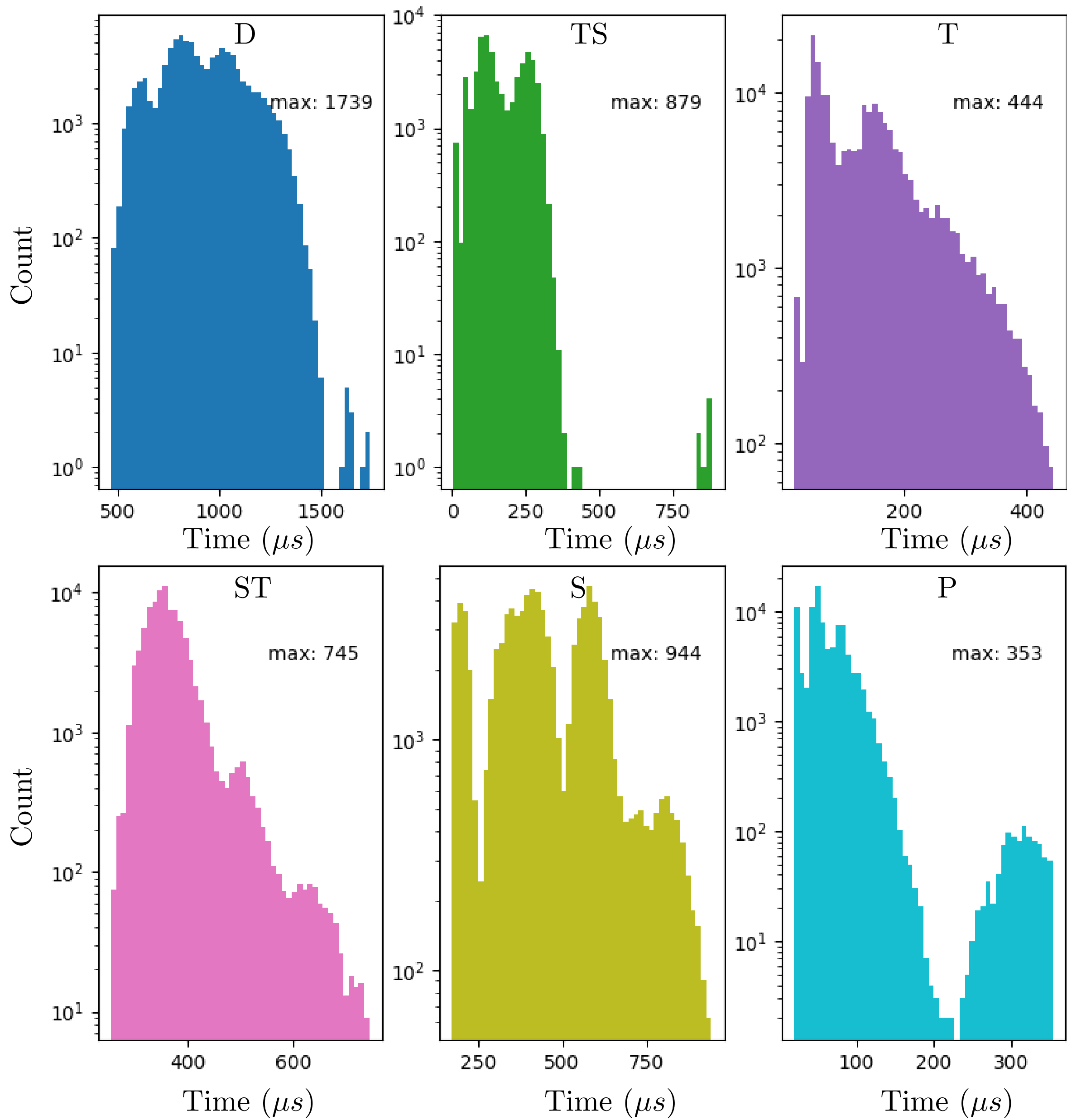


Figure VII.F12: *Overheads for TT-LRS with No Migration Allowed*

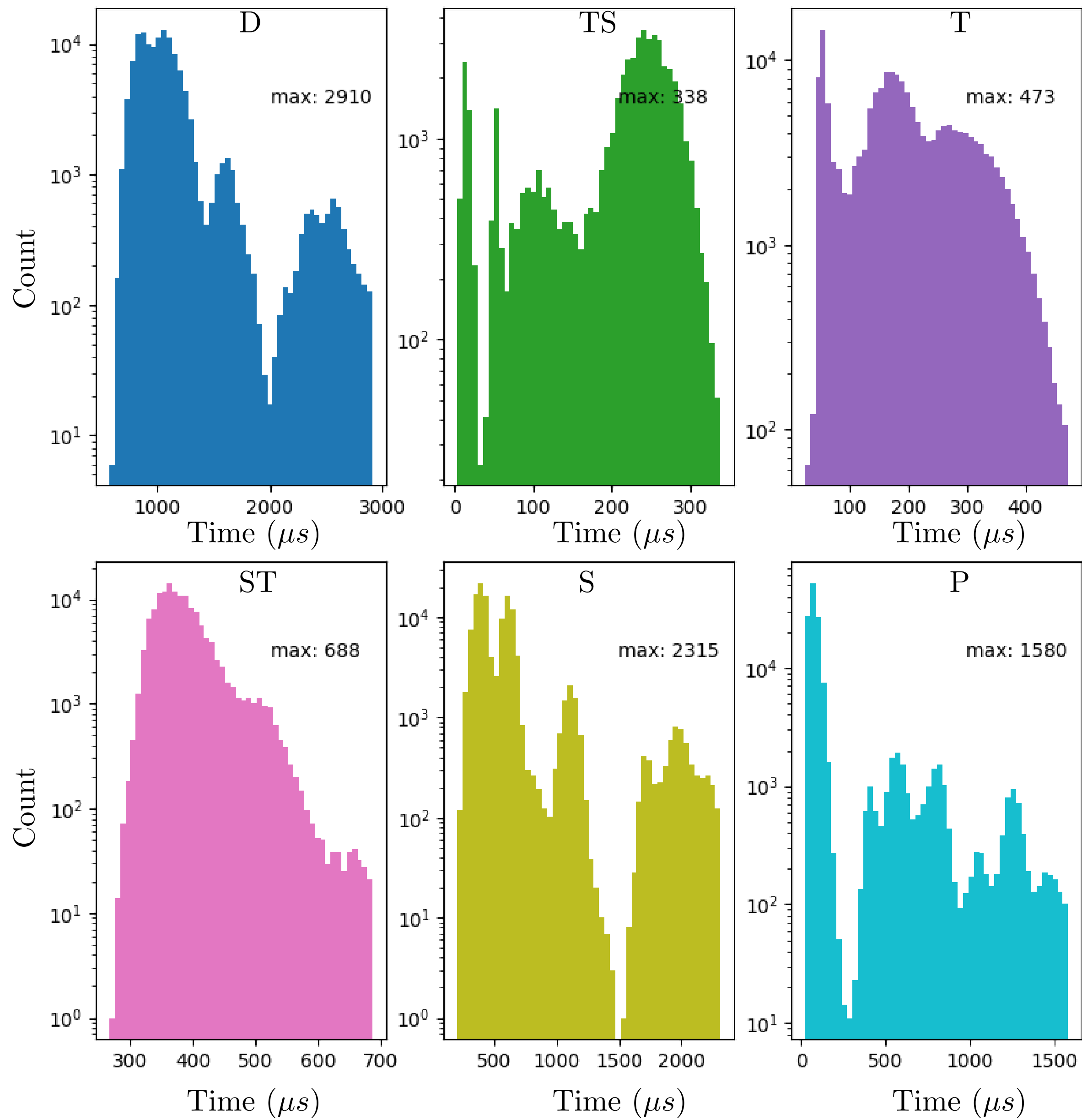


Figure VII.F13: *Overheads for TT-LRS with Migration Allowed*

Discussion - Adding Runtime Flexibility to TT Schedule in Cloud Nodes

To add new Event-Triggered (ET) VMs to the scheduling table at runtime, we propose an initial idea for an algorithm inspired by the global Slot-Shifting [231] to the LRMs.

We propose to group a set of consecutive slots into a *capacity interval* (or simply *interval* for short). We define an interval as a set of consecutive slots that possess the same mapping M of VM(s) to CPU cores. In other words, a new interval starts on slot S_s when the set of VMs assigned to this slot are different from the ones assigned to the previous slot (S_{s-1}). Intervals may contain some cores that have not been assigned any VM, resulting in idle slots within the interval. These slots define the *Spare CPU Capacity* of an interval ι_f ($SCPU_f$).Furthermore, each core p is reserved memory bandwidth BW_f^p depending on which VM it was assigned for interval ι_f. The unreserved memory bandwidth for ι_f is denoted as SBW_f.

An LRM can use $SCPU_f$ and SBW_f to add an ET VMs to the mapping M at runtime. To do so, the LRM must execute an *acceptance test* and a *guarantee routine* similar to Slot-Shifting [231]. When a new VM arrives, the acceptance test determines if there is enough available $SCPU$ and SBW in the intervals before the VM's deadline. If the acceptance test is successful, the guarantee routine is executed to add the VM to the mapping M and update the $SCPU$ and SBW of all affected intervals (and slots). As a result, this routine shall guarantee the resource allocation to the new accepted VM. Note that, unlike previous work, the new acceptance test and guarantee routine should take into account the node's SBW and the VM's ϕ. We leave the detailed implementation of the modified acceptance test and guarantee routine to future work.

If an LRM is unable to accept (and guarantee) a new ET VM due to unavailability of resources, the LRM requests the GRM for cloud-wide adaptation. The GRM, in turn, searches for another node with more appropriate resource availability and dispatches the VM to that node. Similarly, if a node is unable to execute an existing VM or the VM does not achieve the desired QoS due to a change in availability of resources, the LRM informs this to the GRM. The GRM, in turn, redeploys the VM to another node in the cloud. The change in availability of the resources can result from failure or a change in the demand for resources.

VII.7 Intel Memory Bandwidth Allocation (MBA) Evaluation

Preventive methods are often used for memory bandwidth regulation, for example, Memguard [88]. These methods limit the number of memory requests that each CPU core can issue. The memory accesses performed by a core are determined by using a Performance Monitor Counter (PMC), usually programmed to count the LLC misses. However, PMCs can be highly accurate or highly inaccurate. Their accuracy depends upon how they are accessed, the application running on the core, and the measured event [232]. Moreover, existing approaches have high overheads and do not scale well with the increase in CPU cores. In addition, they reduce the performance of the CPU. For example, suppose a core utilizes the assigned number of memory accesses in the regulation period. In that case, the application running on the core is preempted, and

the core is forced to idle for the remaining time of the memory bandwidth regulation period, thus reducing the CPU performance.

As explained in Section IV.6.3, Intel has introduced a new programmable bandwidth controller in the 2nd Gen. Intel Xeon processors [30] as part of the MBA hardware architecture. This approach does not require interrupting cores to regulate the memory bandwidth. In theis Section, we present some initial results from our evaluation of Memory Bandwidth Allocation (MBA) capability to regulate the memory bandwidth.

As explained in Section IV.7.3, we developed an LRS to interface with this hardware feature. We used this LRS to set the MBA delay values and observe the effect on a synthetic memory read benchmark. We ran the benchmark application on a fixed core (core 2) of an Intel Xeon Silver 4208 processor ($2.10GHz$). We assigned a $512KB$ array to the benchmark application. The benchmark application accessed memory (data load) with $64bytes$ steps (= LLC line size). This ensured cache misses and stressed the memory bandwidth. We performed the experiments under two scenarios:

- Scenario *No-Write-Back (NWB)*: In this scenario, the LLC was (almost) empty before the benchmark was executed. Each memory access from the benchmark caused an LLC miss. As a result the corresponding cache line was fetched from the memory and stored in the LLC.

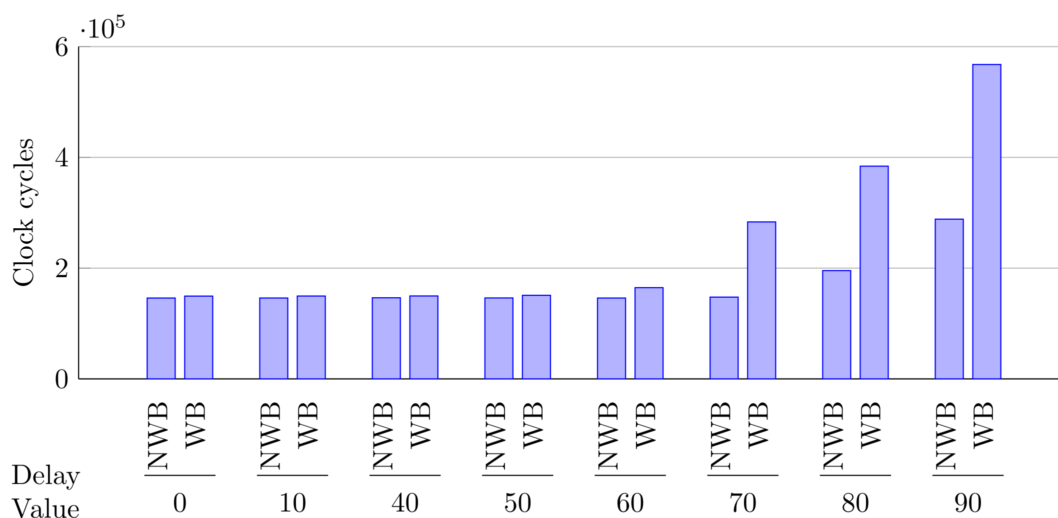
- Scenario *Write-Back (WB)*: In this scenario, the LLC was completely filled with random data before the benchmark was executed. Each memory access from the benchmark caused an LLC miss. As the LLC was completely full, an existing cache line was evicted before a new cache line was fetched from the memory and stored in the LLC. The evicted cache line caused additional memory traffic due to write-backs.

The main aim of the experiment was to observe the variability in the execution time of the benchmark and the number of memory accesses performed by the benchmark with different MBA delay values. We measure the execution time by using a hardware MON (a core PMU programmed to count clock cycles). Figure VII.F14a shows the observed execution time of the benchmark in both scenarios with different delay values based on 60 runs of the benchmark. As per the errata published by Intel [194], delay values > 10 and < 40 written to the MBA delay value register (MSR 0xD50 to 0xD57) may be read back as 10. We experimentally confirmed that this issue is present in our platform. Hence, we do not show delay values 20 and 30 in the graph. As observed in the graph, only delay values of 70, 80 and 90 showed considerable change in the execution time of the benchmark as compared to 0 delay.

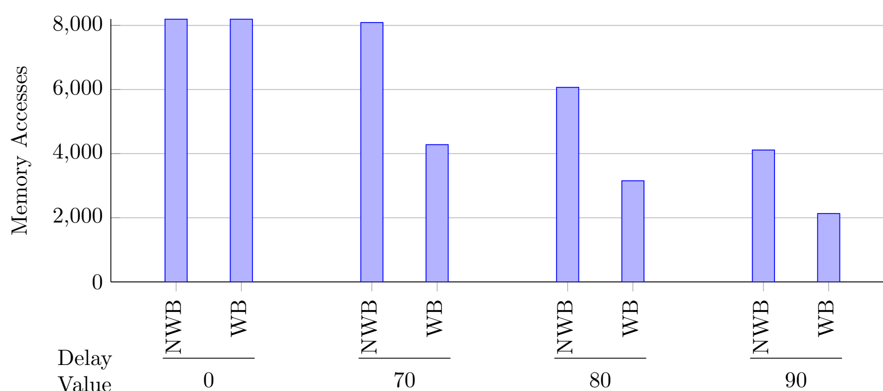
Furthermore, we performed experiments to compare the number of memory accesses the benchmark could perform in a fix amount of time (= to time taken to perform 8192 memory accesses with no delay). We count the memory access by using a hardware MON (an uncore PMU programmed to count read Column Access Strobe (CAS) of the Integrated Memory Controller (IMC)). We only considered delay values of 70, 80 and 90 as they showed change in the execution time. Figure VII.F14b shows the results based on 60 runs of the benchmark. The number of memory accesses between the two scenarios

reduces by almost half in each case. This shows that MBA inserts delay to the memory fetches as well as write-backs. This is an additional advantage over existing techniques that are based on LLC misses and ignore memory bandwidth consumed by write-backs.

As part of the ongoing work, we are in the process of experimentally analyzing the impact of multicore contention on MBA and determining how to use MBA for allocating memory bandwidth to cores running safety-critical applications/VMs for ensuring that they meet their deadlines.



(a) *Variablity in Maximum Execution Time*



(b) *Number of Memory Accesses in a Fixed Time Period*

Figure VII.F14: *Impact of MBA Delay Values on a Benchmark*

VII.8 Distributed Global Resource Management Evaluation

In this section, we give a proof-of-concept demonstration and measure the delay for global reconfiguration with the safe and secure blockchain-based distributed global resource management (Section V.6) using a core failure scenario. The core failure scenario is similar

to the one we considered for evaluation in the avionics use case (Section VI.5.1). However, here we consider a KVM-based cloud with up to 12 Sawtooth-enabled (simulated) nodes with DGRM smart contract for global resource management (instead of a single central GRM).

As a reminder, we designed a new Sawtooth transaction family called the Resource Manager (RM) transaction family (Figure V.F8b on Page 137) for implementing the DGRM and the LRM on the nodes. The Resource Manager (RM) transaction family consists of the following:

1. An LRM Client (CLI) to handle the local resource management logic.

2. A data model to record the cloud-wide system state in the blockchain.

3. A DGRM smart contract (Transaction Processor (TP)) that runs in the blockchain and handles the global resource management logic for distributed decision-making.

We implemented the LRM CLI in a VM running Ubuntu server 18.04 LTS and Sawtooth v1.1 on a dedicated core. In reality, only one LRM CLI VM (not including replicas) must be present on a node. However, due to limitation of available experimental hardware, we implemented 4 LRM CLIs on the first node (16 core Intel Xeon processor running at 2.3 GHz). Each LRM CLI on this node managed a set of four fixed cores. From these four cores, each LRM ran on a dedicated (housekeeping) core while the other three cores were used to execute TT VMs. We simulated another 8 LRM CLIs on a second node (8 core Intel Xeon processor running at 2.3 GHz). Each simulated LRM CLI ran in a VM on a dedicated core of the second node.

Initially, we provided each LRM with a local reconfiguration graph. Each local configuration contained a per slot mapping of TT VMs to TT CPU cores. Linux scheduler (of KVM) ran core failure MONs together with TT-LRS on each slot boundary. LRMs detected core failure via these MONs. Based on the information of the failed cores and the mapping of VMs to cores, each LRM locally maintained a list of available cores. For simplicity of demonstration, we assume the housekeeping core does not fail. However, in reality this situation is possible. We can deal with it by replicating LRM instances on other cores in a way similar to that explained in Section IV.9.1.

For the simulated LRMs on the second node, we used simulated core failure MONs and TT-LRS. The simulated MONs gave simulated information to their respective LRM CLIs about core failures, while the simulated LRSs gave the LRM CLIs an impression that they have scheduled VMs as per the LRM directives.

All Sawtooth enabled VMs are connected by a 100Mbps (external) router. The data model stored in the blockchain consists of the current configuration of each LRM (node). A DGRM smart contract instance is present together with each LRM CLI VM and executes on the blockchain. Together, the DGRM smart contract instances maintain the current configuration of each LRM in the blockchain.

Similar to the update messages in the avionics use case (Section VI.3.5), LRM CLIs generate two kinds of Sawtooth transactions:

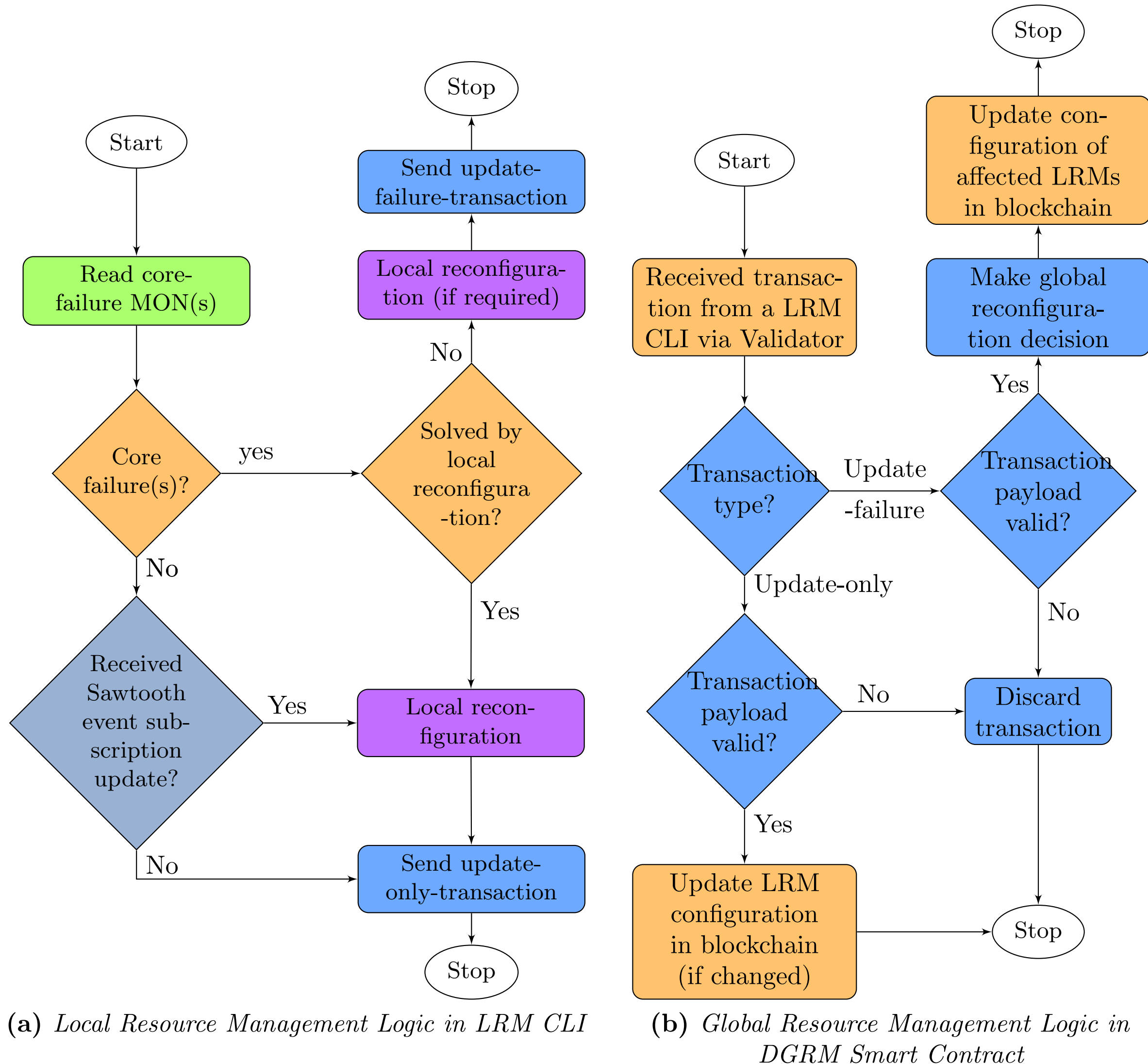
1. *Update-Only-Transactions* to send an update to the DGRM smart contract (via the validator) when any changes occur to the availability of CPU cores. This transaction consists of the current LRM ID (unique) and the current configuration.
2. *Update-Failure-transaction* to notify when the LRM CLI requires a global (cloud-wide) reconfiguration. This transaction also consists of the current LRM ID and the current configuration.

When a LRM CLI detects a core failure (via the MON), it changes to a new configuration with reduced number of cores and sends an update-only-transaction via the Sawtooth validator. Similar to the avionics use case, when the LRM CLI reconfigured to a configuration that could not host all the allocated VMs, it requires assistance from the global resource management. In this case, the LRM CLI sends an update-failure-transaction to the DGRM smart contract via the Sawtooth validator.

In both cases, the transaction is processed via the Validator and DGRM smart contracts as explained in Section V.7.1. When the DGRMs receives a transaction, they process it in a distributed manner as per the type of the transaction. For an update-only-transaction, the DGRMs perform a sanity check and update the current configuration of the LRM accordingly in the blockchain (based on Practical Byzantine Fault Tolerance (PBFT) consensus). For an *update-failure-transaction*, the DGRMs perform an additional step. They also search for a new global configuration similar to the avionics use case. When the DGRMs finds a new global configuration, they update the configuration of the involved LRM in the blockchain.

Based on the Sawtooth event subscription for changes to the state space of the LRM configurations in the blockchain, the LRM CLIs involved in the reconfiguration received a notification about the change in the configuration. This notification acts as the order message. The LRM CLIs further instruct their respective LRS according to the new configuration. Figure VII.F15 shows the local and global resource management logic that we implemented in the LRM CLI and the DGRM smart contract of the RM transaction family.

To measure the global reconfiguration delay, we used the Hardware MONs (with PMU programmed to count the clock cycles). We started measuring the time in a resource management VM when a LRM CLI sent an Update-failure-transaction and stopped measuring the time when the DGRM smart contracts assigned a new configuration and updated the blockchain. To find out when to stop measuring, we enabled Sawtooth event subscriptions on the node sending the update-failure-transaction to get a notification when any change in configuration occurs for address space in the the blockchain corresponding to any LRM. The plot in Figure VII.F16 shows the delay observed for 1000 global reconfiguration for system with 4, 8, and 12 DGRMs participating in making global resource management decisions. As observed in the plot, the time for global re-orchestration scales well with an increase in the participating DGRMs (Sawtooth enabled nodes). Depending on the number of DGRMs, use cases that can tolerate a maximum global re-orchestration delay as shown in the graph can benefit from this approach.



(a) *Local Resource Management Logic in LRM CLI* **(b)** *Global Resource Management Logic in DGRM Smart Contract*

Figure VII.F15: *Use Case Specific Resource Management Logic of the RM transaction family*

We tested the global resource management safety and security in multiple experiments by injecting faults in an DGRM instance. We also experimented by shutting down some Sawtooth enabled VMs so that they cannot participate in global decision-making. Besides, we created a malicious VM sending false transactions. The blockchain-based distributed global resource management functioned correctly in all experiments and the LRM configuration stored in the blockchain was not lost on shutting down up to f Sawtooth enabled VMs.

We assume non-critical applications also use the blockchain. The blockchain is not present only for use by global resource management. Thus, we did not measure the overhead of the blockchain. However, a previous work [233] presented such overheads.

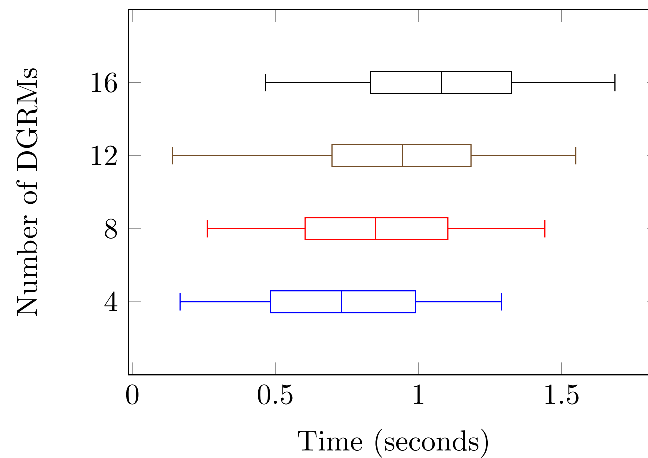


Figure VII.F16: *Global Reconfiguration Delay*

VII.8.1 Discussion

In the avionics use case, global reconfiguration delay in a core-failure scenario for a system with three multicore nodes was equal to one Major Frame (MaF) of the system. In the experiment for blockchain-based distributed global resource management, the worst case observed delay for a similar scenario was $1.29s$. We cannot compare the results directly as the avionics use case had time-triggered communication resulting in a deterministic global reconfiguration delay. Here we aimed at dynamic Real-Time Systems (RTS) or Mixed-Critical Systems (MCS), such as railways and industrial automation, with focus on safety and security and lower overheads for global reconfiguration delay.

It should be noted that the distributed global resource management only decides on the global configuration. The LRSs perform the actual node-level resource scheduling based on the global reconfiguration orders and local reconfiguration graph of their LRMs. The LRSs ensure that the system meets the demand of resources of the allocated real-time VMs. Hence, the global resource reconfiguration decision-making being non-deterministic does not impact real-time VMs that are currently executing on each node.

VIII

Conclusion

"Finally, in conclusion, let me say just this."

– Peter Sellers

Real-Time Systems (RTS) and Mixed-Critical Systems (MCS) with dynamically changing availability and demand of resources need global resource management to coordinate and dynamically adapt system-wide resource allocations. In addition, resource management can dynamically adapt applications to changing availability of resources and maintains a global (system-wide) view of resources and applications. The overall aim of the resource management is to 1) ensure real-time applications meet their end-to-end deadlines even in the presence of faults and changing environmental conditions, 2) ensure efficient resource utilization to improve the Quality of Service (QoS) of co-executing Best-Effort (BE) (or non-critical) applications.

In this dissertation, we proposed a domain-independent global resource management framework for distributed MCS and RTS consisting of heterogeneous nodes based on multicore processors or Multi-Processor System on Chips (MPSoCs). We initially developed the framework with the French Aerospace Lab - ONERA and Thales Research & Technology during the DREAMS project and later extended it during SECREDAS and other internal projects.

A single fault in global resource management can render it useless. In the worst case, it can make wrong resource management decisions leading to a deadline miss in real-time applications. With the advent of Industry 4.0, cloud computing, and Internet of Things (IoT), it has become essential to combine stringent real-time constraints and reliability requirements with the need for an open-world assumption. As a result, the global resource management for these systems becomes an inviting target for passive and active attackers as it can actively decide on the system's resource management. Hence, unlike previous resource management frameworks MCS and RTS, we considered both safety and security for the framework itself.

To enable real-time industries to use cloud computing and enter a new market segment – real-time operation as a cloud-based service, we proposed a Real-Time Cloud (RT-Cloud) based on global resource management for hosting RTS and MCS.

Finally, we considered two actual industrial use cases for the resource management framework. We presented an avionics use case to evaluate the global resource management

framework for use with a MCS and a railway use case to motivate the use of RT-Clouds with global resource management.

VIII.1 Overview of Contributions

VIII.1.1 Resource Management Framework for MCS and RTS

Our resource management framework combines the benefits of local and global resource management strategies and keeps the overheads low by decoupling global resource management from local resource management. The resource management architecture consists of a Global Resource Manager (GRM) in combination with a set of Local Resource Managers (LRMs).

Our resource management framework efficiently reallocates the resources and adapts the QoS or modes of applications upon fluctuations and changes in operating conditions. Furthermore, it supports the reallocation of resources at runtime upon the occurrence of resource failures. In addition, it provides 1) monitoring service to monitor the behavior of applications, and availability or operational status of resources, 2) scheduling service to deterministically schedule access from applications to resources and ensure the application requirements are met, and 3) local and global reconfiguration services to allocate resources and adapt applications based on the current availability of resources and the operational conditions.

Our resource management architecture allows multiple monitoring and scheduling techniques without tightly coupling them with the implementation of the framework. Such a design allows a system designer to select the appropriate monitoring and scheduling technique for each resource as per requirement without significant modifications to the framework implementation. We achieve this via Local Resource Monitor (MON) modules that monitor resources and applications, and Local Resource Scheduler (LRS) modules that schedule and control access of applications to resources. These modules can be plugged in each LRM.

Two resource management architectures are possible –

1. A flat architecture where the GRM is at the top of the hierarchy with a complete view of the entire system and directly supervises and controls all the LRMs.

2. A hierarchical architecture where the GRM sits at the top of the hierarchy and LRMs are present at different levels in the hierarchy. The GRM directly communicates with the LRMs at the second-highest level of the hierarchy. In turn, these LRMs communicate with the LRMs below them. Each LRM communicating to another LRM or set of LRMs introduces a new level in the architecture. The hierarchical architecture allows the LRMs to act as a granularity interface and hides fine-grained activities of a sub-system from the GRM's view. As a result, the GRM receives a limited number of resource updates. The LRMs send reconfiguration requests to the GRM only when a reconfiguration of the entire system is necessary. The final result is a scalable resource management architecture that manages a distributed system

consisting of heterogeneous nodes with different operating speeds and locations in the system structure.

We took advantage of the hierarchical resource management architecture to introduce the concept of the resource management domains for effective reconfiguration at all levels. We considered five different domains in the system structure to perform resource management – System Domain, Cluster Domain, Node Domain, Virtualization Domain, and Virtual Machine (VM)/application Domain. The domains represent the composition of the system from the resource management perspective. Conceptually, they also correspond to the architecture's hierarchy levels.

It can be error-prone and tedious for a system designer to correctly configure resource management according to each platform's low-level details in heterogeneous distributed systems. Therefore, our resource management framework allows the system designers to provide resource management configuration parameters abstractly and select MONs and LRSs for each platform without the need to know or set fine-grained platform-specific configurations.

VIII.1.2 Local Resource Monitor (MON)

We designed and implemented the following new MONs:

- MONs to interface with the hardware-specific monitoring features such as Performance Monitor Unit (PMU) and Intel Memory Bandwidth Monitoring (MBM) and Cache Monitoring Technology (CMT). We proposed methods to use these MON with XtratuM hypervisor and Linux/Kernel Virtual Machine (KVM) to gather information about VMs/tasks.

- MON[1] to detect permanent core failures on multicore platforms.

- MON[1] for detecting potential deadline overrun by a critical VM in the presence of concurrently executing non-critical VMs.

- MON that can use the XtratuM Health Monitoring (HM) API to write partition status or errors (not handled by XtratuM) to a HM log.

VIII.1.3 Local Resource Scheduler (LRS)

We designed and implemented the following new LRSs:

- LRS[1] for scheduling tasks of a critical application running in a XtratuM hypervisor partition.

- LRS[1] to assist the LRM in reconfiguring scheduling plan (modes) of XtratuM hypervisor.

[1]Designed and implemented together with ONERA and Thales R&T.

- LRS to provide an interface for the resource management to interact with Intel Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) hardware features.
- LRS to support scheduling of Time-Triggered (TT) tasks/VMs in Linux/KVM (with PREEMPT_RT patch).

VIII.1.4 Resource Management Policies

Fault-Tolerance via Local and Global Reconfiguration in Distributed Systems with Multicore nodes

When a core fails, the applications or VMs hosted on it cannot execute further. Since modern multicore platforms provide a high number of CPU cores, we decided to provide a strategy[1] to manage core failures via reconfiguration.

Upon detection of a core failure (via a MON), a LRM performs local reconfiguration based on online or offline methods to reallocate the cores to applications or VMs. To protect the LRM of a node itself against core failure, we proposed an approach based on synchronous execution of LRM replicas on each core.

If an LRM alone cannot host all applications based on the new availability of core, then it requests the GRM for a global reconfiguration. Based on the global view of the system maintained by the GRM and offline or online methods, the GRM determines a new global configuration. The new configuration redeploys the applications that could not be hosted to new nodes. The GRM only makes the reconfiguration decision and informs the LRMs of the nodes involved in the global reconfiguration. These LRMs, in turn, apply the new configuration via their LRSs.

In the avionics use case, we implemented reconfiguration based on mode[2] changes. We assumed that all applications that need redeployment must be moved entirely to a different node and that the GRM is fail-safe. We provided the LRMs and GRM with local and global reconfiguration graphs obtained offline via the DREAMS toolchain (Xoncrete and GREC tools). For network reconfiguration, we selected the super schedule approach. The super schedule was generated by the modified TTPlan tool of the DREAMS toolchain. Upon detection of core failure via a MON, an LRM applies new modes (based on the local reconfiguration graph) by using the LRS for reconfiguring XtratuM hypervisor scheduling plans. As far as resource availability allows, the LRM selects a new mode so that the critical applications can be locally reconfigured upon core failure. Suppose any non-critical applications cannot be locally reconfigured on the node. In that case, the LRM sends a reconfiguration request to the GRM, which in turn selects a new global configuration (based on the global reconfiguration graph). It informs the LRM involved in the global reconfiguration to apply new local configurations on their nodes.

We successfully validated our approach using the avionics demonstrator and experimentally evaluated the worst-case observed overheads for the MON, the LRM (including the LRS for reconfiguration), and the GRM. In addition, we ensured determinism for the maximum global reconfiguration delay. It is equal to one or two Major Frame (MaF)

[2] Also referred to as configurations or plans by hypervisors

depending on the placement of the GRM with respect to the LRMs and the TT-Virtual Link (VL) periods for the update and order channels.

Simultaneous Execution of Mixed-Criticality (Avionics) Applications by Local Temporal Overload Management

If we provide resources to critical applications based on their Worst-case Execution Time (WCET), we under-utilize the multicore platform in an average case as the Average Execution Time (AET) is much lower than the WCET. Thus, we designed a strategy[1] for an LRM to support over-utilization of a multicore platform based on potential deadline overrun monitoring of critical applications. When a deadline overrun MON detects a potential deadline miss, the LRM interrupts best-effort applications to ensure that the critical ones can execute safely. The LRM resumes the best-effort applications once the application finishes its execution in the current hyperperiod/MaF.

We proposed two different approaches for executing the potential deadline overrun MON instances. The easiest option is to place them in between two tasks of a critical application. The main advantage of this approach is that it does not require the instrumentation of critical applications. However, the MONs can only detect potential deadline overruns at task boundaries. Another option is to place observation points inside each task of the application. In this approach, the MONs can detect potential deadline overruns sooner than the previous approach. However, this approach requires the instrumentation of critical application tasks.

We proposed three different approaches for executing the LRM instances for potential deadline overrun management. An immediate adaptation is possible if we dedicate an entire core to execute the LRM. However, the CPU performance of the node reduces as an entire core is dedicated to the LRM. To improve CPU utilization, we need to execute LRM instances at predefined time slots. However, it has a much longer delay from the time the MON detects the overrun to the time the LRM applies the adaptation. In the third approach, we provide adaptation by application/VM domain LRM instances executing directly after the MON instances. The adaptation is almost immediate, and the performance impact is low. The only restriction on the use of this approach is that the underlying hypervisor must allow reconfiguration from within a VM.

In the avionics use case, we placed the potential deadline overrun MONs in between two tasks of a critical application to avoid task instrumentation. In addition, we placed the LRM instances directly after the MON instances to keep the overhead low and ensure fast adaptation. Due to confidentiality issues, only Thales R&T experimentally evaluated the approach with the avionics use case. The deadline overrun management helped to reduce critical application slot deadline significantly for all three avionics critical applications) compared to the interference scenario with no management. In all experiments, the local resource management ensured that the observed execution time remained below the critical application slot deadline while the non-critical applications still achieved considerable QoS.

VIII.1.5 Resource Management Communication

The GRM and the LRMs communicate with each other to exchange resource management information. We proposed three conceptual communication channels between the central GRM and the LRMs (and between higher and lower domain LRMs) – Update, Order and Membership channels.

An LRM uses an update channel to send status updates and global reconfiguration requests to the GRM (or a higher domain LRM). The GRM (or a higher domain LRM) uses an order channel to send reconfiguration orders to an LRM. An LRM uses the membership channel to send heartbeat messages to the GRM for membership and failure detection purposes. If an LRM sends periodic updates to the higher domain resource manager, then the update channel messages also serve as the Membership messages.

Security Sublayer for Resource Management

Unsecured resource management has several weak spots that an attacker can exploit. Therefore, we[3] analyzed the resource management communication, LRM, and GRM from a security viewpoint. Resource management requires 1) confidentiality to ensure the privacy of information, 2) integrity to ensure that data is not modified, 3) authenticity to ensure that data is genuine and that the actual origin of the data is the same as the claimed origin, and 4) access control to allow access based on permissions.

We proposed security services for resource management communication to prevent various security attacks. We provided the four essential security services by the cryptographic mechanisms. For ensuring integrity, authenticity, and access control, we used authentication codes. In addition, we used encryption to provide confidentiality.

We proposed three different security levels for resource management communication. Level 0 provides no security. Level 1 security service provides integrity to prevent the manipulation of resource management messages as well as authenticity to verify the source of the messages and ensure that the received messages originated from a trusted resource manager. Level 2 provides confidentiality in addition to Level 1. In addition, levels 1 and 2 ensure protection against replay attacks by using a time-varying code (both levels provide integrity service to check the time-varying code). These security levels are helpful depending on the amount of security required for different use cases. Moreover, Level 1 is suitable for the membership channel where there is no confidential information, while Level 2 is suitable for update and order channels.

We proposed two options to implement the security services – implementation as a support module or implementation as a layer between the resource managers and the underlying hardware or software (hypervisor or Operating System (OS)). The two implementations are different only in terms of the handling of the communication. We implemented the second option as it supports the direct communication between security sublayers of different nodes and provides transparency for the resource managers. Contrarily, in an implementation with a security support module, the resource managers must handle the communication while the security module only provides security services. We[3] decided to implement two options for algorithms in the security sublayer of the

[3]Based on inputs from security experts at the University of Siegen

resource management framework – ChaCha20-Poly1305 and CLEFIA (in Offset CodeBook (OCB) operation mode).

In the avionics use case, we experimentally evaluated the worst-case observed overheads for all security levels and both security algorithms. Overhead for security level 0 (no security) is the least, followed by security level 1 (authenticity and integrity). Security level 2 has the highest overhead as it adds confidentiality on top of level 1. Furthermore, the evaluation also indicates that the ChaCha20-Poly1305 algorithm is much faster than the CLEFIA algorithm. However, stream ciphers, such as ChaCha20, are generally used only in few applications and, thus, not closely scrutinized regarding security. Contrarily, block ciphers, such as CLEFIA, are heavily used in many applications, and thus, are more trusted. In the end, the trade-off between using ChaCha20-Poly1305 and CLEFIA lies in speed versus trust in the algorithm.

Using Existing Communication Protocols For Resource Management

Apart from using resource management communication with the security sublayer, we can use these existing communication protocols, especially in Linux-based use cases. We provided an overview about using two popular existing protocols, Message Queue Telemetry Transport (MQTT) and Open Platform Communications Unified Architecture (OPC UA), for resource management communication, taking into consideration requirements such as membership, security, and reliable message delivery.

VIII.1.6 Blockchain-Based Distributed Global Resource Management

We discussed several limitations of a single central global resource manager. We proposed extending the concepts to make global resource management safe and secure using distributed global decisions instead of centralized decisions. To do so, we eradicated the central GRM and added a new resource management component, the Distributed Global Resource Manager (DGRM) component, on all nodes with LRMs in the highest level of the hierarchy. The corresponding LRMs function is almost the same as before. The only difference is that instead of sending updates (including reconfiguration requests) to the central GRM, each LRM sends updates to its corresponding DGRM. Instead of the central GRM making the global decision on its own, the DGRM components on all these nodes work together for making global decisions based on distributed coordination. A similar concept applies to the flat resource management architecture.

If we used DGRMs instead of the central GRM, there is neither a single point of failure nor a bottleneck in the resource management anymore. However, we must ensure that DGRMs are safe and secure themselves. Therefore, we identified the challenges and requirements concerning safety and security for designing the DGRMs.

We choose a private (permissioned) blockchain called the Hyperledger Sawtooth [138] for our implementation as it helps us to meet the challenges and requirements concerning safety and security. We designed a new Sawtooth transaction family called the Resource Manager (RM)-transaction family for implementing the DGRMs and the LRMs of our resource management framework.

As a reminder, the distributed global resource management only decides on the global configuration. The LRSs perform the actual node-level resource scheduling based on the global reconfiguration orders and local directives of their LRMs. The LRSs ensure that the system meets the demand of resources of the allocated real-time VMs. Hence, the global resource reconfiguration decision-making can be implemented via a blockchain without impacting real-time applications or VMs that are currently executing on each node.

We gave a proof-of-concept demonstration and measured the delay for global reconfiguration with the distributed global resource management using a core failure scenario in a KVM-based cloud with up to 12 Sawtooth-enabled (simulated) nodes with DGRM smart contracts. We successfully validated the distributed global resource management and observed that the global reconfiguration delay increased with the increase in Sawtooth-enabled nodes with DGRM smart contract. In the experiments, the worst-case observed delay for a scenario similar to that in the avionics use case was $1.29s$. However, we cannot compare the results with the avionics use case as it had time-triggered communication. Table VIII.T1 highlights the differences of the block-based distributed global resource management and the single central GRM.

Global resource management	Use case	Deterministic node-level scheduling & allocation	Deterministic global reconfiguration delay	Fault-tolerance	Security
Single Central GRM	Avionics (Section VI.5.1)	Yes	Yes (with TT-Ethernet (TTE) – 1 or 2 MaF)	For LRMs via replication (Section IV.9.1)	Via security sublayer (Section V.2)
Blockchain-based DGRM (Section V.6)	RT-Cloud (Section VII.8)	Yes	No (max. observed delay in Figure VII.F16)	Byzantine fault-tolerance for global-level, For LRMs via replication	Yes (via secure Sawtooh network)

Table VIII.T1: *Distributed vs. Central Global Resource Management*

VIII.1.7 Safety-Critical Railway Operation as a Cloud-Based Service

Cloud computing is rapidly gaining popularity in many domains as they provide benefits such as higher availability, scalability, and efficient hardware resource utilization. We explored virtualization technologies and cloud computing for migrating an existing real-time safety-critical railway use case from dedicated hardware solutions. We considered four existing cloud virtualization technologies (hypervisors) – KVM, Xen, Microsoft

Hyper-V, and VMware ESXi, for use with safety-critical railway VM running the Thales TAS control platform.

Based on the qualitative analysis, we concluded that KVM and Xen are better suited than ESXi or Hyer-V. ESXi and Hyer-V either miss certain features or have limited support for essential features such as predefined CPU models, Libvirt, and real-time scheduling options. Moreover, they do not allow us to benefit from the Intel Xeon-specific features. Finally, since they are closed source, we cannot add missing or new functionalities or resource management features as per requirement. To further narrow down our choice, we performed a quantitative analysis of Xen and KVM by running benchmarks to compare their performance.

We proposed a KVM-based private RT-Cloud containing three zones to increase availability, improve fault-tolerant VMs and help protect against unexpected failures. As observed in the quantitative analysis, when we executed multiple VMs in parallel with any hypervisor, there was contention in the shared resources. Hence, we proposed using our resource management framework to monitor, control, and coordinate the cloud nodes.

Taking into account the qualitative and the quantitative analysis performed as part of this dissertation, Thales Austria GmbH selected KVM as the hypervisor of choice to host the safety-critical railway VMs. Thales in coordination with the Austrian federal railways (ÖBB) successfully demonstrated the feasibility of using KVM to host a virtualized Triple Modular Redundancy (TMR) Radio Block Center (RBC) application (with TAS Control Platform).

VIII.1.8 Time-Triggered (TT) Scheduling in Cloud Nodes

To support TT scheduling of VMs, we proposed to use our TT-LRS with an LRM of a RT-Cloud node. We provided a CPU-and memory bandwidth-aware offline scheduler that generates a TT scheduling table for use with an LRM and a TT-LRS.

The TT-LRS introduced a new scheduling policy in KVM (Linux). The new scheduling policy has the highest priority and sits on top of the hierarchy of (native) Linux scheduling classes. If there are no runnable TT class VMs (or Linux processes), the Linux scheduler looks for runnable VM (or Linux processes) in each class in decreasing priority order. Unlike the existing scheduling modules in Linux, the LRM (running on the housekeeping core) must explicitly enable the TT-LRS on a set of CPU cores at runtime. Before the start of a slot, the LRM provides the TT-LRS on each core with a TT VMs. The LRM itself runs as a real-time periodic task on a non-TT (housekeeping) core and has a period equal to one slot length.

We measured the maximum observed overhead for using the TT-LRS with migration between CPU cores enabled or disabled. The overhead included the time taken for the complete Linux scheduler, including the time for executing the TT-LRS. The observed results were considerably better than inbuilt Linux scheduling classes and slightly better than that reported for the Tableau scheduler with Xen.

VIII.2 Ongoing and Future Work

"Every new beginning comes from some other beginning's end."

– Seneca

VIII.2.1 Adding Flexibility to Time-Triggered (TT) Scheduling on RT-Cloud Nodes

Existing TT approaches for cloud nodes, such as Tableau [229] or ARINC-653 scheduler for Xen, require high-overhead table regeneration at runtime for adding new VMs. They do not have the flexibility of appending new VMs to the existing table at runtime, thus, undermining some important advantages of cloud computing. Moreover, these approaches cannot execute non-TT VMs in the slack of TT VM.

Our TT LRS already allows execution of non-TT VMs in the slots where no TT VM are assigned. The next step is to include support to add new Event-Triggered (ET) VMs to the scheduling table at runtime. In Section VII.6.1, we discussed an initial idea for an algorithm inspired by the global Slot-Shifting [231] to the LRMs. To do so, the LRM must execute an acceptance test, and a guarantee routine similar to the Slot-Shifting algorithm [231]. When a new ET VM arrives, the acceptance test determines if there is enough available CPU and memory bandwidth before the VM's deadline. If the acceptance test is successful, the guarantee routine must be executed to add the VM to the scheduling table and update the CPU and memory bandwidth availability.

As part of our ongoing work, we have currently developed the modified acceptance test and guarantee routine and implemented them as part of an LRM (running on the housekeeping core). Initial experiments performed on an Intel i7 9700k (3GHz, 7 out of 8 cores with TT scheduling) show promising results. We determined the overhead for the LRM performing the acceptance test and guarantee routine for the following cases: i) 1 ET VM arrives and is rejected due to the lack of resource availability, and ii) when 1, 2, and 4 ET VMs arrive on a node, and they are all accepted and added to the scheduling table. In the worst case, we observed an overhead of 2.94 μs. Previous approaches, such as Tableau [229], take much more time for table regeneration ($> 100ms$) when just adding one new VMs to the scheduling table.

VIII.2.2 Evaluation of Global Resource Management in Time-Triggered (TT) RT-Cloud

If an LRM is unable to accept (and guarantee) a new ET VM due to unavailability of resources, the LRM requests the GRM for cloud-wide adaptation. The GRM, in turn, searches for another node with more appropriate resource availability and dispatches the VM to that node. Similarly, if a node is unable to execute an existing VM or the VM does not achieve the desired QoS due to a change in availability of resources, the LRM informs this to the GRM. The GRM, in turn, redeploys the VM to another node in the RT-Cloud. As part of the ongoing work, we are working towards a strategy to deterministically redeploy VMs in an RT-Cloud.

VIII.2.3 Energy-aware Time-Triggered (TT)-Event-Triggered (ET) Joint Scheduling in RT-Cloud Nodes

Energy consumption is not only a problem in portable devices but also in servers. Unchecked power consumption leads to a large amount of excess heat production. As a result, cooling costs also increase. Moreover, heating can lead to a reduction in the life span of the hardware. Although servers have unlimited access to electricity, power saving is required to keep down the electricity costs.

As part of our ongoing work, we are working towards making the LRM (and the TT LRS) energy aware. The main idea is that the offline scheduler produces a scheduling table with the aim to keep the frequency of the cores as low as possible while still meeting the deadlines of the safety-critical VMs. At runtime, if a VM finishes execution before its WCET, the LRM uses the newly available CPU slots to slow down the cores further. Alternatively, at runtime, the LRM can also increase the frequency of cores to free up slots and be able to accept and guarantee newly arrived ET VMs.

VIII.2.4 Memory bandwidth regulation for Safety-Critical real-time System (SCS) on Intel Xeon Processors

Preventive methods, such as Memguard [88], are often used for memory bandwidth regulation. These algorithms limit the number of memory requests that each CPU core issues. The memory accesses of a core are determined by a PMC, usually programmed to count the LLC misses. However, PMCs can be highly accurate or highly inaccurate. Their accuracy depends upon how they are accessed, the application running on the core, and the measured event [232]. Moreover, existing approaches have high overheads and do not scale well with the increase in CPU cores. In addition, they reduce the performance of the CPU. For example, suppose a core ends up utilizing the assigned number of memory accesses in the regulation period. In that case, the application running on the core is preempted, and the core is forced to idle for the remaining time of the regulation period, thus reducing CPU performance.

Intel has introduced a new programmable bandwidth controller in the 2nd Gen. Intel Xeon processors [30] as part of the MBA hardware architecture. The controller is present between each core and the shared on-chip interconnect. It allows inserting a programmable delay for the memory accesses from each core and supports up to 90% throttling and in 10% steps. Thus, this approach does not require interrupting cores to regulate the memory bandwidth. As part of our ongoing work, we are performing experiments to understand the capability of the MBA architecture to deterministically allocate memory bandwidth to cores.

VIII.2.5 Simultaneously Execution of Critical Applications in Multicore Nodes

As explained in Chapter IV, local temporal overload management (via potential deadline overrun monitoring) allowed us to execute multiple non-criticality applications concurrently with critical applications. Future work here involves developing resource

management techniques to execute applications of the same criticality levels concurrently.

There exist some work on control mechanisms that support concurrent execution of critical applications (e.g., [234]). Combining such approaches with potential deadline overrun monitoring could improve the utilization of a node and allow multiple critical as well as non-critical applications simultaneously.

VIII.2.6 Application State and Fault-Tolerance via Global Reconfiguration

The global fault-tolerance mechanism implemented for the avionics use case considers that the reconfigured applications are stateless. Therefore, the current state of the applications is completely lost on a global reconfiguration that results in the deployment of an application on a different node. Future work must consider the state of an application before a global reconfiguration and how it can be recovered on a new node without compromising the system.

VIII.2.7 Peripherals in Mixed-Critical System (MCS)

Current research strongly focuses on the use of multicores, and MPSoCs for MCS. However, very few works addressed the usage of peripherals in MCS. In MCS many functionalities depend on several sensors and external information. The applications running on the CPU cores acquire external data from the sensors using specialized interfaces rather than the memory hierarchy. These interfaces introduce new sources of interference that should be addressed by the resource management.

VIII.2.8 Blockchain-Based Distributed Global Resource Manager (DGRM) in RT-Cloud with Industrial Use Case

Future work involves integrating and testing our blockchain-based distributed global resource management for a RT-Cloud with an industrial use case.

VIII.2.9 Resource Management Communication with MQTT and OPC UA

Future work involves experimental evaluation of resource management communication via OPC UA and MQTT as well as testing the feasibility with an industrial use case.

Appendix A

Configuration File Examples

Listing A.L1 and A.L2 show examples of a Platform configuration (Library) file and a system configuration file (user-defined) for use with the resource management framework.

Listing A.L1: *Platform Configuration (Library) File Example*

```
ZC706_Zynq7000:
#Config file for Xilinx Zynq 7000 ZC706 board
  hw_desc:
    num_cores: 2
    memory_layout:
    # Default size is in MBs
     - { type: rom, start: 0x0, size: 1}
     - { type: sdram, start: 0x00100000, size: 1023 }
    devices:
     Uart: [ { id: 1, baud_rate: 115200, name: Uart } ]
     TTE:
       name: TTEthernet_1
       base_address: 0x50000000
  [...]
# default values for use if not specified in
# resource management configuration file
  LRM_desc:
  #LRM defaults for the platform
    shared_data:
      size: 32
    LRM:
      memory_area_size: 2
      flags: [...]
      console: Uart
    MON:
      memory_area_size: 2
      [...]
# XtratuM hypervisor defaults for the board
  xm_hypervisor:
```

```
    console: Uart
    features: XM_HYP_FEAT_FPGA_PART_ACCESS
    physical_memory_area: { size: 2 }
  [...]
[...]
```

Listing A.L2: *System Configuration File Example (User-Defined)*

```
nodes:
  - {name: Node1, id: 1, type:ZC706_Zynq7000}
  - {name: Node2, id: 2, type:T4240_e6500}
  - {name: Node3, id: 3, type:T4240_e6500}
  [...]
devices:
  - {name: TTESW1, id: 4, type:TTESW_8port}
  [...]
cluster:
  id:0
  nodes: [Node1,Node2,Node3]
  connections:
    - [Node1,TTESW1]
    - [Node2,TTESW1]
    - [Node2,TTESW1]
  [...]
[...]
```

Listing A.L3 and A.L4 show examples of resource management configuration file (user-defined) and application configuration file (user-defined) similar to those used in the avionics use case.

Listing A.L3: *Example of Resource Management Configuration File for Avionics Use Case*

```
Node:
  - id: 1 # This node is Zynq ZC706
    hypervisor: xtratum
    Hypervisor_LRS_schedule:
      schedule_table:
       - id: 0 # core 0
         configuration:
           - id: 0
             major_frame: 4000# MaF
             slots:
               # Application Slots(s)
               # {Slot id, Start time, Duration,
                 Partition (VM), vCPU}
               - { id: 0, start: 0, duration: 100,
                 APPpart: A1, vcpu: 0 }
```

```
      # Resource Management Slots(s)
      - { id: 1, start: 100, duration: 50,
        RMpart: CoreMON, vcpu: 0 }
      - { id: 2, start: 800, duration: 3000,
        RMpart:LRM, vcpu:0 }
  - id: 1
    major_frame:4000
    slots:
      - { id: 0, start: 0, duration: 50,
        RMpart: CoreMON, vcpu: 0 }
      - { id: 1, start: 50, duration: 100,
        APPpart: A1, vcpu: 0 }
      - { id: 2, start: 800, duration: 3000,
        RMpart:LRM, vcpu: 0 }
   - id: 2
     major_frame: 4000
     slots:
       - { id: 0, start: 800, duration: 3000,
         RMpart: LRM, vcpu: 0 }
   - id: 3
     major_frame: 4000
     slots:
       - { id: 0, start: 0, duration: 50,
         RMpart: CoreMON, vcpu: 0 }
       - { id: 1, start: 50, duration: 100,
         APPpart: A1, vcpu: 0 }
       - { id: 2, start: 800, duration: 3000,
         RMpart: LRM, vcpu: 0 }
   - id: 1
   configuration:
     - id: 0
     major_frame: 4000
     slots:
       - { id: 1, start: 0, duration: 50,
         RMpart: CoreMON, vcpu: 0 }
       - { id: 2, start: 50, duration: 200,
         RMpart: GRM, vcpu: 0 }
       - { id: 3, start: 300, duration: 200,
         APPpart: A2}
       - { id: 4, start: 800, duration: 3000,
         RMpart: LRM, vcpu: 0 }
  - id: 1
    major_frame: 4000
    slots:
```

```
            - { id: 1, start: 0, duration:   50,
              RMpart: CoreMON, vcpu: 0 }
            - { id: 2, start: 50, duration: 200,
              RMpart: GRM, vcpu: 0 }
            - { id: 3, start: 800, duration: 3000,
              RMpart: LRM, vcpu: 0 }
          - id: 2
            major_frame: 4000
            slots:
              - { id: 1, start: 0, duration:   50,
                RMpart: CoreMON, vcpu: 0 }
              - { id: 2, start: 50, duration:  100,
                APPpart: A1, vcpu: 0 }
              - { id: 3, start: 150, duration:
                200, RMpart: GRM, vcpu: 0 }
              - { id: 4, start: 800, duration: 3000,
                RMpart: LRM, vcpu: 0 }
          - id: 3
            major_frame: 4000
            slots:
            - { id: 1, start: 0, duration: 50,
              RMpart: CoreMON, vcpu: 0 }
            - { id: 2, start: 50, duration: 100,
              APPpart: A2, vcpu: 0 }
            - { id: 3, start: 150, duration: 200,
              RMpart: GRM, vcpu: 0 }
            - { id: 4, start: 800, duration: 3000,
              RMpart: LRM, vcpu: 0 }
reconfiguration_table:
# Local reconfiguration table
  - [ -1 , -1 ]
  - [  2 , -1 ]
  - [ -1 , -1 ]
  - [ -1 , -1 ]
global_reconfiguration_table:
# global reconfiguration table
  - { msg: [2,9,13,15], new: [-1,-1],
    current_configuration: [-1,-1] }
  - { msg: [3,4,5,10,11,14], new: [-1,6],
    current_configuration: [-1,1] }
  - { msg: [6,7,8,12], new: [-1,6],
    current_configuration: [1,1] }
  - { msg: [16,17,18,19,20,21,22,23,24,25,26,27,28,29],
    new: [-1,-1], current_configuration: [-1,-1] }
```

```
RM_description:
#General resource management description
  shared_mem:
    start: 0x10000000
    size: 32
  security_level:#options:0,1,2
    update_channel: 2# Level 2 security
    order_channel: 2
    Membership_channel: -1# channel not required
  security_algorithm: 1# ChaCha20-Poly1305
  GRM:
  #GRM description
    memory_area_size: 2
    part_desc_flags: [ boot, fp, system ]
    part_desc_console: Uart
    TTE_ports:
      order_ports:
      #1st T4240
        - {node: 1, tte_ap_id: 1, ports: [8,10,12,14] }
        #TTE ES port IDs in Master LRM order
      #2nd T4240
        - {node: 2, tte_ap_id: 1, ports: [9,11,13,15] }
      update_ports:
      #1st T4240
        - {node: 1, tte_ap_id: 2, ports: [32,40,48,56] }
        #TTE ES port IDs in Master LRM order
      #2nd T4240
        - {node: 2, tte_ap_id: 2, ports: [64,72,80,88] }
  LRM:
  #LRM description
    memory_area_size: 2
    part_desc_flags: [ boot, fp, system ]
    part_desc_console: Uart
    max_no_messages: 32
    master_order:[0,1,2,3]
    MON: [CoreMON, HWMON, DeadlineMON]
    [...]
  CoreMON:
  # Core Failure Monitor
    memory_area_size: 2
    part_desc_flags: [ boot, fp, system ]
    part_desc_console: Uart
APPpart_description:
# Scheduling specific description of
```

```
    # application partitions
      - id: 0 # XtratuM partition ID
        name: A1 # XtratuM partition name
        APPid: 1 # applicaiton ID
        APPname: App_crit1  # optional
        DeadlineMON: 0 # not required
        configurations:
          - id: 1
            slots:
              - { id: 1, tasks: [ Task0 ] }
          - id: 2
            slots:
              - { id: 2, tasks: [ Task0 ] }
          - id: 3
            slots:
              - { id: 1, tasks: [ Task0 ] }
      - id: 1
        name: A2
        APPid: 2
        APPname: App_crit2 # optional
        DeadlineMON: 1 # required
        configurations:
          - id: 3
          slots:
            - { id: 0, tasks: [ Task0, Task1, Task2 ],
              DeadlineMON: [ 30, 50, 80 ] }
             # Observation points for potential
             # deadline overrun monitoring
Node: 2
[...]
```

Listing A.L4: *Application Configuration File for use with Listing A.L3*

```
apps:
#All apps in the system
  # First App
  - &myapp1
    id: 1
    name: App_crit1
    LRS: LRS_critical
    # Use critical partition LRS
    tasks:
      - { name: Task0, func:  user_partition_launch,
        state: task0_state}
  - &myapp2
```

```
  # Second App for this node
    name: App_crit2
    id: 2
    LRS: LRS_critical
    tasks:
       - { name: Task0, func:  user_partition_launch,
         state: task0_state}
       - { name: Task1, func:  user_partition_launch,
         state: task1_state}
       - { name: Task2, func:  user_partition_launch,
         state: task2_state}
[...]
Node:
  -id: 1 # This node is Zynq ZC706
   hypervisor: xtratum
    App_description:
    # Description of each application
      - id: 1 # App 1
        # node specific description
        flags: [ system, boot, fp ]
        console: Uart
        physical_memory_areas:
        - { start: 0x4000000, size: 2 }
        app: *myapp1
        tasks: # node specific task parameters
          - { name: Task0, WCETiso: 10}
      - id: 2
        flags: [ system, boot, fp ]
        console: Uart
        physical_memory_areas:
          - { start: 0x4200000, size: 2 }
        app: *myapp2
        tasks:
          - { name: Task0, WCETiso: 10}
          - { name: Task1, WCETiso: 10}
          - { name: Task2, WCETiso: 10}
Node: 2
[...]
```

Appendix B

Examples of Useful PMU events

Appendix B provides examples of some useful PMU events with respect to resource management on the three hardware platforms: NXP T4240, Xilinx Zynq 7000, and Intel Xeon Gold 5218.

1. Table B.T1 provides the examples of PMU events for the e6500 cores [176] of NXP T4240 [12]
2. Table B.T2 and Table B.T3 present the examples of Cortex A9 core PMU events [177] and the AXI shared interconnect uncore PMU (APM IP block [174]) events in the Xilinx Zynq 7000 [13]
3. Table B.T4 and Table B.T5 supply the examples of Intel Xeon Gold 5218 core and uncore PMU events [179]

Table B.T1: *Core PMU event examples of e6500 cores in T4240*

Event number	Event mnemonic	Description
Ref:1	Processor cycles	Counts every processor cycle
Ref:2	Instructions completed	For counting every completed instruction. The value can increment by either 0, 1 or 2 per processor cycle
Com:222	Data L1 cache misses	L1-D cache load and store misses. Similar event is available for L1-I
Com:41	Data L1 cache reloads	L1-D cache reloads for any reason. Similar event is available for L1-I
Com:466	L2 miss per thread	L2 cache load and store misses by a specific core or a thread
Com:465	L2 access per thread	L2 cache access by a specific core or a thread
Com:185	Memory Barriers completed	For counting the completed memory barier instructions such as msync, mbar, and miso.
Com:506	BIU master request	For counting the number of time a master accesses the corenet shared interconnect

Table B.T2: *Core PMU events of Cortex-A9 (Zynq 7000 MPSoC)*

Event number	Event mnemonic	Description
0x01	L1I_CACHE_REFILL	L1-I cache refill
0x14	L1I_CACHE	L1-I cache access
0x03	L1D_CACHE_REFILL	L1-D cache refill
0x04	L1D_CACHE	L1-D cache access
0x08	INST_RETIRED	Instruction architecturally executed.
0x11	CPU_CYCLES	CPU cycle
0x13	MEM_ACCESS	Data memory access
0x16	L2D_CACHE	L2(-D) cache access
0x17	L2D_CACHE_REFILL	L2(-D) cache refill (miss in L2)
0x19	BUS_ACCESS	Bus access
0x1D	BUS_CYCLES	Bus cycles
0x1A	MEMORY_ERROR	Local memory error

Table B.T3: *APM (uncore PMU) events (Zynq 7000 MPSoC)*

Event name	Description
Write Transaction Count	Total number of write requests from/by an agent
Read Transaction Count	Total number of read requests from/by an agent
Write Latency	Time from issuing the write address to the first/last write to the slave. At time of instantiating the APM, it is possible to configure the latency to calculate till either first of last write. Similar event is available for read latency
Write Count Bytes	Total number of bytes written to/by an agent
Read Count Bytes	Total number of bytes read by/from an agent

Table B.T4: *Core PMU events of Intel Xeon scalable processors (2nd Gen.)*

Event number	Event mnemonic	Description
Events of Fixed PMC		
EventSel=00H UMask=01H	INST_RETIRED.ANY	Number of instructions completed (retired from execution).
EventSel=00H UMask=02H	CPU_CLK_UNHALTED. THREAD	Number of core clock cycles while a thread is not in a halt state
EventSel=00H UMask=02H Anythread=1	CPU_CLK_UNHALTED.ANY	Number of core clock cycles while any thread of a CPU is not in a halt state
EventSel=00H UMask=03H	CPU_CLK_UNHALTED. REF_TSC	Number of reference cycles when the core is not in a halt state
Events of Programmable PMC		
EventSel=A3H UMask=02H CMask=2	CYCLE_ACTIVITY.CYCLES_ L3_MISS	Cycles while L3 cache miss demand load is outstanding. Similar events are available for L1-I, L1-D and L2 caches as well
EventSel=24H UMask=24H	L2_RQSTS.CODE_ RD_MISS	All instructions that have L2 cache misses
EventSel=24H UMask=27H	L2_RQSTS.DEMAND_ DATA_RD_MISS	All data requests that have L2 cache misses
EventSel=24H UMask=3FH	L2_RQSTS.MISS	All L2 Cache misses
EventSel=24H UMask=FFH	L2_RQSTS.REFERENCES	ALL L2 Requests. Similar events for L2 instruction and data requests are available
EventSel=B0H UMask=10H	OFFCORE_REQUESTS.L3_ MISS_ DEMAND_RD	Data request with L3 cache misses
EventSel=2EH UMask=41H	LONGEST_LAT_ CACHE.MISS	All cachable requests that have a L3 miss
EventSel=2EH UMask=4FH	LONGEST_LAT_ CACHE.REFERENCE	All requests cachable in L3
EventSel=D3H UMask=01H	MEM_LOAD_L3_MISS_ RETIRED.LOCAL_DRAM	Completed load instructions where data sources had L3 miss (and thus, needed data from DRAM).

Table B.T5: *Uncore PMU events of Intel Xeon scalable processors ($2^n d$ Gen.)*

Event number	Event mnemonic	Description
EventSel=00H UMask=00H	UNC_M_CLOCKTICKS	Counts the Clock ticks of the IMC
EventSel=00H UMask=02H	UNC_M_ACT_COUNT.WR	Counts DRAM page ACT command due to a write request. Similar events are available for PRE and CAS
EventSel=02H UMask=01H	UNC_M_PRE_COUNT. PAGE_MISS	Counts the number of page misses
EventSel=43H UMask=00H	UNC_M_POWER_SELF_ REFRESH	Counts cycles for which IMC is performing a DRAM refresh
EventSel=80H UMask=00H	UNC_M_RPQ_OCCUPANCY	Counts read pending queue occupancy of the IMC. Similar event is present for write pending queue.
EventSel=01H UMask=00H	UNC_UPI_CLOCKTICKS	Counts Clock ticks of the UPI interconnect

Appendix C

Pseudo-Code for MONs, LRSs, LRM, and GRM

C.1 MONitors (MONs)

C.1.1 Hardware MONs

Listing C.L1 shows a pseudo-code of a hardware MON (Section IV.4.1).

Listing C.L1: *Pseudo-code for Hardware MON*

```
int HW_MON_init(int TotalEvents,
        int EventList[MAX_SUPPORTED_PMU_EVENTS]){
//LRM uses this function to initialize the PMU
//TotalEvents = Total number of events to configure
//EventList = List of Event Numbers
        if(TotalEvents>MAX_SUPPORTED_PMU_EVENTS)
                return CONFIG_ERROR;
        if(enable_PMU()!=SUCCESS)
                return PMU_ERROR;
        for(int e=0; e < TotalEvents; e++){
                configure_event_PMC(e,EventList[e]);
        }
        return SUCCESS;
}

int HW_MON_reset(){
//This function resets the PMCs of the PMU
        if(reset_PMU()!=SUCCESS)
                return PMU_ERROR;
        else return SUCCESS;
}

void HW_MON_read(int TotalEvents,
        lrm_mon_status_t *status){
//This function is used to read count of the
//initialized events stored in the PMC
```

```
        for(int e=0; e < TotalEvents; e++){
                status->pmc[1]=read_PMC(e);
        }
}
```

Listing C.L2 shows a code section of the header file for T4240 hardware MON.

Listing C.L2: *Example of PMU events in Hardware MON Header file*

```
//#define Common_Event_Names #EventNumbers
#define CLOCK_CYCLES 1
//Original name: Processor cycles
#define INSTRUCTIONS_COMPLETED 2
//Original name: Instructions completed
#define LLC_CACHE_MISSES 466
//original name: L2 Cache miss per thread
```

The hardware MON provides two routines for gathering resource consumption of the application as shown in Listing C.L3.

Listing C.L3: *Pseudo-code for XtratuM Hardware MON*

```
void HW_MON_slot_start(PartitionID,SlotID,TaskID){
//This sub routine runs at start of every partition slot
        HW_MON_read(&slot_start_values);
}

void HW_MON_slot_end(PartitionID,SlotID,TaskID){
//This sub routine runs at end of every partition slot
        HW_MON_read(&end);
        start = &slot_start_values;
        for (i = 0; i < MAX_SUPPORTED_PMU_EVENTS; i++){
                update_stats(&data->slots[slotID].pmc[i],
                end.pmc[i] - start->pmc[i]);
        }
        data->slots[slotID].updated = 1;
        //To indicate to the LRM that the data is updated
}
```

Listing C.L4 and Listing C.L5 show pseudo-code for hardware MON hooks in Linux/KVM on x86 platforms (Xeon) and ARMv8a.

Listing C.L4: *Pseudo-code for Linux/KVM Hardware MON on Intel Xeon*

```
static int init HW_MON_Tracer_init(void){
//This function must be used to register HW MON hooks
//Register the HW MON hook to occur on when
//the task stars to execute
        if(tracepoint_probe_register(sched_process_exec,
```

```
                HW_MON_start_handler, NULL))
                return TRACEPOINT_ERROR;
        else
                return SUCCESS;
//Similarly, we need to register other HW MON hooks
//for handling further scheduling events, such as
//task completion and context switch.
}
void HW_MON_start_handler(void *data, struct task_struct *p,
        pid_t old_pid){
//This function is called the first time a task is scheduled.
// It is used to maintain the PMC values for each task
}
```

Listing C.L5: *Pseudo-code for Linux/KVM Hardware MON on ARMv8a*

```
static int init HW_MON_Tracer_init(void){
//This function must be used to initialize
// and register HW MON hooks
        if(stp_tracepoint_init())
                return TRACEPOINT_ERROR;
//Register the HW MON hook to occur on when
//the task stars to execute
        if(STP_TRACE_REGISTER(sched_process_exec,
                HW_MON_start_handler));
                return TRACEPOINT_ERROR;
        else
                return SUCCESS;
//Similarly, for other hooks.
}
```

C.1.2 Core Failure MON

The pseudo-code in Listing C.L19 shows the essential parts of the core failure MON (Section IV.4.3).

Listing C.L6: *Pseudo-code for Core Failure MON*

```
void MON_core_failure_init(){
//LRM uses this function to initialize the
//core failure MON
        int *alive = data->alive_buffer;
        //alive points to the data structure shared between
        //the MON instances an the LRM.
        for(int i=0; i < NODE_MAX_CORES; i++){
                alive[i] = 0;
```

```
        }
}
void MON_core_failure_instance_run(){
//Core failure MON instance
        int CoreID = get_CPUcore_id();
        //CoreID contains the ID of the core
        //where this MON instance runs
        int *alive = data->alive_buffer;
        //alive points to the shared data structure
        alive[CoreID] = 1- alive[CoreID];
}
```

C.1.3 Deadline Overrun MON

The pseudo-code in Listing C.L7 shows the essential parts of the potential deadline overrun MON (Section IV.4.4) using the first method (observation points between two tasks). The observation points can be specified via the system designers using the resource management configuration file as shown in Listing C.L8.

Listing C.L7: *Pseudo-code for Deadline Overrun MON*

```
void MON_DL_overrun_instance(){
        Elapsed = HW_MON_vm_elapsed_cycles(); //ET(x)
        Limit = vm->InternalDeadline[TaskID]; //D(x)
        if(TaskID == vm->TotalTasks-1){
                isolation_mode=0;
        //if the current task that finished was
        //the last task of the application, then
        //signal LRM about critical app completion
                return 0;
        }
        if(isolation_mode){
        //if the a potential deadline overrun was
        //already detected in this hyperperiod/MaF
                return 0;
        }
        if(Elapsed>Limit){
                isolation_mode=1;
        //Signal LRM about potential deadline overrun
        //in the critical application
                return 1;
        }
}
```

Listing C.L8: *Observation Points for Deadline Overrun MON in resource management configuration file*

```
Node:
  - id 1
    APPpart_description:
      # Scheduling specific description of
      # application partitions
      - AppID: 1 #Application ID
        APPname: App_crit2 # optional
        DeadlineMON: 1 # required (1) or Not (0)
        configurations:
          - id: 1
            slots:
              - { id: 0, tasks: [ Task0, Task1, Task2 ],
                DeadlineMON: [ 30, 50, 80 ] }
            # Observation points for potential
            # deadline overrun MON in milliseconds
[...]
```

C.1.4 Health MONs (XtratuM)

Listing C.L9 shows application-level MONs (Section IV.4.5) that can use the XtratuM HM API to write application status or errors (not handled by XtratuM) to the HM log. Listing C.L10 shows the resource management configuration file to accept as input (optional) an HM event and action mapping from the system designer to generate the XtratuM configuration.

Listing C.L9: *Pseudo-code for Health MON*

```
void MON_HM_instance(){
//This MON is called in an application upon
//detection of internal errors. The MON determines
//the cause of the error and selects the
//appropriate error event category
        event=XTRATUM_EVENT_TYPE;
//Next, it writes this event to the XtratuM
//Health Monitoring log using the API provided
//by XtratuM
        if(XM_hm_raise_event(event)!=XM_OK)
                return INVALID_EVENT;
        else
                return SUCCESS;
}
```

Listing C.L10: *Health Monitor (HM) Event-Action Mapping for XtratuM in Resource Management Configuration File*

```
Node1:
  LRM_desc:
  #LRM description
    LRM:
      MON: [CoreMON, HealthMON] #attached MON modules
      [...]
    HealthMON_desc:
      # - {Error_Type,Action Type,Event Log,Partition ID}
      # if partition id >= 0, than HealthMON is
      # only for that specific partition
      - [-1, XM_HM_EV_<TYPE1>,
        XM_HM_AC_HYPERVISOR_<ACTION_TYPE1>,TRUE ]
      - [ 0, XM_HM_EV_<TYPE2>,
        XM_HM_AC_HYPERVISOR_<ACTION_TYPE2>,FALSE]
      - [ 0, XM_HM_EV_<TYPE3>,
        XM_HM_AC_HYPERVISOR_<ACTION_TYPE3>,TRUE ]
      [...]
    [...]
  [...]
[...]
```

C.2 Local Resource Schedulers (LRSs)

C.2.1 LRS for Critical Partitions (XtratuM)

The pseudo-code in Listing C.L11 shows the essential parts of the LRS for Critical Partitions (Section IV.7.1).

Listing C.L11: *Pseudo-code for Critical Partition LRS*

```
void LRS_Critical_Partition(){
        //Initialize LRM and HW MON
        TaskID=0;
        LRM_slot_start(PartitionID,SlotID,TaskID);
        HW_MON_slot_start(PartitionID,SlotID,TaskID);
        execute_tasks(task_list[CurrentConfig][TaskID])
        //From second to Last tasks
        for(TaskID=1;TaskID<total_tasks;TaskID++){
                statistics[TaskID-1] =
                  HW_MON_instance(PartitionID,SlotID,TaskID-1);
                Deadline_Overrun_MON_instance(PartitionID,
                  SlotID,TaskID-1);
                LRM_instance(PartitionID,SlotID,TaskID-1);
```

```
                    execute_task(task_list[CurrentConfig][TaskID]);
            }
            //After Last Task
            LRM_slot_end(PartitionID,SlotID,TaskID);
            HW_MON_slot_end(PartitionID,SlotID,TaskID);
}
```

A system designer can define XtratuM plans easily via the resource management configuration file as shown in Listing C.L12.

Listing C.L12: *Time Slot Allocation to Partitions via Resource Management Configuration File*

```
Node:
  - id: 1
    hypervisor: xtratum
    Hypervisor_LRS_schedule:
# if optional paramters are absent, framework uses default
# values for the platform from the platform config library
      schedule_table: #corresponds to XtratuM processor table
        - id: 0# Core 0
          frequency: 400# optional
          configuration:
            - id: 0# corresponds to XtratuM plan 0
              major_frame: 4000# MaF
              slots:
                # Application partition
                - { id: 0, start: 0, duration: 100,
                  APPpart:APP1, vcpu:0}
                # Resource Management Partition
                - { id: 1, start: 100, duration: 50,
                  RMpart:CoreMON, vcpu:0}
                - { id: 2, start: 900, duration: 1000,
                  RMpart:LRM, vcpu:0}
            - id: 1# corresponds to XtratuM plan 1
              major_frame: 4000# MaF
              slots:
              - { id: 0, start: 0, duration: 50,
                RMpart: CoreMoN, vcpu: 0 }
              - { id: 1, start: 50, duration: 500,
                Apppart: APP1, vcpu: 0 }
              - { id: 2, start: 900, duration: 1000,
                RMpart: LRM, vcpu: 0 }
        -id: 1 # Core 1
        [...]
[...]
```

C.2.2 LRS for Online Reconfiguration (XtratuM)

The pseudo-code in Listing C.L13 shows the essential parts of the LRS for Online Reconfiguration (Section IV.7.2).

Listing C.L13: *Pseudo-code for Reconfiguration LRS (XtratuM)*

```
int NODE_LRS_Switch_Config(NewConfigID, Type){
  XM_get_plan_status(&status);
  //XtratuM plan in current MaF
  CurrentConfigID = status->current;
  //XtratuM plan in next MaF
  NextConfigID = status->next;
  //If LRM asks to change to the plan XtratuM is already
  //executing, then something is amiss
  if(NewConfigID==CurrentConfigID)
        return PLAN_SWITCH_ERROR;
  //If LRM asks to change to a plan XtratuM is already
  //scheduled for the next MaF and type is not immediate
  if((NewConfigID==NextConfigID)&&(Type!="Imm"))
    return PLAN_SWITCH_ERROR;
  switch(Type){
    case "Imm"://Immediate configuration change
      //Switch XtratuM mode (plan)
      if(XM_switch_imm_sched_plan(NewConfigID,
        &CurrentConfigID)!=XM_OK){
          return PLAN_SWITCH_ERROR;
      }
    case "Def"://Deferred configuration change
      //Switch XtratuM mode (plan)
      if(XM_switch_sched_plan(NewConfigID,
        &CurrentConfigID)!=XM_OK){
              return PLAN_SWITCH_ERROR;
      }
  }
  return SUCCESS;
}
```

C.2.3 LRS for Intel CAT and MBA

The pseudo-code in Listing C.L11 shows the essential parts of the LRS for Intel CAT and MBA (Section IV.7.3).

Listing C.L14: *Pseudo-code for CAT and MBA LRS*

```
int CAT_MBA_LRS_init(){ //Initializes MBA and CAT
}
```

```
int CAT_MBA_LRS_reset(){//Resets MBA and CAT
}
int CAT_MBA_LRS_set(CLOS[MAX_CLOS],Type){
//CLOS[MAX_CLOS] contains bitmasks for CAT or MBA delay
  switch(Type){
    case "CAT":
      for(c=0;c<MAX_CAT_CLOS;c++){
        set_CAT_CLOS_mask(c,CLOS_mask[c]);
      }
    case "MBA":
      for(c=0;c<MAX_MBA_CLOS;c++){
        set_MBA_CLOS_delay(c,CLOS_mask[c]);
      }
  }
}
int CAT_MBA_LRS_core(cores[MAX_CORE]){
//cores[MAX_CORE] contains mapping of cores to CLOS
  for(c=0;c<MAX_CAT_CLOS;c++){
    if(cores[MAX_CORE]!=-1)
      set_CLOS_to_cores(c,cores[c]);
  }
}
```

C.2.4 TT LRS for Linux/KVM

Listing C.L15 shows the definition of LRS_TT scheduling class (Section IV.7.4), which implements the TT_LRS module.

Listing C.L15: *Pseudo-code for Linux/KVM TT LRS Scheduling Class*

```
DEFINE_SCHED_CLASS(TT_LRS) = {
        .enqueue_task = enqueue_task_TT_LRS,
        .dequeue_task = dequeue_task_TT_LRS,
        .pick_next_task = pick_next_task_TT_LRS,
        #ifdef CONFIG_SMP
        .balance = balance_TT_LRS,
        .select_task_rq = select_task_rq_TT_LRS,
        .set_cpus_allowed = set_cpus_allowed_common,
        #endif
        .task_tick = task_tick_TT_LRS,
        .switched_to = switched_to_TT_LRS
};
```

Listing C.L16 shows the essential parts of the task_tick_TT() function.

Listing C.L16: *Pseudo-code for Linux/KVM TT LRS Task Tick Function*

```
static void task_tick_TT_LRS(struct rq *rq,
  struct task_struct *p, int queued){
//Linux executes this function every task tick
//length (1 ms) on each cpu core
  if (!test_bit(TT_LRS_STARTED, &TT_LRS.flags)) {
    //if the TT LRS was not started by the LRM
    if (p->TT_LRS_id != 0)
      resched_curr(rq);
      rq->TT_LRS.tick_countdown = 1;
  }
  else {
    //if the TT LRS was started by the LRM
    if (--rq->TT_LRS.tick_countdown){
    //At start of a slot, tick_countdown
    //is set TT_LRS_TICK_PERIOD
      return;
    }
    rq->TT_LRS.tick_countdown = TT_LRS_TICK_PERIOD;
    //TT_LRS_TICK_PERIOD = Slot length (in ms) divide
    //by Linux tick length (in ms)
    cpu = cpu_of(rq); //get run queue of current core
    //LRM sets sched_next_id[cpu] variable according to
    //the task to be scheduled on this cpu core in the
    //coming slot according to the offline table
    new_tsk_id = TT_LRS.sched_next_id[cpu];
    //sched_prev_id[cpu] contains the previously scheduled
    //task on this cpu core
    prev_tsk_id = TT_LRS.sched_prev_id[_idcpu];
    if (new_tsk_id != prev_tsk_id) {
    //if the previous and next tasks are not the same
      if (new_tsk_id) {
        rq->TT_LRS.resched_triggered = true;
      }
      //LRM will set sched_next_id[cpu] to 0 if there
      //is no new task in the table for this cpu in
      //the upcoming slot
      resched_curr(rq); //mark CPU core for rescheduling
    }
  }
}
```

Listing C.L17 shows the essential parts of the pick_next_task_TT() function.

Listing C.L17: *Pseudo-code for Linux/KVM TT LRS Pick Next Task Function*

```
static struct task_struct
  *pick_next_task_TT_LRS(struct rq *rq){
//function to pick next task for scheduling on this core
  struct task_struct *next = NULL;
  int cpu = cpu_of(rq); //get current core id
  int sched_task_id = 0;
  if (test_bit(TT_LRS_STARTED, &TT_LRS.flags)) {
  //if the LRM has started the TT LRS, then the
  //LRM sets sched_next_id[cpu] to the next task ID
  //to be scheduled on this cpu core in the upcoming
  //slot according to the offline scheduling table
    sched_task_id = TT_LRS.sched_next_id[cpu];
  //LRM will set sched_next_id[cpu] to 0 if there
  //is no new task in the table for this cpu in
  //the upcoming slot
    if (sched_task_id) {
    //TT_LRS_tasks is configured by LRM during the
    //task initialization phase. It contains the pointer
    //to the task structures
      next = TT_LRS_tasks[sched_task_id];
      if(next->cpu != cpu) {
      //if the task resides on a different core's run
      //queue core, then remove it from that run queue,
      //and add it to the current core's run queue
        src_rq = cpu_rq(next->cpu);
        double_lock_balance(rq, src_rq);
        deactivate_task(src_rq, next, 0);
        set_task_cpu(next, cpu);
        activate_task(rq, next, 0);
        double_unlock_balance(rq, src_rq);
      }
    }
  }
  return next;
}
```

Listing C.L18 shows the essential parts of the task_tick_TT() function.

Listing C.L18: *Pseudo-code for Linux/KVM TT LRS Set Schedule Function*

```
void TT_LRS_set_schedule(
  const unsigned int *sched_slot_task_id){
//this function is called by the LRM to set the tasks
//to be executed in the upcoming slot by the LRS
//sched_slot variable contains task IDs per cpu core
  unsigned int *sched_new_id = TT_LRS.sched_new;
```

```
  for_each_possible_cpu(c) {
  //for all cpu cores under TT LRS
    sched_new_id[c] = sched_slot_task_id[c];
  }
  tmp = TT_LRS.sched_new_id;
  TT_LRS.sched_new_id = TT_LRS.sched_prev_id;
  TT_LRS.sched_prev_id = TT_LRS.sched_next_id;
  TT_LRS.sched_next_id = tmp;

}
```

C.3 Local Resource Manager (LRM)

A system designer must the order of selection of master LRM for core failure management (Section IV.9.1) in the resource management configuration file as shown in Listing C.L19.

Listing C.L19: *LRM Master Order in Resource Management Configuration file*

```
Node:
  - id 1
    RM_description:
      shared_data:
        start_address: 0x10000000
        size: 32 # optional
      physical_memory_addr: 0x12000000
  # if optional paramters are absent, framework uses default
  # for the platform from the platform configuration library
        [...]
      LRM:
        memory_area_size: 2 # optional
        master_order:[0,2,1,3] #optional master order Core ID
        # Node has 4 cores on node. Default Order is in
        # ascending Core ID: [0,1,2,3]
        MON: [CoreMON, HWMON, DeadlineMON]
        [...]
      CoreMON:
        memory_area_size: 2 # optional
        [...]
[...]
```

Listing C.L20 shows the modified parts of the deadline overrun MON to support Solution 1 for improving under-utilization due to deadline overrun management (Section IV.9.2).

Listing C.L20: *Pseudo-code for Deadline Overrun MON to support QoS management of Solution 1*

```
void MON_DL_overrun_instance(){
        Elapsed = HW_MON_vm_elapsed_cycles(); //ET(x)
        Limit = vm->InternalDeadline[TaskID]; //D(x)
        if(TaskID == vm->TotalTasks-1){
                isolation_mode=0;
                throttle_qos_mode=0;
                //if the current task that finished was
                //the last task of the application, then
                //signal LRM about critical app completion
                return 0;
        }
        if(isolation_mode){
                //if the a potential deadline overrun was
                //already detected twice
                return 0;
        }
        if(Elapsed>Limit){
                if(throttle_qos_mode){
                        isolation_mode=1;
                        //if the a potential deadline overrun
                        //was already detected
                        return 1;
                }
        else
                        throttle_qos_mode=1;
                //Signal LRM about potential deadline overrun
                //in the critical application
                return 1;
        }
}
```

C.4 Global Resource Manager (GRM)

System designers can set the security to Level 1/2 or altogether disable it (Level 0) via the resource management configuration file as shown in Listing C.L21.

Listing C.L21: *Security Level and Algorithm Selection in Resource Management Configuration file*

```
RM_description:
#common resource management description for the whole system
  [...]
  # Security level per channel
  # Level 0: no usage security library
```

```
  # Level 1-2: use security library with desired level
  security_level:
    update_channel : 2
    order_channel : 2
    membership_channel : 1 # -1 if channel is not required
  # Security algorithm selection
  # Security algorithm 1: ChaCha20-Poly1305
  # Security algorithm 2: CLEFIA-OCB
  # Security algorithm 3: User-define algorithm module
  security_algorithm: 1 # Explained in later sections
  [...]
[...]
```

Appendix D

Avionics Use Case-Specific Pseudo-Code of GRM and LRM

Listing D.L1 shows a pseudo-code for the functions that send orders and receive updates in the GRM. Similar functions exist in LRMs to receive orders and send updates.

Listing D.L1: *Resource Management Communication from GRM*

```
//Function to send Order to a LRM
send_order ( struct order_message msg,
xm_s32_t OrdersPort, uint32_t LRM_partition){
        //package to two fields into one message
  encode_uintx (&order_msg, msg.config, msg.immediate);
  if(RM_config.security_lvl_order_ch >= 0){
  //if secure RM communication channel level > 0
  //Secure the message with appropriate security level
  //and send it via the required channel
  //XM_PARTITION_SELF = Current (source) partition ID
  //LRM_partition = destination LRM partition ID
  //OrderPort = XtratuM port descriptor
    ret_cod=secure_and_send(order_msg,sizeof(order_msg),
        XM_PARTITION_SELF, LRM_partition, OrderPort,
        SAMPLING,...,RM_config.security_lvl_order_ch);
  }
  else{
    //non-secure RM communication selected by configuration
    ret_code = send_nonsecure_message(order_msg,
      sizeof(order_msg),OrderPort,SAMPLING);
  }
  if( ret_code <0)
    return RM_COMM_ERROR;//should not happen
  else return RM_COMM_SUCCESS;
}
//Function to receive update from a LRM
receive_update (struct update_message *new_msg,
```

```
xm_s32_t UpdatesPort, uint32_t lrm_part ){
  if(RM_config.security_lvl_update_ch >= 0){
    //if secure RM communication channel level > 0
    //Receive the message via the required channel
    //and authenticate and decrypt the message
    //according to the security level
    //XM_PARTITION_SELF = Current(destination) partition ID
    //LRM_partition = source LRM partition ID
    //UpdatePort = XtratuM port descriptor
    ret_code = receive_and_authenticate(update_msg,
    sizeof(update_msg),LRM_partition,XM_PARTITION_SELF,
    UpdatePort,QUEUING,...,RM_config.security_lvl_update_ch);
  }
  else{
    //non-secure RM communication selected by configuration
    ret_code=receive_nonsecure_message(UpdatePort,update_msg,
      sizeof(update_msg),QUEUING);
  }
  if(ret_code == XM_NOT_AVAILABLE)
  //if XtratuM signal no new message has arrived
    return RM_COMM_NO_NEW_UPDATE;//no new Update available
  else if(ret_code<0)
    return RM_COMM_ERROR;//should not happen
  //convert to struct update_message format
  new_msg = decode_uintx(update_msg,
            sizeof(new_msg->type)
            sizeof(new_msg->current_config));
  return RM_COMM_SUCCESS ;
}
```

Listing D.L2 illustrates the pseudo-code of the static C-array for local reconfiguration graph automatically generated by the resource management framework for the example in Figure VI.F10 (Page 159).

Listing D.L2: *Pseudo-code for Local Reconfiguration Graph in LRM of Node N_1 (T4240 1)*

```
//Auto-generated code section by Resource Management
//Framework based on input from GREC for T4240 1 (node1)
//Both T4240 start in LCinit (= 1)
//Configuration of the LRM is coded as follows:
//LCxtratum=0; xtratum boot to this plan.
//We initialize the Apps here and assume that cores do not
//fail during this plan

//LCinit=1, LC0=2, LC01=3, LC02=4, LC03=5, LC012=6, LC013=7,
//LC023=8, LC1=9, LC12=10, LC13=11, LC123=12, LC2=13, LC23=14
```

```
//LC3=15, LCginit=16, LCg0=17, LCg01 =18 , LCg02=19, LCg03=20
//LCg012=21, LCg013=22, LCg023=23, LCg1=24, LCg12=25,
//LCg13=26, LCg123=27, LCg2=28, LCg23=29, LCg3=30

//Local_graph[TOTAL_CONFIGURATION][NUM_CORES]
static int static int Local_graph[30][4] = {
// IDd of the Failed core
//  0, 1, 2, 3
                  //Current Plan
  {-1,-1,-1,-1}, //0.LCxtratum - no core failures assumed
  { 2, 9,13,15}, //1.LCinit: Move to Config2 if core 0 fails
                 //1.LCinit: Move to Config9 if core 1 fails
                 //1.LCinit: Move to Config13 if core 2 fails
                 //1.LCinit: Move to Config15 if core 3 fails
  { 2, 3, 4, 5}, //3.LC0: Move to Config3 if core 0,1 fail
                 //3.LC0: Move to Config4 if core 0,2 fail
                 //3.LC0: Move to Config5 if core 0,3 fail
  { 3, 3, 6, 7}, //4.LC01
  { 4, 6, 4, 8}, //5.LC02
  { 5, 7, 8, 5}, //6.LC03
  {-1,-1,-1,-1}, //7.LC013 - no more configuration possible
  {-1,-1,-1,-1}, //8.LC023 - no more configuration possible
  { 3, 9,12,11}, //9.LC1
  {...}//similarly for LC12, LC13, LC123, LC2, LC23, LC3
  //if we are in LCginit due to global reconfiguration
  {17,24,28,30}, //16.LCginit
  {17,18,19,20}, //17.LCg0
  {18,18,21,22}, //18.LCg01
  {...}//similarly for remaining entries
  };
```

Listing D.L3 illustrates the pseudo-code of the function for global reconfiguration graph automatically generated by the resource management framework for the example in Figure VI.F12 (Page 161).

Listing D.L3: *Pseudo-code for Global Reconfiguration Graph in the GRM*

```
//Auto-generated code section by Resource Management
//Framework based on input from GREC
static int current_configuration [2]={1 ,1};
//for T4240 1 (node1) and 2 (node2)
//Both T4240 start in LCinit (= 1)
//We assume DHP does not fail. So we do not consider DHP here
//LConfiguration of each LRM is coded as follows:
//LCinit=1, LC0=2, LC01=3, LC02=4, LC03=5, LC012=6, LC013=7,
//LC023=8, LC1=9, LC12=10, LC13=11, LC123=12, LC2=13, LC23=14
```

```
//LC3=15, LCginit=16, LCg0=17, LCg01 =18 , LCg02=19, LCg03=20
//LCg012=21, LCg013=22, LCg023=23, LCg1=24, LCg12=25,
//LCg13=26, LCg123=27, LCg2=28, LCg23=29, LCg3=30
//Cases for node N1 only
receive_update(&update_msg , port[0],...);
//Has LRM of Node 1 sent an update?
//Message corresponds to the current configuration
switch(update_msg.config){
  case 2: case 9: case 13: case 15:
  // just update the configuration
    break ;
  case 3: case 4: case 5: case 10:
  case 11: case 14:
  // unique failure in N0
    if ( current_configuration [1]==1){
      //A2 must be reallocated on N2
      order_msg.config=6;
      send_order(order_msg, port[1],...);
      //Send order to LRM of Node 2 sent
    }
    break ;
  case 6: case 7: case 8: case 12:
    if(current_configuration [1]==1
    && current_configuration [0]==1){
      //double failure in N0 during one MaF
      //and both nodes were in Cinit before
      order_msg.config=6;
      send_order(order_msg, port[1],...);
      //Send order to LRM of Node 2 sent
    }
    break ;
  case 16:
  case 17: case 24: case 28: case 30:
    // A2 cannot be reallocated
  case 18: case 19: case 20: case 29:
    // A3 cannot be reallocated as well
  case 21: case 22: case 23: case 25:
  case 26: case 27:
    break ;
}
current_configuration[0]=update_msg.config;
```

Appendix E

Time-Triggered (TT) Scheduling for Cloud Nodes

E.1 Heuristic for Generation of a Time-Triggered (TT) Scheduling Table

- N = total number of Time-Triggered (TT) Virtual Machines (VMs).
- P = total number of TT CPU cores.
- S = Total number of slots. The last slot is equal to the last deadline in the system. Each slot is a multicore slot. For simplicity, we assume each VM requires only one virtual CPU allocated on a physical CPU. Thus, in a system with P TT cores, a slot can execute P VMs at a time.
- $\tau_{t,s} \in \{0, 1\}$ such that VM t has to execute in slot s, than $\tau_{t,s} = 1$: otherwise $\tau_{t,s} = 0$.
- We define a VM τ_n as a tuple $\langle r_n, C_n^s, d, \phi_n, C_n^m \rangle$, where r_n is the start slot of τ_n, C_n^s is worst-case execution time of τ_n (in slots) when running in isolation and with no memory bandwidth restrictions, d_n is the absolute deadline of the τ_n (in slots), ϕ_n is the maximum number of memory accesses τ_n is allowed to issue per regulation interval, and C_n^m is the worst-case execution time of τ_n (in slots) when restricting the memory bandwidth of the τ_n to ϕ_n accesses per regulation interval. We consider τ_n has a constant amount of memory accesses.

Assumptions:

1. We assume that the values ϕ_n and C_n^m are provided by the system designer.
2. We assume that an ongoing memory access is not preemptible. Each VM is granted at most one memory access at a time, resulting in waiting times for accesses from other VMs.

The offline scheduler uses Integer Linear Programming (ILP) to generate a scheduling table.

ILP model

To schedule the VMs as late as possible, the model is shown in equation E.1.

$$maximize \sum_{s=0}^{S} \sum_{n=0}^{N} s * C_n * d_n * \tau_{n,s} \tag{E.1}$$

Subjected to:

- VMs should not be schedule before their release times.

$$\sum_{s=0}^{r_n} \tau_{n,s} = 0; \forall n \in \{0, N\} \tag{E.2}$$

- VMs should not be schedule after their deadlines.

$$\sum_{s=d_n}^{S} \tau_{n,s} = 0; \forall n \in \{0, N\} \tag{E.3}$$

- At max, P (= total number of TT cores) VMs can be scheduled in a slot.

$$\sum_{n=0}^{N} \tau_{n,s} \leq P; \forall s \in \{0, S\} \tag{E.4}$$

- Each VM should be allocated the number of slots exactly equal to its Worst-case Execution Time (WCET).

$$\sum_{s=0}^{S} \tau_{n,s} \leq C_{\tau_n}; \forall n \in \{0, N\} \tag{E.5}$$

- The sum of memory bandwidth consumption of all VMs scheduled in a slot should be $\leq$ 100%. We consider max. bandwidth as "100" (and not "1") as ILP only allows for integer values. Using a maximum bandwidth of 1 means VMs have to be assigned fractional bandwidth which is not possible here.

$$\sum_{n=0}^{N} \tau_{n,s} * \phi_n \leq 100; \forall s \in \{0, S\} \tag{E.6}$$

If the ILP fails, scheduling all the VMs may or may not be feasible. But, if the ILP succeeds, the offline scheduler gives a scheduling table with slots consisting of BW_s^p values, M_s, and SBW_s per slot S_s.

corollary

- Each VM τ_n can have multiple memory bandwidth allocations ϕ_n^i and corresponding WCET values, C_n^i (see assumptions below). An example for VM τ_n is shown in Table E.T1.

Selection variable	Memory Bandwidth	WCET	Number of Interfering cores
x_n^1	ϕ_n^1	C_n^1	IC_n^1
x_n^2	ϕ_n^2	C_n^2	IC_n^2
...	...	...	...
$x_n^{M_n}$	$\phi_n^{M_n}$	$C_n^{M_n}$	$IC_n^{M_n}$

Table E.T1: *Memory Bandwidth and WCET pairs for VM* τ_n

- In table E.T1, x_n^m is a selection variable to select one of the possible (ϕ_n^m, C_n^m) pairs for VM τ_n. M is maximum total number of ϕ_n, C_n pairs for VM τ_n and $m \in \{0, M_n\}$.
- $\tau_{n,s}^m$ indicates VM n running in slot s with a (ϕ_n^m, C_n^m) pair from table E.T1.
- **Assumptions**:
  1. ϕ_n^m are the possible memory bandwidth values for τ_n in presence of interference from all the P CPU cores, i.e., ϕ_n^m takes into account the memory arbitration overhead assuming up to $P-1$ cores access the memory simultaneously as indicated in the last column of table E.T1.
  2. To formulate Table E.T1, following assumptions are taken
     a) VM τ_n runs on Core 0 and is assigned a memory bandwidth ϕ_n^m in pre-determined steps. Only those pairs (ϕ_n^m, C_n^m) are considered where $C_n^m < d_n - r_n$
     b) This is assuming other memory intensive VMs run on the remaining number of cores indicated in the last column of table E.T1 and each of these interfering cores consumes a maximum bandwidth of $(100 - \phi_n^m)/(IC_n^m)$
  3. We assume that a system designer provides Table E.T1 for each VM τ_n.

ILP Model:

The ILP model to schedule the VMs as late as possible and to have as less memory bandwidth consumption as possible is:

$$maximize \sum_{s=0}^{S} \sum_{n=0}^{N} \sum_{m=0}^{M_n} s * C_n * d_n * \tau_{n,s}^m \tag{E.7}$$

(This is similar to Equation E.1).

This model is Subjected to:

- VMs should not be schedule before their release times.

$$\sum_{s=0}^{r_n} \sum_{m=0}^{M_n} \tau_{n,s}^{m} = 0; \forall n \in \{0, N\} \tag{E.8}$$

- VMs should not be schedule after their deadlines.

$$\sum_{s=d_n}^{S} \sum_{m=0}^{M_n} \tau_{n,s}^{m} = 0; \forall n \in \{0, N\} \tag{E.9}$$

- At max, P (= total number of CPU cores) VMs can be scheduled in a slot

$$\sum_{n=0}^{N} \sum_{m=0}^{M_n} \tau_{n,s}^{m} \leq P; \forall s \in \{0, S\} \tag{E.10}$$

- We add a new condition for selection variable as follows:

$$\sum_{m=0}^{M_n} x_n^m = 1; \forall n \in 0, N \tag{E.11}$$

  This condition specifies that only one pair of (ϕ_n^m,C_n^m) must be selected for VM τ_n.

- We add another new condition as follows:

$$\sum_{m=0}^{M_n} \tau_n^m <= 1; \forall n \in 0, N \tag{E.12}$$

  This condition specifies that if a VM executes in a slot, it can only execute using one pair of (ϕ_n^m,C_n^m) that has been selected for VM τ_n.

- Equation E.5 changes as:

$$\sum_{s=r_n}^{d_n} \tau_{n,s}^{m} = C_{\tau_n}^{m} * x_n^m; \forall n \in \{0, N\}, \forall m \in \{0, M_n\} \tag{E.13}$$

- Equation E.6 changes as:

$$\sum_{n=0}^{N} \sum_{m=0}^{M_n} \tau_{n,s}^{m} \leq 100; \forall s \in \{0, S\}, \tag{E.14}$$

- The following new condition ensure that a (ϕ_n^m,C_n^m) pair is selected as per the number of interfering cores (i.e., the number of cores executing VMs in parallel)

  $\forall n \in \{0, N\}$, $\forall m \in \{0, M_n\}$, $\forall s \in \{0, S\}$:

$$\tau_{n,s}^m * (IC_n^m + 1) \geq (\sum_{n=0}^{N} \sum_{m=0}^{M_n} \tau_{n,s}^m) - (1 - \tau_{n,s}^m) * P; \quad \text{(E.15)}$$

$$\tau_{n,s}^m * (IC_n^m + 1) \leq (\sum_{n=0}^{N} \sum_{m=0}^{M_n} \tau_{n,s}^m) + (1 - \tau_{n,s}^m) * P \quad \text{(E.16)}$$

  Explanation: equation E.15 and E.16 together represent the following:

  if($\tau_{n,s}^m$==1) then $\tau_{n,s}^m * (IC_n^m + 1) = (\sum_{n=0}^{N} \sum_{m=0}^{M_n} \tau_{n,s}^m)$

Bibliography

[1] G. Gala, G. Fohler, D. G. Pérez, and C. Pagetti, "Resource management services," in *Distributed Real-Time Architecture for Mixed-Criticality Systems* (H. Ahmadian, R. Obermaisser, and J. Perez, eds.), ch. 9, pp. 377–402, CRC Press, 2018.

[2] R. Obermaisser, M. Abuteir, H. Ahmadian, P. Balbastre, S. Barner, M. Coppola, J. Coronel, A. Crespo, G. Fohler, G. Gala, M. Grammatikakis, A. Larrucea Ortube, T. Koller, Z. Owda, and D. Weber, "Architectural style," in *Distributed Real-Time Architecture for Mixed-Criticality Systems* (H. Ahmadian, R. Obermaisser, and J. Perez, eds.), ch. 2, pp. 7–78, CRC Press, 2018.

[3] G. Gala, J. Castillo Rivera, and G. Fohler, "Work-in-progress: Cloud computing for time-triggered safety-critical systems," in *IEEE Real-Time Systems Symposium (RTSS)*, IEEE TCRTS, 2021.

[4] G. Gala and G. Fohler, "Safe and secure global resource management for real-time and mixed-criticality systems," in *2021 IEEE International Conference on Computing (ICOCO)*, IEEE, 2021. Best paper award.

[5] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, "RT-cloud: Virtualization technologies and cloud computing for railway use-case," in *24th IEEE International Symposium On Real-Time distributed Computing (IEEE ISORC)*, IEEE, 2021.

[6] G. Gala and G. Fohler, "Distributed decision-making for safe and secure global resource management via blockchain: Work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*, pp. 28–30, IEEE, 2020.

[7] G. Fohler, G. Gala, D. G. Pérez, and C. Pagetti, "Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator," in *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), (Toulouse, France), Jan. 2018.

[8] G. Gala, T. Koller, D. G. Pérez, G. Fohler, and C. Ruland, "Timing analysis of secure communication between resource managers in dreams," in *Workshop on Security and Dependability of Critical Embedded Real-Time Systems in conjunction with RTSS*, 2016.

[9] T. Koller, G. Gala, D. G. Pérez, C. Ruland, and G. Fohler, "Dreams: Secure communication between resource management components in networked multi-core systems," in *2016 IEEE Conference on Open Systems (ICOS)*, 2016. Best paper award.

[10] G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, "DREAMS about reconfiguration and adaptation in avionics," in *ERTS 2016*, (Toulouse, France), Jan. 2016.

[11] G. Gala (eds.), G. Fohler (eds.), S. Barner, A. Diewald, R. Obermaisser, T. Koller, D. G. Pérez, B. Nikolic, M. Coppola, M. Grammatikakis, A. Crespo, J. Coronel, K. Chappuis, G. Bouwer, G. Klaes, J. Perez, and A. Larrucea, "White paper on mixed-criticality research and innovation," DREAMS project, Sept. 2017. Available: `https://cordis.europa.eu/docs/projects/cnect/0/610640/080/deliverables/001-DREAMSD932WhitepaperonmixedcriticalityresearchandinnovationR10.pdf`.

[12] *T4240 Product Brief*. Freescale Semiconductor Inc. Document Number: T4240PB, Rev 1, Available: `https://www.nxp.com/docs/en/product-brief/T4240PB.pdf`, Last accessed: March 2021.

[13] *Zynq-7000 SoC Data Sheet: Overview*. Xilinx. Available: Document Number: DS190, Rev v1.11.1 `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`, Last accessed: March 2021.

[14] X. Jean, M. Gatti, G. Berthon, and M. Fumey, *The Use of MULticore proCessORS in airborne systems (MULCORS)*. Thales Avionics and EASA. Dossier ref. CCC/12/006898 - Rev. 07.

[15] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, pp. 263–272, Citeseer, 2009.

[16] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *First International Workshop on Microkernels for Embedded Systems*, vol. 50, 2007.

[17] I. Galanis, D. Olsen, and I. Anagnostopoulos, "A multi-agent based system for run-time distributed resource management," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, 2017.

[18] L. E. Talavera, M. Endler, I. Vasconcelos, R. Vasconcelos, M. Cunha, and F. J. d. S. e. Silva, "The mobile hub concept: Enabling applications for the internet of mobile things," in *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pp. 123–128, March 2015.

[19] C. Xu, K. Wang, and M. Guo, "Intelligent resource management in blockchain-based cloud datacenters," *IEEE Cloud Computing*, vol. 4, no. 6, pp. 50–59, 2017.

[20] H. Xu, P. V. Klaine, O. Onireti, B. Cao, M. Imran, and L. Zhang, "Blockchain-enabled resource management and sharing for 6g communications," *Digital Communications and Networks*, vol. 6, no. 3, pp. 261–269, 2020.

[21] L. Rizvanovic and G. Fohler, "The matrix - a framework for real-time resource management for video streaming in networks of heterogeneous devices," in *2007 Digest of Technical Papers International Conference on Consumer Electronics*, pp. 1–2, Jan 2007.

[22] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino, "Resource management on multicore systems: The actors approach," *IEEE Micro*, vol. 31, no. 3, pp. 72–81, 2011.

[23] H. Kopetz, "The artemis cross-domain architecture for embedded systems," in *2007 Design, Automation Test in Europe Conference Exhibition*, pp. 1–2, 2007.

[24] I. Cardei, R. Jha, M. Cardei, and A. Pavan, "Hierarchical architecture for real-time adaptive resource management," in *Middleware 2000* (J. Sventek and G. Coulson, eds.), (Berlin, Heidelberg), pp. 415–434, Springer Berlin Heidelberg, 2000.

[25] J. W. Ramsey, *Integrated Modular Avionics: Less is More.* Aviation Today. Available: `https://www.aviationtoday.com/2007/02/01/integrated-modular-avionics-less-is-more/`, Last accessed: March 2021.

[26] *AMBA Overview.* ARM Developer. Available: `https://developer.arm.com/architectures/system-architectures/amba`, Last accessed: March 2021.

[27] *An Introduction to the Intel QuickPath Interconnect.* Intel. Available: Document Number: 320412-001US, `https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf`, Last accessed: March 2021.

[28] L. Bononi, N. Concer, M. Grammatikakis, M. Coppola, and R. Locatelli, "Noc topologies exploration based on mapping and simulation models," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pp. 543–546, 2007.

[29] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, *Design of cost-efficient interconnect processing units: Spidergon STNoC.* CRC press, 2020.

[30] *Second Generation Intel Xeon Scalable Processors.* Intel. Available: `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/2nd-gen-xeon-scalable-processors-brief-Feb-2020-2.pdf`, Last accessed: March 2021.

[31] *Intel Xeon Gold 5218 Processor.* Intel. Available: `https://www.intel.com/content/www/us/en/products/sku/192444/intel-xeon-gold-5218-pro`

cessor-22m-cache-2-30-ghz/specifications.html, Last accessed: March 2021.

[32] O. I. de Normalización, *ISO 26262 Road vehicles - Functional safety.* Aviation Today.

[33] RTCA, “DO-178b,” *Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[34] P. Graydon and I. Bate, “Safety assurance driven problem formulation for mixed-criticality scheduling,” *Proc. WMC, RTSS*, pp. 19–24, 2013.

[35] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[36] A. K.-L. Mok, *Fundamental design problems of distributed systems for the hard-real-time environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[37] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24. Springer Science & Business Media, 2011.

[38] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pp. 239–243, IEEE, 2007.

[39] S. Baruah and A. Burns, “Implementing mixed criticality systems in ada,” in *International Conference on Reliable Software Technologies*, pp. 174–188, Springer, 2011.

[40] A. Burns and S. Baruah, “Timing faults and mixed criticality systems,” in *Dependable and Historic Computing*, pp. 147–166, Springer, 2011.

[41] S. K. Baruah, A. Burns, and R. I. Davis, “Response-time analysis for mixed criticality systems,” in *2011 IEEE 32nd Real-Time Systems Symposium*, pp. 34–43, IEEE, 2011.

[42] P. Ekberg and W. Yi, “Bounding and shaping the demand of generalized mixed-criticality sporadic task systems,” *Real-time systems*, vol. 50, no. 1, pp. 48–86, 2014.

[43] RTCA, “DO-178c,” *Software Considerations in Airborne Systems and Equipment Certification, Crosstalk Magazine*, 2011.

[44] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, “How realistic is the mixed-criticality real-time system model?,” in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pp. 139–148, 2015.

[45] A. E. E. Committee, "Avionics application software standard interface part 1-required services," *ARINC Document ARINC Specification 653P1-2, Aeronautical Radio, Annapolis, Maryland*, 2006.

[46] *AUTOSAR*. Available: `https://www.autosar.org/`, Last accessed: March 2021.

[47] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pp. 193–202, 2001.

[48] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, Oct. 2011.

[49] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor scheduling for real-time systems*. Springer, 2015.

[50] H. Kopetz, "Sparse time versus dense time in distributed real-time systems.," in *ICDCS*, pp. 460–467, 1992.

[51] G. J. Fohler, *Flexibility in statically scheduled hard real-time systems*. Citeseer, 1994.

[52] W. Steiner, "An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks," in *2010 31st IEEE Real-Time Systems Symposium*, pp. 375–384, IEEE, 2010.

[53] H. Kopetz, "Event-triggered versus time-triggered real-time systems," in *Operating Systems of the 90s and Beyond*, pp. 86–101, Springer, 1991.

[54] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proceedings 16th IEEE Real-Time Systems Symposium*, pp. 152–161, 1995.

[55] S. Schorr, *Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures*. doctoral thesis, Technische Universität Kaiserslautern, 2015.

[56] A. A. J. Syed, *Model-Based Design and Adaptive Scheduling of Distributed Real-Time Systems*. doctoral thesis, Technische Universität Kaiserslautern, 2018.

[57] J. Real, S. Sáez, and A. Crespo, "A hierarchical architecture for time- and event-triggered real-time systems," *Journal of Systems Architecture*, vol. 101, p. 101652, 2019.

[58] H.-K. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources," in *2011 International Conference on Parallel Processing*, pp. 753–762, 2011.

[59] M. Nasri and B. B. Brandenburg, "Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper)," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 75–86, 2017.

[60] M. M. van den Heuvel, R. J. Bril, X. Zhang, S. M. J. Abdullah, and D. Isovic, "Limited preemptive scheduling of mixed time-triggered and event-triggered tasks," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–9, 2013.

[61] Y. Xie, H. Yan, and Z. Pang, "Mixed time-triggered and event-triggered controller for industrial iot applications," in *2016 IEEE International Conference on Industrial Technology (ICIT)*, pp. 2064–2067, 2016.

[62] H. Kopetz, "Hard real-time systems i," 1991.

[63] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered ethernet (tte) design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pp. 22–33, 2005.

[64] H. Kopetz and G. Grunsteidl, "Ttp - a time-triggered protocol for fault-tolerant real-time systems," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 524–533, 1993.

[65] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 109–118, IEEE, 2014.

[66] P. Radojković, S. Girbal, A. Grasset, E. Quinones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–25, 2012.

[67] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on wcet analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.

[68] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 741–746, 2010.

[69] J. Yan and W. Zhang, "Wcet analysis for multi-core processors with shared l2 instruction caches," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 80–89, 2008.

[70] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multi-core platforms," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pp. 99–108, 2012.

[71] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling Cache Coherence to Expose Interference," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)* (S. Quinton, ed.), vol. 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 18:1–18:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[72] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "Wcet(m) estimation in multi-core systems using single core equivalence," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 174–183, 2015.

[73] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler, "Analysis of dynamic memory bandwidth regulation in multi-core real-time systems," 2018.

[74] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in *2007 44th ACM/IEEE Design Automation Conference*, pp. 264–265, 2007.

[75] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches," in *2009 30th IEEE Real-Time Systems Symposium*, pp. 68–77, IEEE, 2009.

[76] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pp. 49–60, 2007.

[77] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 215–224, 2010.

[78] T. Kelter, T. Harde, P. Marwedel, and H. Falk, "Evaluation of resource arbitration methods for multi-core real-time systems," in *13th International Workshop on Worst-Case Execution Time Analysis* (C. Maiza, ed.), vol. 30 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 1–10, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[79] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 213–222, 2011.

[80] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE*

Real-Time and Embedded Technology and Applications Symposium, pp. 269–279, 2011.

[81] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on arm-based heterogeneous platforms," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pp. 1297–1302, 2017.

[82] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[83] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, "Deterministic execution model on cots hardware," in *Architecture of Computing Systems – ARCS 2012* (A. Herkersdorf, K. Römer, and U. Brinkschulte, eds.), (Berlin, Heidelberg), pp. 98–110, Springer Berlin Heidelberg, 2012.

[84] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 240–250, 2018.

[85] C. M. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*, pp. 103–120. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[86] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment." Poster präsentiert auf 14th Symposium on Operating Systems Principles (SOSP 1997), Saint-Malo, Frankreich, 5.–8. Oktober 1997, 1997.

[87] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 299–308, 2012.

[88] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 55–64, 2013.

[89] M. Behnam, R. Inam, T. Nolte, and M. Sjödin, "Multi-core composability in the face of memory-bus contention," *SIGBED Rev.*, vol. 10, p. 35–42, Oct. 2013.

[90] J. Nowotsch and M. Paulitsch, "Quality of service capabilities for hard real-time applications on multi-core processors," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, (New York, NY, USA), p. 151–160, Association for Computing Machinery, 2013.

[91] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)* (M. Bertogna, ed.), vol. 76 of *Leibniz International*

Proceedings in Informatics (LIPIcs), (Dagstuhl, Germany), pp. 2:1–2:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[92] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)* (B. B. Brandenburg, ed.), vol. 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 3:1–3:26, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[93] *Zynq UltraScale+ MPSoC Data Sheet:Overview.* Xilinx. Available: Document Number: DS891, Rev v1.9 `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`, Last accessed: May 2021.

[94] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-warp: A system-wide framework for memory bandwidth profiling and management," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, pp. 345–357, 2020.

[95] A. Stevens, "Quality of service (qos) in arm® systems: An overview," *ARM, Cambridge, UK, White Paper*, 2014.

[96] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems,"

[97] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 367–378, 2008.

[98] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, May 2017.

[99] *FreeBSD.* Available: `https://docs.freebsd.org/en/articles/vm-design/`, Last accessed: March 2021.

[100] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice—a low-cost high-speed address translation mechanism," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), p. 355–363, Association for Computing Machinery, 1990.

[101] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (New York, NY, USA), p. 89–102, Association for Computing Machinery, 2009.

[102] "Intel® resource director technology (intel® rdt)."

[103] "Amd64 technology platform quality of service extensions."

[104] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, 2014.

[105] G. Fohler, "Realizing changes of operational modes with a pre run-time scheduled hard real-time system," in *Responsive Computer Systems* (H. Kopetz and Y. Kakuda, eds.), (Vienna), pp. 287–300, Springer Vienna, 1993.

[106] H. Kopetz, R. Nossal, R. Hexel, A. Krüger, D. Millinger, R. Pallierer, C. Temple, and M. Krug, "Mode handling in the time-triggered architecture," *IFAC Proceedings Volumes*, vol. 30, no. 15, pp. 11–16, 1997. IFAC Workshop on Distributed Computer Control Systems (DCCS'97), Seoul, Korea, 28-30 July 1997.

[107] F. Heilmann, A. Syed, and G. Fohler, "Mode-changes in cots time-triggered network hardware without online reconfiguration," *SIGBED Rev.*, vol. 13, p. 55–60, Nov. 2016.

[108] F. Jahanian, R. Lee, and A. Mok, "Semantics of modechart in real time logic," in *[1988] Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track*, vol. 2, pp. 479–489, 1988.

[109] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," 2012.

[110] *KVM*. Available: `https://www.linux-kvm.org/`, Last accessed: March 2021.

[111] *Xen Project*. Available: `https://xenproject.org/`, Last accessed: March 2021.

[112] *Virtual Box*. Available: `https://www.virtualbox.org/`, Last accessed: March 2021.

[113] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[114] D. Marshall, "Understanding full virtualization, paravirtualization, and hardware assist," *VMWare White Paper*, vol. 17, p. 725, 2007.

[115] *Virtio*. Available: `https://wiki.libvirt.org/page/Virtio`, Last accessed: March 2021.

[116] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, "Ensuring robust partitioning in multicore platforms for ima systems," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pp. 7A4–1–7A4–9, 2012.

[117] FP7-ICT, *MULTI-cores PARtitioning for Trusted Embedded Systems - MULTI-PARTES project.* Grant agreement ID: 287702.

[118] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," in *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

[119] S. Campagna, M. Hussain, and M. Violante, "Hypervisor-based virtual hardware for fault tolerance in cots processors targeting space applications," in *2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 44–51, 2010.

[120] *Deadline Task Scheduling.* Available: `https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html`, Last accessed: March 2021.

[121] *RTDS-Based-Scheduler.* Available: `https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler`, Last accessed: March 2021.

[122] S. H. VanderLeest, "Arinc 653 hypervisor," in *29th Digital Avionics Systems Conference*, pp. 5.E.2–1–5.E.2–20, 2010.

[123] J. Perez, D. Gonzalez, S. Trujillo, T. Trapman, and J. M. Garate, "A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning," in *11th International TUV Rheinland Symposium "Functional Safety in Industrial Applications*, 2014.

[124] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, "Cloud computing: a perspective study," *New generation computing*, vol. 28, no. 2, pp. 137–146, 2010.

[125] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, 2014.

[126] H. Kopetz, "The fault hypothesis for the time-triggered architecture," in *Building the Information Society* (R. Jacquart, ed.), (Boston, MA), pp. 221–233, Springer US, 2004.

[127] D. Powell, "Failure mode assumptions and assumption coverage," in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 386–395, 1992.

[128] J. H. Wensley, "Sift: Software implemented fault tolerance," in *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), (New York, NY, USA), p. 243–253, Association for Computing Machinery, 1972.

[129] A. Gerstinger, H. Kantz, and C. Scherrer, "Tas control platform: A platform for safety-critical railway applications," *ERCIM News*, vol. 2008, no. 75, 2008.

[130] S. Porcarelli, M. Castaldi, F. Di Giandomenico, A. Bondavalli, and P. Inverardi, "A framework for reconfiguration-based fault-tolerance in distributed systems," in *Architecting Dependable Systems II* (R. de Lemos, C. Gacek, and A. Romanovsky, eds.), (Berlin, Heidelberg), pp. 167–190, Springer Berlin Heidelberg, 2004.

[131] E. Strunk, J. Knight, and M. Aiello, "Assured reconfiguration of fail-stop systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pp. 2–11, 2005.

[132] E. Strunk and J. Knight, "Dependability through assured reconfiguration in embedded system software," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 172–187, 2006.

[133] S. Ellis, "Dynamic software reconfiguration for fault-tolerant real-time avionic systems," *Microprocessors and Microsystems*, vol. 21, no. 1, pp. 29–39, 1997. Proceedings of the 1996 Avionics Conference and Exhibition.

[134] R. Fuchsen, "Ima nextgen: A new technology for the scarlett program," *IEEE Aerospace and Electronic Systems Magazine*, vol. 25, no. 10, pp. 10–16, 2010.

[135] *Distributed equipment Independent environment for Advanced avioNic Applications (DIANA).* FP6-2005-AERO-1, Coordinator: Skysoft Portugal, Available: `https://trimis.ec.europa.eu/project/distributed-equipment-independent-environment-advanced-avionic-applications`, Last accessed: May 2021.

[136] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, pp. 203–226, 2019.

[137] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[138] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: An introduction," *The Linux Foundation, Jan*, 2018.

[139] V. Dhillon, D. Metcalf, and M. Hooper, "The hyperledger project," in *Blockchain enabled applications*, pp. 139–149, Springer, 2017.

[140] R. Obermaisser and H. Kopetz, "From ARTEMIS Requirements to a Cross-Domain Embedded System Architecture," in *ERTS2 2010, Embedded Real Time Software and Systems*, (Toulouse, France), May 2010.

[141] A. Eckel, P. Milbredt, Z. Al-Ars, S. Schneele, B. Vermeulen, G. Csertán, C. Scheerer, N. Suri, A. Khelil, G. Fohler, *et al.*, "Indexys, a logical step beyond genesys," in *International Conference on Computer Safety, Reliability, and Security*, pp. 431–451, Springer, 2010.

[142] G. Jolliffe, "Cost-efficient methods and processes for safety relevant embedded systems (cesar)–an objective overview," in *Making Systems Safer*, pp. 37–50, Springer, 2010.

[143] G. Kornaros, M. D. Grammatikakis, and M. Coppola, "Towards full virtualization of heterogeneous noc-based multicore embedded architectures," in *2012 IEEE 15th International Conference on Computational Science and Engineering*, pp. 345–352, 2012.

[144] D. Kaule, J. Becker, H.-U. Michel, and O. Sander, "Automotive railway avionics multicore systems : aramis : Schlussbericht : Laufzeit des vorhabens: 01.12.2011-30.11.2014, aramis: Schlussbericht bmw forschung und technik gmbh, aramis: Final report bmw forschung und technik gmbh," tech. rep., BMW Forschung und Technik GmbH, München, 2015.

[145] *Automotive Railway Avionics Multicore Systems (ARAMiS) II.* Coordinator: Karlsruhe Institute of Technology, Available: https://www.aramis2.org/, Last accessed: May 2021.

[146] R. Görgen, K. Grüttner, F. Herrera, P. Peñil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, and D. Quaglia, "Contrex: Design of embedded mixed-criticality control systems under consideration of extra-functional properties," in *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 286–293, 2016.

[147] W. Weber, A. Hoess, F. Oppenheimer, B. Koppenhöefer, B. Vissers, and B. Nordmoen, "Emc2 a platform project on embedded microcontrollers in applications of mobility, industry and the internet of things," in *2015 Euromicro Conference on Digital System Design*, pp. 125–130, 2015.

[148] *SAFety and secURity by design for interconnected mixed-critical cyber-physical systems (SAFURE).* H2020-EU.2.1.1.1, Coordinator: TECHNIKON Forschungs- und Planungsgesellschaft mbH, Available: https://cordis.europa.eu/project/id/644080/results, Last accessed: May 2021.

[149] *Safe and secure mixed-criticality systems with low power requirements (SAFEPOWER).* H2020-EU.2.1.1.1, Coordinator: IKERLAN S. COOP, Available: http://safepower-project.eu/project/, Last accessed: May 2021.

[150] *Dependable Real-time Infrastructure for Safety-critical Computer (DE-RISC).* H2020-EU.3. and H2020-EU.2.1., Coordinator: fentISS: Fent Innovative Software Solutions, Available: https://derisc-project.eu/, Last accessed: May 2021.

[151] *Technology Readiness Levels (TRL).* HORIZON 2020 - Work programme 2014-2015, General Annexes, Available:https://ec.europa.eu/research/participants

/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf, Last accessed: May 2021.

[152] Thales, *ÖBB Infrastruktur and Thales work on cloud-interlocking*. Sep 2020. Available:https://www.thalesgroup.com/en/austria/news/obb-infrastruktur-and-thales-work-cloud-interlocking, Last accessed: May 2021.

[153] A. Burns, T. Fleming, and S. Baruah, "Cyclic executives, multi-core platforms and mixed criticality applications," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 3–12, 2015.

[154] *Product SEcurity for Cross domain REliable Dependable Automated Systems*. H2020-EU.2.1.1.7., Coordinator: NXP Semiconductors Netherlands BV, Available: https://secredas-project.eu/, Last accessed: May 2021.

[155] CENELEC, E.N., "50126-Railway Applications: The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)," *European Committee for Electrotechnical Standardization*, 1999.

[156] CENELEC, E.N., "50128-Railway Applications: Software for Railway Control and Protection Systems," *European Committee for Electrotechnical Standardization*, 2011.

[157] CENELEC, E.N., "50129-Railway Applications: Communication, signalling and processing systems - Safety related electronic systems for signalling," *European Committee for Electrotechnical Standardization*, 2018.

[158] S. Resch, A. Steininger, and C. Scherrer, "A composable real-time architecture for replicated railway applications," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 472–485, 2015.

[159] J. Peleska, "New distribution paradigms for railway interlocking," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 434–448, Springer, 2020.

[160] R. Lardennois, "Safety: single coded processor architecture combined with asic provide a cost efficient and flexible solution to safety issues," *IFAC Proceedings Volumes*, vol. 27, no. 12, pp. 971–976, 1994.

[161] N. Shankaran, N. Roy, D. C. Schmidt, X. D. Koutsoukos, Y. Chen, and C. Lu, "Design and performance evaluation of an adaptive resource management framework for distributed real-time and embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1–20, 2008.

[162] N. Shankaran, X. D. Koutsoukos, D. C. Schmidt, Y. Xue, and C. Lu, "Hierarchical control of multiple resources in distributed real-time and embedded systems," *Real-Time Systems*, vol. 39, no. 1, pp. 237–282, 2008.

[163] D. Schmidt and F. Kuhns, "An overview of the real-time corba specification," *Computer*, vol. 33, no. 6, pp. 56–63, 2000.

[164] D. Schmidt, A. Gokhale, T. H. Harrison, D. Levine, and C. Cleeland, "Tao: a high-performance orb endsystem architecture for real-time corba," *Document submitted as an RFI response to the OMG Special Interest Group on Real-time CORBA*, 1997.

[165] Y. Krishnamurthy, I. Pyarali, C. Gill, L. Mgeta, Y. Zhang, S. Torn, and D. Schmidt, "The design and implementation of real-time corba 2.0: dynamic scheduling in tao," in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, pp. 121–129, 2004.

[166] M. Maggio, E. Bini, G. Chasparis, and K.-E. Årzén, "A game-theoretic resource manager for rt applications," in *2013 25th Euromicro Conference on Real-Time Systems*, pp. 57–66, 2013.

[167] G. Chasparis, M. Maggio, K.-E. Årzén, and E. Bini, "Distributed management of cpu resources for time-sensitive applications," in *2013 American Control Conference*, pp. 5305–5312, 2013.

[168] V. Millnert, J. Eker, and E. Bini, "End-To-End Deadlines over Dynamic Topologies," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)* (S. Quinton, ed.), vol. 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 10:1–10:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[169] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "Autonomic resource management in virtualized data centers using fuzzy logic-based approaches," *Cluster Computing*, vol. 11, no. 3, pp. 213–227, 2008.

[170] W. Tärneberg, A. Mehta, E. Wadbro, J. Tordsson, J. Eker, M. Kihl, and E. Elmroth, "Dynamic application placement in the mobile cloud network," *Future Generation Computer Systems*, vol. 70, pp. 163–177, 2017.

[171] G. Durrieu and C. Pagetti, "Grec: Automatic computation of reconfiguration graphs for multi-core platforms," *ACM Trans. Embed. Comput. Syst.*, vol. 18, Oct. 2019.

[172] A. Kritikakou, C. Pagetti, M. Roy, C. Rochange, M. Faugère, S. Girbal, and D. Gracia Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *22nd International Conference on Real-Time Networks and Systems*, (Versailles, France), Oct. 2014.

[173] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[174] *AXI Performance Monitor v5.0.* Xilinx. Available: Document Number: PG037 https://www.xilinx.com/support/documentation/ip_documentation/axi_perf_mon/v5_0/pg037_axi_perf_mon.pdf, Last accessed: March 2021.

[175] *Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring.* Intel. Available: https://software.intel.com/content/www/us/en/develop/download/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html, Last accessed: March 2021.

[176] *EREF: A Programmer's Reference Manual for Freescale Power Architecture Processors.* Freescale Semiconductor Inc. Rev. 1 (EIS 2.1), Available: https://www.nxp.com/docs/en/product-brief/T4240PB.pdf, Last accessed: March 2021.

[177] *Cortex-A9.* ARM. Available: Document Number: rp41 file:///tmp/mozilla_gala0/DDI0388I_cortex_a9_r4p1_trm.pdf, Last accessed: March 2021.

[178] *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Intel. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4; Available: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, Last accessed: March 2021.

[179] *Events for Intel microarchitecture code name Cascade Lake-X.* Intel. Available: https://perfmon-events.intel.com, Last accessed: Aug 2021.

[180] *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition.* ARM ltd., DDI 0406C.d (ID040418), Available: https://developer.arm.com/documentation/ddi0406/latest, Last accessed: May 2021.

[181] *Accuracy of the Performance Monitors.* ARM ltd., Available: https://developer.arm.com/documentation/ddi0406/cb/Debug-Architecture/The-Performance-Monitors-Extension/Accuracy-of-the-Performance-Monitors, Last accessed: May 2021.

[182] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 215–224, 2013.

[183] *The Linux Kernel documentation.* Available: https://www.kernel.org/doc/html/latest/index.html, Last accessed: Aug 2021.

[184] R. Obermaisser, P. Peti, W. Elmenreich, and T. Losert, "Monitoring and configuration in a smart transducer network," in *IEEE Real-time embedded system workshop*, pp. 1–7, 2001.

[185] B. Bonakdarpour and S. Fischmeister, "Runtime monitoring of time-sensitive systems," in *Runtime Verification* (S. Khurshid and K. Sen, eds.), (Berlin, Heidelberg), pp. 19–33, Springer Berlin Heidelberg, 2012.

[186] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *International Symposium on Formal Methods*, pp. 88–102, Springer, 2011.

[187] S. Fischmeister and Y. Ba, "Sampling-based program execution monitoring," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pp. 133–142, 2010.

[188] R. Medhat, B. Bonakdarpour, D. Kumar, and S. Fischmeister, "Runtime monitoring of cyber-physical systems under timing and memory constraints," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 4, pp. 1–29, 2015.

[189] M. Neukirchner, K. Lampka, S. Quinton, and R. Ernst, "Multi-mode monitoring for mixed-criticality real-time systems," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, 2013.

[190] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*, vol. 2050. Springer Science & Business Media, 2001.

[191] M. Åsberg, T. Nolte, C. M. Otero Pérez, and S. Kato, "Execution time monitoring in linux," in *2009 IEEE Conference on Emerging Technologies Factory Automation*, pp. 1–4, 2009.

[192] M. T. Khan, D. Serpanos, and H. Shrobe, "A rigorous and efficient run-time security monitor for real-time critical embedded system applications," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 100–105, 2016.

[193] *Real-Time Linux Wiki*. Available: `https://rt.wiki.kernel.org/index.php/Main_Page`, Last accessed: March 2021.

[194] *Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual*. Intel. Rev. 1.0, Available: `https://software.intel.com/content/dam/develop/external/us/en/documents/180115-intel-rdtcascadelake-serverreferencemanual-806717.pdf`, Last accessed: Aug 2021.

[195] H. Ahmadian and R. Obermaisser, "Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems," in *2015 Euromicro Conference on Digital System Design*, pp. 693–699, 2015.

[196] H. Ahmadian, R. Obermaisser, and M. Abuteir, "Time-triggered and rate-constrained on-chip communication in mixed-criticality systems," in *2016 IEEE*

10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC), pp. 117–124, 2016.

[197] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, pp. 3–5, 2008.

[198] D. J. Bernstein, "The poly1305-aes message-authentication code," in *International workshop on fast software encryption*, pp. 32–49, Springer, 2005.

[199] ISO/IEC, *ISO/IEC 29192-2:2012. Information technology - Security techniques - Lightweight cryptography - Part 2: Block ciphers*. International Organization for Standardization, Geneva, CH, Standard, Jan 2012.

[200] T. Koller, *Communication Security for Distributed Mixed-Criticality Systems*. doctoral thesis, Universität Siegen, 2018.

[201] M. Robshaw and O. Billet, *New stream cipher designs: the eSTREAM finalists*, vol. 4986. Springer, 2008.

[202] ISO/IEC, *ISO/IEC 29192-1:2012. Information technology - Security techniques - Lightweight cryptography - Part 1: General*. International Organization for Standardization, Geneva, CH, Standard, Jan 2012.

[203] D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov, "Triathlon of lightweight block ciphers for the internet of things," *Journal of Cryptographic Engineering*, vol. 9, no. 3, pp. 283–302, 2019.

[204] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pp. 19–22, 2001.

[205] A. Langley, *TLS Symmetric Crypto*. Available: `https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html`, Last accessed: Aug 2021.

[206] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International workshop on cryptographic hardware and embedded systems*, pp. 450–466, Springer, 2007.

[207] OPC, *OPC 10000-5: OPC Unified Architecture*. Online versions of OPC UA specifications and information models. Available: `https://reference.opcfoundation.org/v104/Core/docs/Part5/`, Last accessed: Aug 2021.

[208] S.-H. Leitner and W. Mahnke, "Opc ua–service-oriented architecture for industrial applications," *ABB Corporate Research Center*, vol. 48, no. 61-66, p. 22, 2006.

[209] MQTT, *MQTT: The Standard for IoT Messaging*. Available: `https://mqtt.org/`, Last accessed: Aug 2021.

[210] S. Cavalieri and G. Cutuli, "Performance evaluation of opc ua," in *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*, pp. 1–8, 2010.

[211] M. Handosa, D. Gračanin, and H. G. Elmongui, "Performance evaluation of mqtt-based internet of things systems," in *2017 Winter Simulation Conference (WSC)*, pp. 4544–4545, 2017.

[212] Thales, *Thales unveils avionics 2020, the cockpit of the future*. Available: `https://onboard.thalesgroup.com/thales-unveils-avionics-2020-the-cockpit-of-the-future/`, Last accessed: Aug 2021.

[213] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS'14)*, (TOULOUSE, France), Feb. 2014.

[214] D. Isovic and G. Fohler, "Quality aware mpeg-2 stream adaptation in resource constrained systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pp. 23–32, 2004.

[215] Aeronautical Radio Inc, *Avionics Application Software Standard Interface*.

[216] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, P. Balbastre, and J.-J. Metge, "Xoncrete: a scheduling tool for partitioned real-time systems," in *ERTS2 2010, Embedded Real Time Software & Systems*, 2010.

[217] "The virtualization api."

[218] *RT-Xen Real-time Virtualization*. Available: `https://sites.google.com/site/realtimexen/`, Last accessed: March 2021.

[219] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pp. 10–pp, IEEE, 2005.

[220] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour, "Defects of the posix sporadic server and how to correct them," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 35–45, IEEE, 2010.

[221] "What is esxi?: Bare metal hypervisor: Esx," Dec 2020.

[222] BenjaminArmstrong, "Hyper-v technology overview."

[223] A. Chierici and R. Veraldi, "A quantitative comparison between xen and kvm," in *Journal of Physics: Conference Series*, vol. 219, p. 042005, IOP Publishing, 2010.

[224] S. G. Soriga and M. Barbulescu, "A comparison of the performance and scalability of xen and kvm hypervisors," in *2013 RoEduNet International Conference 12th Edition: Networking in Education and Research*, pp. 1–6, 2013.

[225] *RT-Tests*. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests, Last accessed: March 2021.

[226] L. Abeni and D. Faggioli, "Using xen and kvm as real-time hypervisors," *Journal of Systems Architecture*, vol. 106, p. 101709, 2020.

[227] *RT-Tests*. Available: https://wiki.gentoo.org/wiki/Sysbench, Last accessed: March 2021.

[228] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. Available: https://iperf.fr/, Last accessed: March 2021.

[229] M. Vanga, A. Gujarati, and B. B. Brandenburg, "Tableau: A high-throughput and predictable vm scheduler for high-density workloads," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[230] *Using the KVM API*. Available: https://lwn.net/Articles/658511/, Last accessed: March 2021.

[231] S. Schorr and G. Fohler, "Online admission of non-preemptive aperiodic tasks in offline schedules," *Proceedings Work-in-Progress Session*, 2010.

[232] W. Korn, P. Teller, and G. Castillo, "Just how accurate are performance counters?," in *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210)*, pp. 303–310, 2001.

[233] B. Ampel, M. Patton, and H. Chen, "Performance modeling of hyperledger sawtooth blockchain," in *2019 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pp. 59–61, 2019.

[234] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pp. 8D4–1–8D4–15, 2015.

Summary

Resource Management for Real-Time and Mixed-Critical Systems

Chapter I

This chapter gives a high-level overview of the dissertation.

Chapter II

This chapter introduces the terms and concepts used throughout this dissertation and presents the relevant work and relevant EU projects. In addition, it provides an overview of existing techniques to deal with contention in shared resources. The chapter also presents the state-of-the-art resource management frameworks for Real-Time Systems (RTS) and Mixed-Critical Systems (MCS). Lastly, this chapter describes the problem statements addressed in this dissertation.

Chapter III

This chapter presents our resource management framework and explains various requirements and challenges for the design of our framework.

The chapter explains how the framework combines the benefits of local and global resource management strategies and keeps the overhead low by decoupling global resource management from local resource management via a Global Resource Manager (GRM) and a set of Local Resource Managers (LRMs). In addition, the chapter gives an overview of how the framework allows multiple monitoring and scheduling techniques without tightly coupling them with the framework implementation (via the introduction of Local Resource Monitor (MON) and Local Resource Scheduler (LRS) modules).

The chapter introduces two resource management architectures:

1. A flat architecture where the GRM is at the top of the hierarchy and has a complete view of the entire system and directly supervises and controls all the LRMs.

2. A hierarchical architecture where the GRM sits at the top of the hierarchy while LRMs are present at different levels in the hierarchy. The chapter also explains

the advantages of hierarchical architecture for developing a scalable resource management framework to manage a distributed system consisting of heterogeneous nodes with different operating speeds and locations in the system structure.

The chapter also presents the concept of resource management domains based on the hierarchical resource management architecture. Resource management domains allow for effective reconfiguration at different levels in the system.

It can be error-prone and tedious for a system designer to correctly configure resource management according to each platform's low-level details, especially in heterogeneous distributed systems. The chapter explains how the resource management framework makes it easier for the system designers to provide resource management configuration parameters abstractly and select MONs and LRSs for each platform without the need to know or set fine-grained platform-specific configurations.

Chapter IV

This chapter explains the Local Resource Manager (LRM) of the resource management framework. There are two main types of modular LRM sub-components:

1. *Local Resource Monitor (MON)*: Each resource or application managed by the LRM has one or more MONs, each providing different monitoring services, such as availability monitoring or reliability monitoring. The chapter presents the following new MONs that we designed and implemented:
   - MON to interface with the hardware-specific monitoring features such as Performance Monitor Unit (PMU), Intel Memory Bandwidth Monitoring (MBM), and Cache Monitoring Technology (CMT).
   - MON[1] to detect permanent core failures on multicore nodes.
   - MON[1] for detecting potential deadline overrun by a critical Virtual Machine (VM) in the presence of interference from concurrently executing non-critical VMs.
   - MON that can use the XtratuM Health Monitoring (HM) API to write partition status or errors (not handled by XtratuM) to a HM log.
2. *Local Resource Scheduler (LRS)*: Each resource managed by the resource management framework is paired with a LRS. Each LRS schedules the use of the resource and controls application access to the resource. The chapter presents the following new LRSs that we designed and implemented:
   - LRS[1] for scheduling tasks of a critical application running in a XtratuM hypervisor partition (VM).
   - LRS[1] to reconfigure scheduling plans (modes) of XtratuM hypervisor.
   - LRS to provide an interface for the resource management to interact with Intel Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) hardware features.

[1] Designed and implemented together with ONERA and Thales R&T.

- LRS to support scheduling of Time-Triggered (TT) tasks or VMs in Linux or Kernel Virtual Machine (KVM).

Moreover, this chapter explains the various services provided by an LRM, such as reading information from its MONs, calculating the abstract state of resources, sending/receiving updates/orders to/from the GRM or a lower- or higher-domain LRM, making local reconfiguration decisions, translating global orders, and configuring the LRSs. Finally, this chapter presents LRM policies[1] to manage the following situations:

1. Permanent failure of CPU cores on a node.

2. Node-level temporal overload situation (by potential deadline overrun monitoring).

Chapter V

This chapter introduces the Global Resource Manager (GRM) of our resource management architecture. The GRM provides various services, such as gathering updates from LRMs, making global reconfiguration decisions, obtaining or computing a new global configuration, and sending/receiving orders/updates to/from the LRMs. It also manages external input. Conceptually, only one GRM exists in the system, although distribution is possible for fault tolerance and scalability. Therefore, the GRM can be realized either by a single node or by a set of nodes.

The chapter proposes three conceptual communication channels among the central GRM and the LRMs (and between higher and lower domain LRMs) to exchange information – Update, Order and Membership channels. In addition, the chapter provides an overview about using two popular existing protocols (Message Queue Telemetry Transport (MQTT) and Open Platform Communications Unified Architecture (OPC UA)) for resource management communication, taking into consideration requirements such as membership, security, and reliable message delivery.

Unsecured resource management has several weak spots that an attacker can exploit. Therefore, the chapter analyzes the resource management communication, LRM, and GRM from a security viewpoint[2]. The chapter proposes security services for resource management communication to prevent various security attacks. In addition, it explains the three different security levels (Levels 0, 1, and 2) for resource management communication and two options to implement the security services – implementation as a support module or implementation as a layer between the resource managers and the underlying hypervisor or Operating System (OS). Moreover, the chapter discusses two options for security algorithms[2] – ChaCha20-Poly1305 and CLEFIA (in offset codebook operation mode).

Finally, this chapter discusses several limitations of a single central global resource manager. The chapter proposes distributed global decision-making instead of centralized decision-making to ensure fault-tolerance for global resource management. It describes a new type of component, the Distributed Global Resource Manager (DGRM), to replace the single central GRM. Furthermore, the chapter identifies the challenges and

[2]Based on inputs from the security experts at the University of Siegen

requirements concerning safety and security for the DGRMs. The chapter explains how a private (permissioned) blockchain called the Hyperledger Sawtooth [138] can help us to meet the challenges and requirements concerning safety and security and presents a design for a new Sawtooth transaction family called the Resource Manager (RM)-transaction family for implementing the DGRMs and the LRMs of our resource management framework.

Chapter VI

This chapter presents the avionics use case of the EU DREAMS project [2] consisting of two multicore processors and a Multi-Processor System on a Chip (MPSoC) connected via a TT-Ethernet (TTE) network. The system hosts three safety-critical avionics applications and multiple instances of a best-effort application.

The chapter also presents the avionics use case-specific implementation of the local and global resource management policy for core failure. It is based on mode[3] changes. It uses local and global reconfiguration graphs obtained offline via the DREAMS toolchain (Xoncrete and GREC tools). The chapter also gives an overview of how network reconfiguration can be achieved through a super schedule approach.

Furthermore, the chapter evaluates the proposed resource management framework with the avionics use case in two scenarios:

1. Permanent core failure: The evaluation focuses on the fault tolerance capacity of the resource management against core failures and the global reconfiguration delay for the avionics use case.

2. Temporal overload condition: The evaluation focuses on the efficiency of the resource management adaptation to improve the system utilization while ensuring the critical applications meet their timing requirements.

Finally, the chapter presents an experimental evaluation for determining the overhead of secure resource management communication (Chapter V) in the avionics use case. The evaluation considers all three security levels and both security algorithms (Chapter V).

Chapter VII

This chapter presents an existing real-time safety-critical railway use case from the EU SECREDAS project [154]. It explores virtualization technologies and cloud computing for migrating this use case from dedicated hardware solutions. The chapter examines existing virtualization technologies (hypervisors) for deploying a (private) Cloud on Commercial-Off-The-Shelf (COTS) server hardware to run the use case while meeting stringent safety requirements. The chapter presents qualitative and quantitative analyses of relevant cloud hypervisors considering railway-specific requirements. Based on the insights gained, the chapter gives suggestions for using an existing hypervisor, Kernel

[3] Also referred to as configurations or plans by hypervisors

Virtual Machine (KVM), with a new component to form a Real-Time Cloud (RT-Cloud) that can safely run the railway use case applications. The new component is a resource management layer that is based on our resource management framework. Finally, the chapter presents an evaluation for our TT-LRS (Chapter IV), initial experiments with Intel Memory Bandwidth Allocation (MBA) for memory regulation, and the evaluation of our blockchain-based distributed global resource management (Chapter V).

Chapter VIII

This chapter gives the conclusions. In addition, it describes the ongoing and future work.

Zusammenfassung

Ressourcenmanagement für Echtzeit- und gemischt-kritische Systeme

Echtzeitsysteme müssen die erwarteten logischen Ergebnisse innerhalb strenger Zeitvorgaben liefern. Sie lassen sich in verschiedenen Bereichen wiederfinden, wie z.B. Multimedia, der Luft- und Raumfahrt, der Bahn- und Automobilindustrie, in Kernkraftwerken und dem Gesundheitswesen. Ein beträchtlicher Anteil der Echtzeitsysteme sind eingebettete Systeme bei denen es nicht sofort ersichtlich ist, dass ein Computer beteiligt ist. Ein Beispiel für ein eingebettetes Echtzeitsystem, das uns im alltäglichen Leben begegnet, ist ein digitaler Medienplayer, bei dem die Nichteinhaltung von Fristen zu unerwünschten Verzögerungen führen kann. Wir bezeichnen solche Echtzeitsysteme, bei denen die Missachtung von Echtzeitvorgaben zu Funktions- oder Leistungsverlust ohne katastrophale Folgen führt, als unkritische Echtzeitsysteme. Oft legen wir, ohne uns dessen Bewusst zu sein, unser Leben in die Hände von Echtzeitsystemen und verlassen uns auf ihr ordnungsgemäßes Funktionieren. Die Nichteinhaltung von Fristen in den Echtzeitsystemen eines Flugzeugs würde z.B. katastrophale Folgen haben. Als sicherheitskritische Echtzeitsysteme werden Systeme bezeichnet, bei denen die Nichteinhaltung von Echtzeitvorgaben zum Tod, zu schweren Verletzungen, zur Beschädigung bzw. zum Verlust von Eigentum/Ausrüstung oder zu Umweltschäden führen kann. Sicherheitskritische Echtzeitsysteme müssen von Zertifizierungsstellen nach den entsprechenden Industrienormen zertifiziert werden. Die Zertifizierung ist erforderlich, um ihren sicheren Betrieb zu gewährleisten und die Risiken auf ein angemessenes Maß an Ausfallsicherheit zu reduzieren.

Viele Echtzeitsysteme verwenden traditionellerweise föderierte Architekturen, bei denen jede Echtzeitanwendung auf einer eigenen Hardwareplattform (einem so genannten Knoten) läuft. Die Knoten tauschen untereinander nur Steuer- und Sensordaten aus. Die föderierten Architekturen gewährleisten eine Fehlereingrenzung, begrenzen die Ausbreitung von Fehlern in einem Knoten auf andere Knoten und vermeiden unerwünschte Wechselwirkungen durch ihr Design. So kann jede Anwendung von den anderen isoliert zertifiziert werden. Die derzeitige Zunahme der implementierten Anwendungen in diesen Bereichen hat jedoch die Anzahl der Knoten im System drastisch erhöht. Infolgedessen nehmen Größe, Gewicht und Leistung (Engl.: Size, Weight and Power - SWaP), die erforderliche Verkabelung und die damit verbundenen Kosten zu. Diese Überlegungen

haben die Industrie veranlasst, sich von föderierten Architekturen abzuwenden und zu integrierten Architekturen überzugehen.

Integrierte Architekturen implementieren mehrere Echtzeitanwendungen auf einem einzigen gemeinsamen Knoten. Integrierte Architekturen haben an Popularität gewonnen, da sie den Nachteilen föderierter Architekturen entgegenwirken. Integrierte Architekturen verwenden häufig COTS-Multikern-Prozessoren (Commercial-Off-The-Shelf-Multicore-Processors) und Multiprozessorsysteme auf dem Chip (MPSoCs). Multicore-Prozessoren und MPSoCs haben mehrere CPU-Kerne auf einem einzigen Chip. Dies ermöglicht es ihnen eine bessere Leistung zu bieten und die Integration einer größeren Anzahl von Anwendungen ohne die physikalischen Beschränkungen von Einprozessoren, welche nur einen CPU-Kern auf dem Chip verwenden, zu unterstützen. So tragen sie zu einer weiteren Reduzierung von SWaP, Verkabelung, Kosten und Umweltbelastung bei. Zusätzlich zu den genannten Vorteilen stellt die Industrie auf Multicores um, weil sie erwartet, dass Uniprozessoren bald auf dem Massenmarkt überflüssig werden.

Ein weiterer bemerkenswerter Trend bei integrierten Architekturen sind Systeme mit gemischter Kritikalität. In gemischt-kritischen Systemen können Anwendungen unterschiedlicher Kritikalitätsstufen gleichzeitig auf einem Knoten ausgeführt werden und sich die Ressourcen des Knotens teilen, z.B. kann das sicherheitskritische Flugmanagementsystem und die nicht-kritische Bordunterhaltung für die Passagiere parallel auf einem einzigen gemeinsamen Knoten laufen.

Die Vorteile der integrierten Architektur sind für die Industrie ein überzeugender Grund sie einzusetzen. Allerdings gibt es auch einige Nachteile. Es ist schwierig, die erforderliche Isolierung zu erreichen, insbesondere bei Multicore-Plattformen und MPSoCs. Der Mangel an Isolierung entsteht durch die Konkurrenz um die gemeinsam genutzten Ressourcen wie den CPU, dem Gemeinsamen-Bus, dem Speicherkontroller und Netzwerk. Diese gemeinsam genutzten Ressourcen können unvorhersehbare Verzögerungen verursachen, die bei Echtzeitanwendungen zu Fristüberschreitungen führen. Außerdem sind die Grenzen für die Fehlerisolierung und -kontaminierung nicht so scharf definiert wie bei der föderierten Architektur. Daher ist es eine Herausforderung, sicherzustellen, dass Echtzeitsysteme ihre Fristen einhalten, vor allem, wenn es keine geeigneten Techniken zum Ressourcenmanagement gibt. Solche Techniken können die Isolierung und vorhersehbare Verzögerungen beim Zugriff auf gemeinsame Ressourcen garantieren. Die Zertifizierung von sicherheitskritischen Echtzeitanwendungen ohne diese Garantien ist schwierig.

Außerdem gibt es gegensätzliche Ziele für sicherheitskritische und nicht-kritische/"best-effort"-Anwendungen, die das bereits bestehende Problem des Ressourcenmanagements in integrierten Architekturen verschärfen. Die pessimistischen WCET (engl.: Worst Case Execution-Time) -Schätzungen der sicherheitskritischen Anwendungen führen im Durchschnitt zu einer erheblichen Unterauslastung der Ressourcen. Gleichzeitig erfordern die nicht-kritischen/"best-effort"-Anwendungen eine effiziente Ressourcennutzung, um die bestmögliche Servicequalität (engl.: Quality of Service, QoS) zu bieten.

Es gibt viele Allokations- und Scheduling-Methoden für Echtzeitsysteme und Systeme mit gemischter Kritikalität. Diese Methoden beruhen auf impliziten Annahmen einer konstanten Verfügbarkeit der einzelnen Ressourcen, insbesondere der CPU. Klassische Scheduling-Algorithmen wie Round-Robin oder Earliest-Deadline-First (EDF) gehen von

der Kontrolle über die gesamte CPU oder einen einzelnen Kern aus. Methoden wie der XtratuM-Hypervisor [15] oder PikeOS [16] planen nach festen Anteilen und erlauben hierarchisches Scheduling. Bei diesen bestehenden Ansätzen wird davon ausgegangen, dass die Verfügbarkeit einer konstanten Verarbeitungsmenge den Fortschritt der Anwendungen garantiert. Solche im Voraus geplanten Annahmen können auf einzelnen Knoten gemacht werden; in verteilten Systemen mit mehreren Knoten, die jeweils über mehrere Ressourcen verfügen, sind sie jedoch weniger sinnvoll. Wenn sich die Anwendungen, die Verfügbarkeit von Ressourcen oder die Systemkonfigurationen ändern, wird es schwierig, Annahmen über Ressourcen zu treffen. Es ist eine Herausforderung, Ende-zu-Ende-Einschränkungen zu erfüllen, wenn jede Ressource oder jeder Knoten einzeln betrachtet wird. Ein System mit sich dynamisch ändernder Verfügbarkeit und Anforderungen an Ressourcen erfordert ein globales Ressourcenmanagement. Das globale Ressourcenmanagement ermöglicht eine globale (systemweite) Sicht auf die Ressourcen und Anwendungen. Es koordiniert und passt die systemweiten Ressourcenallokationen an. Darüber hinaus muss es Anwendungen an die sich ändernde Ressourcenverfügbarkeit anpassen.

Die meisten bestehenden Ressourcenmanagement-Frameworks, wie z. B. [17, 18, 19, 20], konzentrieren sich auf Nicht-Echtzeitsysteme. Einige Beispiele für bestehende Konzepte für Ressourcenmanagement-Frameworks in Echtzeitsystemen sind das Matrix-Framework [21], das ACTORS-Framework [22], das ACROSS-Framework [23] und RT-ARM [24]. Mit dem Matrix-Ressourcenmanagement-Framework wurde eine Methode zur Verwaltung nicht-kritischer Echtzeitsysteme mit Einzelkernprozessoren vorgestellt. Das Ressourcenmanagement-Framework des ACTORS-Projekts ermöglichte die zeitliche Isolierung von Echtzeitsystemen auf CPU-Ebene durch Ressourcenreservierung in einer einzigen Multicore-Plattform. Im Projekt ACROSS, bot ein Trusted Resource Manager die Möglichkeit, die Kommunikation auf NoC umzuplanen. RTARM ist ein adaptives Ressourcenmanagement-Framework für Ende-zu-Ende-Ressourcenzuweisungen auf heterogenen COTS-Knoten. Es bietet den Anwendungen garantierte QoS. Diese bestehenden Arbeiten haben jedoch nicht eine Architektur berücksichtigt, die aus verteilten Systemen mit gemischter Kritikalität und Echtzeitsystemen besteht. Sie haben auch keine heterogenen Knoten auf der Basiskomponente von Multicore-Prozessoren oder MPSoCs berücksichtigt.

Ein einziger Fehler im globalen Ressourcenmanagement kann es unbrauchbar machen. Im schlimmsten Fall kann es fehlerhafte Entscheidungen über die Ressourcenallokation treffen, was zu einer Fristüberschreitung bei Echtzeitanwendungen führt. Daher ist es wichtig, die Sicherheit des globalen Ressourcenmanagements durch die Bereitstellung von Fehlertoleranz in seinen Komponenten zu gewährleisten. Mit dem Aufkommen von Industrie 4.0, Cloud Computing und dem Internet der Dinge (IoT) ist es notwendig geworden, strenge Echtzeitbeschränkungen und zuverlässigkeitsanforderungen mit dem Erfordernis einer offenen Welt zu kombinieren. Die globale Ressourcenmanagement für diese Systeme wird zu einem einladenden Ziel für passive und aktive Angreifer, da sie aktiv über die Ressourcenmanagement des Systems entscheiden kann. Im schlimmsten Fall kann es fehlerhafte Entscheidungen über die Ressourcenallokation treffen, was zu einer Fristüberschreitung bei Echtzeitanwendungen führt. Daher ist es wichtig, die Sicherheit des globalen Ressourcenmanagements durch die Bereitstellung von Fehlertoleranz in

seinen Komponenten zu gewährleisten. Mit dem Aufkommen von Industrie 4.0, Cloud Computing und dem Internet der Dinge (IoT) ist es notwendig geworden, strenge Echtzeitbeschränkungen und zuverlässigkeitsanforderungen mit dem Erfordernis einer offenen Welt zu kombinieren. Die globale Ressourcenverwaltung für diese Systeme wird zu einem einladenden Ziel für passive und aktive Angreifer, da sie aktiv die Ressourcenallokation bestimmen kann. Die Angreifer können sich beispielsweise als Ressourcenmanagementkomponente ausgeben und falsche Entscheidungen über die Ressourcenallokation treffen oder sensible Systeminformationen aus der Kommunikation des Ressourcenmanagements erhalten. Keines der vorhandenen Konzepte berücksichtigt sowohl Fehlertoleranz als auch Sicherheit für das globale Ressourcenmanagement in Echtzeitsystemen oder Systemen mit gemischter Kritikalität.

In dieser Dissertation wird ein domänenunabhängiges globales Ressourcenmanagement-Framework für verteilte gemischt-kritische Systeme und Echtzeitsysteme vorgeschlagen. Diese bestehen aus heterogenen Knoten, die sich aus Multicore-Prozessoren oder MPSoC zusammensetzen. Das globale Ressourcenmanagement-Framework kann eine effiziente Ressourcennutzung sicherstellen. Außerdem bietet es die erforderliche Ressourcenisolierung und ein vorhersehbares Verhalten beim Ressourcenzugriff. Das Ziel des Frameworks ist es, zu garantieren, dass alle Echtzeitanwendungen ihre Fristen einhalten (und sicherheitskritische Anwendungen ihre Sicherheitsstufen einhalten). Dieses Framework bietet auch Fehlertoleranz oder Wiederherstellung für Echtzeitanwendungen bei Änderungen der Betriebs- oder Umgebungsbedingungen. Gleichzeitig kann das Framework Ressourcen zu nicht-kritischen/"best-effort"-Anwendungen allokieren, um die QoS zu verbessern. In der Dissertation werden sowohl die Fehlertoleranz als auch die Sicherheit des Frameworks berücksichtigt.

Um die Echtzeitindustrie in die Lage zu versetzen, Cloud Computing zu nutzen und ein neues Marktsegment zu erschließen, z. B. den Bahnbetrieb als Cloud-basierten Dienst, wird in dieser Dissertation das globale Ressourcenmanagement-Framework erweitert, um eine Echtzeit-Cloud zu entwickeln. Diese kann Echtzeitsysteme und Systeme mit gemischter Kritikalität hosten. Abschließend wird in der Dissertation ein Anwendungsfall aus der Avionik vorgestellt, um ein globales Ressourcenmanagement-Framework für gemischt-kritische Systeme zu deMONtrieren, sowie ein Anwendungsfall aus dem Eisenbahnbereich, um den Einsatz von Echtzeit-Clouds mit globalem Ressourcenmanagement zu begründen.

Kapitel I

Dieses Kapitel gibt einen allgemeinen Überblick über die Dissertation.

Kapitel II

In diesem Kapitel werden die in dieser Dissertation verwendeten Begriffe und Konzepte eingeführt und die relevanten Arbeiten und EU-Projekte vorgestellt. Darüber hinaus gibt es einen Überblick über bestehende Techniken zum Umgang mit Konflikten in gemeinsam genutzten Ressourcen. In diesem Kapitel werden auch die aktuellen Rahmenwerke zur Ressourcenverwaltung für Echtzeitsysteme und gemischt-kritische Systeme vorgestellt. Anschließend werden die in dieser Dissertation behandelten Problemstellungen beschrieben.

Kapitel III

In diesem Kapitel wird unser Rahmenwerk für die Ressourcenverwaltung vorgestellt und es werden verschiedene Anforderungen und Herausforderungen des Designs unseres Rahmenwerks erläutert.

Das Kapitel erklärt, wie das Framework die Vorteile lokaler und globaler Ressourcenverwaltungsstrategien kombiniert und den Overhead niedrig hält, indem die globale Ressourcenverwaltung von der lokalen Ressourcenverwaltung über einen globalen Ressourcenmanager (GRM) und einer Reihe von lokalen Ressourcenmanagern (LRM) entkoppelt wird. Darüber hinaus gibt das Kapitel einen Überblick darüber, wie das Framework mehrere Überwachungs- und Scheduling-Techniken ermöglicht, ohne sie eng mit der Framework-Implementierung zu koppeln (durch die Einführung von lokalen Ressourcenmonitor- (MON) und lokalen Ressourcen-Scheduler (LRS) -Modulen).

Das Kapitel stellt zwei Architekturen zur Ressourcenverwaltung vor:

1. Eine flache Architektur, die den GRM an die Spitze der Hierarchie stellt und dieser einen vollständigen Überblick über das gesamte System hat und alle LRM direkt überwacht und kontrolliert.

2. Eine hierarchische Architektur, bei der der GRM an der Spitze der Hierarchie steht, während die LRM auf verschiedenen Ebenen der Hierarchie vorhanden sind. In diesem Kapitel werden auch die Vorteile einer hierarchischen Architektur für die Entwicklung eines skalierbaren Rahmens für das Ressourcenmanagement zur Verwaltung eines verteilten Systems erläutert, das aus heterogenen Knoten mit unterschiedlichen Betriebsgeschwindigkeiten und Standorten in der Systemstruktur besteht.

In diesem Kapitel wird auch das Konzept der Ressourcenmanagement-Domänen auf der Grundlage der hierarchischen Ressourcenmanagement-Architektur vorgestellt. Ressourcenmanagement-Domänen ermöglichen eine effektive Rekonfiguration auf verschiedenen Ebenen im System.

Für einen Systementwickler kann es, insbesondere in heterogenen verteilten Systemen, fehleranfällig und mühsam sein, die Ressourcenverwaltung entsprechend der Low-Level-Details der einzelnen Plattformen korrekt zu konfigurieren. In diesem Kapitel wird erläutert, wie das Rahmenwerk für die Ressourcenverwaltung den Systementwicklern die abstrakte Bereitstellung von Konfigurationsparametern für die Ressourcenverwaltung und die Auswahl von MON und LRS für jede Plattform erleichtert, ohne dass sie die detaillierten plattformspezifischen Konfigurationen kennen oder festlegen müssen.

Kapitel IV

In diesem Kapitel wird der LRM des Ressourcenmanagementrahmens erläutert. Es gibt zwei Haupttypen von modularen LRM-Unterkomponenten:

1. Ressourcenmonitor-Module (MON): Jede Ressource oder Anwendung, die vom LRM verwaltet wird, verfügt über einen oder mehrere MON, die jeweils unterschiedliche Überwachungsdienste bereitstellen, wie z.B. die Verfügbarkeits- oder Zuverlässigkeitsüberwachung. In diesem Kapitel werden die folgenden neuen MON vorgestellt, die wir entworfen und implementiert haben:
   - MON, um mit den hardwarespezifischen Überwachungsfunktionen wie Performance Counter, Intel Memory Bandwidth Monitoring (MBM) und Cache Monitoring Technology (CMT) zu kommunizieren.
   - MON[1], um permanente Kernausfälle auf Multicore-Knoten zu erkennen.
   - MON[1] zur Erkennung einer potenziellen Fristüberschreitung durch eine kritische virtuelle Maschine (VM) in Gegenwart von Störungen durch gleichzeitig ausgeführte nicht-kritische virtuelle Maschinen.
   - MON[1], das die XtratuM Health Monitoring (HM) API verwenden kann, um den Partitionsstatus oder Fehler (nicht von XtratuM gehandhabt) in ein HM Protokoll zu schreiben.
2. Lokale Ressourcen-Scheduler-Module (LRS): Jede vom Ressourcenmanagement-Framework verwaltete Ressource ist mit einem LRS gekoppelt. Jeder LRS plant die Nutzung der Ressource und steuert den Anwendungszugriff auf die Ressource. In diesem Kapitel werden die folgenden neuen LRS vorgestellt, die wir entworfen und implementiert haben:
   - LRS[1] zur Planung von Aufgaben einer kritischen Anwendung, die in einer XtratuM-Hypervisor-Partition (VM) läuft.
   - LRS[1] zur Neukonfiguration der Scheduling-Pläne (Modi) des XtratuM-Hyper visors.
   - LRS zur Bereitstellung einer Schnittstelle für die Ressourcenverwaltung zur Interaktion mit Intel Cache Allocation Technology (CAT) und Memory Bandwidth Allocation (MBA) Hardwarefunktionen.

[1] Entwickelt und implementiert in Zusammenarbeit mit ONERA und Thales R&T.

- LRS zur Unterstützung der Planung von zeitgesteuerten Aufgaben oder VMs in Linux oder Kernel Virtual Machine (KVM).

Darüber hinaus erläutert dieses Kapitel die verschiedenen von einem LRM bereitgestellten Dienste, wie z. B. das Lesen von Informationen von seinen MON, die Berechnung des abstrakten Zustands von Ressourcen, das Senden/Empfangen von Aktualisierungen/-Bestellungen an/von den/dem GRM oder einem LRM einer niedrigeren oder höheren Domäne, das Treffen lokaler Rekonfigurationsentscheidungen, die Übersetzung globaler Bestellungen und die Konfiguration des LRS. Schließlich werden in diesem Kapitel LRM-Richtlinien[1] vorgestellt, um die folgenden beiden Situationen zu bewältigen:

1. Permanenter Ausfall von CPU-Kernen auf einem Knoten.

2. Zeitliche Überlastungssituation auf Knotenebene (durch potenzielle Fristüberschreitungsüberwachung).

Kapitel V

Dieses Kapitel stellt den GRM unserer Ressourcenverwaltungsarchitektur vor. Der GRM bietet verschiedene Dienste an, wie z.B. das Sammeln von Aktualisierungen von LRMs, das Treffen von globalen Rekonfigurationsentscheidungen, das Erhalten oder Berechnen einer neuen globalen Konfiguration und das Senden/Empfangen von Aufträgen/Aktualisierungen an/von die/den LRMs. Es verwaltet auch externe Eingaben. Konzeptionell existiert nur ein GRM im System, obwohl eine Verteilung auf mehrere GRM aus Gründen der Fehlertoleranz und Skalierbarkeit möglich ist. Daher kann der GRM entweder durch einen einzelnen Knoten oder durch eine Gruppe von Knoten realisiert werden.

Das Kapitel schlägt drei Kommunikationskanäle zwischen dem zentralen GRM und den LRMs (und zwischen LRMs auf höheren und niedrigeren Domänen) zum Austausch von Informationen vor: Aktualisierungs-, Bestell- und Mitgliedschaftskanäle. Darüber hinaus bietet das Kapitel einen Überblick über die Verwendung von zwei gängigen etablierten Protokollen (MQTT und OPCUA) für die Kommunikation des Ressourcenmanagements, wobei Anforderungen wie Mitgliedschaft, Sicherheit und zuverlässige Nachrichtenübermittlung berücksichtigt werden.

Ungesichertes Ressourcenmanagement hat mehrere Schwachstellen, die ein Angreifer ausnutzen kann. Daher werden in diesem Kapitel die Ressourcenmanagement-Kommuni kation, LRM und GRM aus einer Sicherheitsperspektive analysiert. Das Kapitel schlägt Sicherheitsdienste für die Kommunikation des Ressourcenmanagements vor, um verschiedene Sicherheitsangriffe zu verhindern. Darüber hinaus werden die drei verschiedenen Sicherheitsstufen (Level 0, 1 und 2) für die Ressourcenmanagement-Kommunikation und zwei Optionen zur Implementierung der Sicherheitsdienste erläutert - die Implementierung als Support-Modul oder als Schicht zwischen den Ressourcenmanagern und dem zugrunde liegenden Hypervisor oder Betriebssystem. Darüber hinaus werden in diesem Kapitel zwei Optionen für Sicherheitsalgorithmen erörtert – ChaCha20-Poly1305 und CLEFIA (in der Betriebsart Offset-Codebook).

Anschließend werden in diesem Kapitel mehrere Einschränkungen eines einzigen zentralen globalen Ressourcenmanagers erörtert. Das Kapitel schlägt eine verteilte globale Entscheidungsfindung anstelle einer zentralen Entscheidungsfindung vor, um Fehlertoleranz für die globale Ressourcenverwaltung zu gewährleisten. Es beschreibt einen neuen Komponententyp, den verteilten globalen Ressourcenmanager (VGRM), der den zentralen GRM ersetzen soll. Des Weiteren werden in diesem Kapitel die Herausforderungen und Anforderungen an die Sicherheit der VGRM ausfindig gemacht. Das Kapitel erklärt, wie eine private (genehmigte) Blockchain namens Hyperledger Sawtooth uns dabei helfen kann, die Herausforderungen und Anforderungen, welche die Sicherheit betreffen, zu erfüllen, und stellt einen Entwurf für eine neue Sawtooth-Transaktionsfamilie namens Ressourcenmanager (RM) -Transaktionsfamilie zur Implementierung der VGRM und LRM unseres Ressourcenverwaltungsrahmens vor.

Kapitel VI

In diesem Kapitel wird der Avionik-Anwendungsfall des EU-DREAMS-Projekts vorgestellt, der aus zwei Multicore-Prozessoren und einem über ein zeitgesteuertes Ethernet-Netzwerk verbundenen MPSoC besteht. Das System hostet drei sicherheitskritische Avionik-Anwendungen und mehrere Instanzen einer Best-Effort-Anwendung.

Das Kapitel stellt auch die Avionik-Anwendungsfall-spezifische Implementierung der lokalen und globalen Ressourcenmanagement-Richtlinie für Kernausfälle vor. Sie basiert auf Änderungen der Modi (von Hypervisor-Entwicklern auch als Konfigurationen oder Pläne bezeichnet). Sie verwendet lokale und globale Rekonfigurationsgraphen, die offline über die DREAMS-Toolchain (Xoncrete- und GREC-Tools) gewonnen wurden. Das Kapitel gibt außerdem einen Überblick darüber, wie die Netzwerkrekonfiguration durch einen „Super-Scheduler“-Ansatz erreicht werden kann.

Darüber hinaus wird in diesem Kapitel der vorgeschlagene Rahmen für das Ressourcenmanagement anhand des Anwendungsfalls Avionik in zwei Szenarien evaluiert:

1. Permanenter Kernausfall: Die Bewertung konzentriert sich auf die Fehlertoleranzkapazität des Ressourcenmanagements gegenüber Kernausfällen und die globale Rekonfigurationsverzögerung für den Anwendungsfall Avionik.
2. Zeitweilige Überlastbedingungen: Die Bewertung konzentriert sich auf die Effizienz der Anpassung des Ressourcenmanagements, um die Systemauslastung zu verbessern und gleichzeitig sicherzustellen, dass die kritischen Anwendungen ihre zeitlichen Anforderungen erfüllen.

Anschließend wird in diesem Kapitel eine experimentelle Evaluierung zur Bestimmung des Overheads der sicheren Ressourcenmanagement-Kommunikation im Anwendungsfall Avionik vorgestellt. Die Bewertung berücksichtigt alle drei Sicherheitsstufen und beide Sicherheitsalgorithmen.

Kapitel VII

In diesem Kapitel wird ein bestehender sicherheitskritischer Echtzeit-Eisenbahn-Anwend ungsfall aus dem EU-Projekt SECREDAS präsentiert. Es untersucht Virtualisierungs-

technologien und Cloud Computing, um diesen Anwendungsfall von dedizierten Hardwarelösungen zu migrieren. Das Kapitel untersucht bestehende Virtualisierungstechnologien (Hypervisoren) für den Einsatz einer (privaten) Cloud auf COTS-Serverhardware, um den Anwendungsfall unter Einhaltung strenger Sicherheitsanforderungen auszuführen. Das Kapitel präsentiert qualitative und quantitative Analysen relevanter Cloud-Hypervisoren unter Berücksichtigung bahnspezifischer Anforderungen. Basierend auf den gewonnenen Erkenntnissen gibt das Kapitel Vorschläge für die Verwendung eines bestehenden Hypervisors, KVM, mit einer neuen Komponente, um eine Echtzeit-Cloud zu bilden, welche die Anwendungen des Anwendungsfalls Eisenbahn sicher ausführen kann. Bei der neuen Komponente handelt es sich um eine Ressourcenverwaltungsschicht, die auf unserem Ressourcenverwaltungsrahmen basiert. Letztlich werden in diesem Kapitel eine Evaluierung unseres zeitgesteuerten LRS, erste Experimente mit Intel MBA zur Speicherregulierung und unseres Blockchain-basierten verteilten globalen Ressourcenmanagements vorgestellt.

Kapitel VIII

Dieses Kapitel enthält die Schlussfolgerungen. Es beschreibt auch die laufenden und zukünftigen Arbeiten.

Gautam Gala

Chair of Real Time Systems
TU Kaiserslautern, Postfach 3049
67653 Kaiserslautern

+49 (0)631 205 2715

gala@eit.uni-kl.de

www.linkedin.com/in/gautamgala/

ResearchGate: Gautam-Gala-2

ORCID: 0000-0003-4625-9694

Languages

Language	Level
English	●●●●●●
German	●●●●○○
Hindi	●●●●●●
Gujarati	●●●●●●
Kutchi	●●●●●●
Marathi	●●●●○○

Extracurricular Activities

- Flugsportverein Kaiserslautern e.V. (2019-present)
- Chairmain of Electrical Engineering Students' European assoCiation (EESTEC) Kaiserslautern branch (2013 - 2016)
- Convener of Dhristi Technical Hobby club (2009/10)
- Team leader for SVNIT Robocon Team (2008-2010)

Education

Nov/2021 – Nov/2017
Ph.D. in Electrical and Computer Engineering
Technical University of Kaiserslautern, Germany
PhD Thesis: Resource Management for Real-Time and Mixed-Critical Systems
Research Supervisor: Prof. Dr. Gerhard Fohler

Apr/2014 – Oct/2011
M.Sc. in Electrical and Computer Engineering
Technical University of Kaiserslautern, Germany
Master Thesis: Slot Reservation-based Shared-Bus Arbitration Scheme for Real-time Systems
Research Supervisor: Prof. Dr. Gerhard Fohler

May/2011 – July/2007
Bachelor of Technology in Electronics
SV. National Institute of Technology, Surat - India
Bachelor Thesis: Wireless Home Automation (using Zigbee)
Research Supervisor: Prof. Dr. Shweta Shah

Apr/2007
All India Senior School Certificate Examination
Rajhans Vidyalaya, Mumbai - India
Main Subjects: Computer Science (C++), Physics, Mathematics, Chemistry, and English

Work Experience and Training

Present – July/2014
Researcher (Wissenschaftlicher Mitarbeiter)
Chair of Real-time Systems
Technical University of Kaiserslautern, Germany

Apr/2014 – Sept/2012
Assistant Researcher (Hilfswissenschaftler)
Chair of Real-time Systems
Technical University of Kaiserslautern, Germany

Apr/2014 – Sept/2013
Assistant Researcher (Hilfswissenschaftler)
German Research Center for Artificial Intelligence (DFKI), Kaiserslautern - Germany

July/2012 – Aug/2011
German Language Training
Technical University of Kaiserslautern, Germany

June/2009 – May/2009
Innovative Microsystem Pvt. Ltd.
Internship; Mumbai - India

July/2008 – June/2008
Vertical Infinity Pvt. Ltd.
Internship; Mumbai - India

Teaching Expirience

- Lectures on system startup and bootloaders, virtualization and hypervisors, real-time resource management, and energy-aware real-time systems
- Exercises on bootloaders, real-time scheduling methods, and energy-aware real-time systems
  - Best exercise (Beste Übung) award by Fachschaft EIT (TU Kaiserslautern) in winter semester 2019/20
- Laboratory sessions on Uboot bootloader, PikeOS hypervisor (ARINC 653 and ELINOS), FreeRTOS, processors and threads, resource management techniques, and Linux driver development

Fachbereich Elektrotechnik und Informationstechnik
Technischen Universität Kaiserslautern

Postfach 3049
67653 Kaiserslautern
Germany

https://www.eit.uni-kl.de/