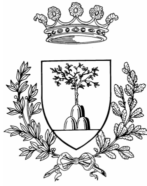


UNIVERSITÀ DEGLI STUDI DI FERRARA

DOTTORATO DI RICERCA IN MATEMATICA

CICLO XXX, COORDINATORE PROF. MASSIMILIANO MELLA

Predictable Communication Semantics For Industrial Real-Time Systems

Settore Scientifico Disciplinare MAT/08

Candidato

Dr. Ignacio Sañudo Olmedo

Tutore

Prof. Marko Bertogna

Anni 2014/2017

Abstract

With the increasing computational performance per Watt provided by multi-/many-core architectures, several industries are facing a transition from single-core to multi-/many-core systems. At the same time, there is a trend in the automotive market aiming at integrating multiple software components into the same MPSoC (MultiProcessor System-On-Chip). In this context, the partitioning and integration of mixed-criticality applications on top of multi-/many-core architectures is a serious challenge for embedded-software architects. In order to tackle this problem, a plausible solution is to virtualize the hardware resources. The adoption of this approach in the automotive industry has increased in recent years in order to achieve the temporal and spatial isolation demanded by Tier-1 and OEMs for the execution of mission-critical real-time applications. However, there are still many concerns and challenges regarding the correct management of shared hardware devices.

At software level, modern automotive embedded applications are composed of multiple components with different levels of criticality. In particular these software components communicate through shared memory. This type of inter-task communication can lead to potential data inconsistency issues. In this sense, distinct novel and standard communication models are proposed, providing different levels of predictability and data consistency.

This thesis focuses on investigating and analyzing problems related to the execution of real-time applications running on top of modern embedded multi-/many-core architectures. Specifically, this dissertation makes the following contributions: 1) identification of the main sources of unpredictability in multi-core architectures at I/O level in virtualized platforms, 2) analysis of different mechanisms that are used for shared-memory inter-task communication in the automotive domain 3) end-to-end latency characterization of effect chains implemented with different communication paradigms, and 4) a model-based code generator tool that serves to test and prove the findings done during the PhD.

© Università di Modena e Reggio Emilia

Abstract

Con l'aumento delle prestazioni computazionali per Watt fornito da architetture multi-/many-core, diverse industrie stanno affrontando una transizione da sistemi single-core a multi-/many-core. Allo stesso tempo, c'è una tendenza nel mercato automobilistico che punta all'integrazione di più componenti software nello stesso MPSoC (Multi-Processor System-On-Chip). In questo contesto il partizionamento e l'integrazione di applicazioni a criticità mista su architetture multi-/many-core rappresenta una seria sfida per gli architetti del software embedded. Per affrontare questo problema una soluzione plausibile è virtualizzare le risorse hardware. L'adozione di questo approccio nell'industria automobilistica è aumentata negli ultimi anni al fine di raggiungere l'isolamento temporale e spaziale richiesto da Tier-1 e OEM per l'esecuzione di applicazioni mission-critical in tempo reale. Tuttavia, ci sono ancora molte preoccupazioni e sfide riguardanti la corretta gestione dei dispositivi hardware condivisi.

A livello software, le moderne applicazioni embedded automobilistiche sono composte da più componenti con diversi livelli di criticità. In particolare questi componenti software comunicano attraverso la memoria condivisa. Questo tipo di comunicazione tra le attività può portare a potenziali problemi di incoerenza dei dati. In questo senso, vengono proposti modelli di comunicazione nuovi e standard distinti, che forniscono diversi livelli di prevedibilità e coerenza dei dati.

Questa tesi si concentra sull'analisi dei problemi relativi all'esecuzione di applicazioni in tempo reale, che si basano su architetture multi-core/many-core. Nello specifico questa tesi fornisce i seguenti contributi: 1) identificazione delle principali fonti di imprevedibilità nelle architetture multi-core a livello I/O in piattaforme virtualizzate; 2) analisi di diversi meccanismi che vengono utilizzati per la comunicazione inter-task a memoria condivisa nel dominio automobilistico; 3) caratterizzazione della end-to-end latency di effect chain implementate con diversi paradigmi di comunicazione e 4) uno strumento generatore di codice basato su modelli che serve per testare e dimostrare i risultati ottenuti durante il dottorato.

Acknowledgements

I would first like to express my most sincere gratitude to my advisor, Marko Bertogna for giving me support, motivation and guidance during my PhD. The passion and enthusiasm he has for work was contagious for me. I would also like to extend my gratitude to Lidia Diaz for the fruitful advices and for having welcomed me in Italy with remarkable kindness.

I am grateful to all my colleagues of the HipeRTLab with whom I had the pleasure to work during my academic journey (too many to name them all). I am especially grateful to Paolo Burgio for the professional guidance during these years, he taught me a lot of things and made research a lot easier for me. Also to Jorge Martinez for the useful exchange of ideas about the automotive industry over the last year.

A special thanks to all my friends in Spain and Italy who have stood by me in the good times and the bad, specially to Giovanni Dolza. I will remember your daily-joy and your cook-skills forever, I am sure that I have gained a friend for life.

Lastly, I would like to express my gratitude to my family who supported and encouraged for my great success. None of this would have been possible without you.

Table of Contents

Abstract **II**

Acknowledgements **III**

List of Tables **VI**

List of Figures **VII**

List of Acronyms **1**

1 Introduction **1**
1.1 Organization of the thesis 6

2 Background **7**
2.1 Functional safety 7
2.1.1 ISO-26262 9
2.2 AUTOSAR 11
2.2.1 AUTOSAR Software Development 13
2.2.2 AUTOSAR Execution Model 15
2.2.3 AUTOSAR Communication Model 18
2.3 Model-Driven Development 20
2.3.1 AMALTHEA 22

3 Towards predictability in virtualization platforms **25**
3.1 Introduction to Virtualization 26
3.2 Motivation 29
3.3 Virtualization and safety issues 32
3.4 I/O scheduling in virtualized environments 35
3.5 Multi-Core partitioning and virtualization 37
3.6 Perfomance and security issues introduced by I/O virtualization 38
3.7 Deadline-aware I/O scheduling 39
3.8 Real-time issues in SSDs 39
3.9 Summary 40

4 Towards predictability in AUTOSAR **42**
4.1 Introduction . . . 43
4.1.1 Data consistency and time determinism issues . . . 44
4.2 Related Work . . . 47
4.3 System model, terminology and notation . . . 49
4.3.1 Analysis for Preemptive Tasks . . . 52
4.3.2 Analysis for Cooperative Tasks . . . 53
4.4 Inter task communication . . . 55
4.4.1 Implicit communication . . . 56
4.4.2 LET communication . . . 58
4.5 End-To-End latency characterization . . . 63
4.5.1 Explicit Communication . . . 66
4.5.2 Implicit Communication . . . 68
4.5.3 LET Communication . . . 70
4.6 System analysis . . . 71
4.6.1 System and model constraints . . . 71
4.6.2 End-To-End latency analysis . . . 74
4.7 Summary . . . 78

5 Code generation support for automotive and general purpose platforms **80**
5.1 Motivation . . . 80
5.2 Related Work . . . 82
5.3 Hipert Generator Tool - HGT . . . 82
5.3.1 Front-end . . . 83
5.3.2 Core . . . 85
5.3.3 AMALTHEA mapping onto HGT . . . 87
5.3.4 Back-end . . . 90
5.4 Experimental evaluation . . . 91
5.4.1 Memory Phase . . . 91
5.4.2 Execution Phase . . . 92
5.5 Summary . . . 94

6 Conclusions **95**
6.1 Future Work . . . 96

Bibliography **99**

List of Tables

Table 2.1: ASIL risk matrix. 10

Table 4.1: *Find-Publishing-Point* Algorithm (left) *Find-Reading-Point* Algorithm (right) 71
Table 4.2: System memory usage 73
Table 4.3: End-to-End latency characterization of $EC1$ (top) and $EC2$ (bottom). . 76

Table 5.1: AMALTHEA and RT-DOT model mapping. 90

List of Figures

Figure 1.1: Global scheduling (left), Partitioned scheduling (middle), Virtualization (right). . . . 3
Figure 1.2: Logical Execution Time semantic. . . . 4

Figure 2.1: ISO-26262 structure. . . . 11
Figure 2.2: Old automotive paradigm & AUTOSAR paradigm. . . . 12
Figure 2.3: AUTOSAR architecture (a) BSW sublayers (b). . . . 13
Figure 2.4: AUTOSAR Virtual Function Bus. . . . 14
Figure 2.5: AUTOSAR RunTime Environment. . . . 15
Figure 2.6: AUTOSAR ECU deployment. . . . 16
Figure 2.7: Task state transition model [22]. . . . 17
Figure 2.8: AUTOSAR Communication Patterns. . . . 19
Figure 2.9: Code generation pipeline process. . . . 21
Figure 2.10: Structure of the AMALTHEA model. . . . 24

Figure 3.1: Physical and virtualization approaches. . . . 26
Figure 3.2: Hypervisor design, Type 1 (right) & Type 2 (left) . . . 27
Figure 3.3: Summary of I/O experiments in Xen. . . . 30
Figure 3.4: Freedom From Interference Example - ISO-26262. Definition 1.49. . . 33

Figure 4.1: Write and read conflicts. . . . 45
Figure 4.2: Buffer (Implicit) and lock mechanisms. . . . 45
Figure 4.3: End-to-end effect chains composed of three tasks with parameters $T_1 = 5, T_2 = 10, T_3 = 20$ and $C_1 = C_2 = C_3 = 1$. . . . 46
Figure 4.4: End-to-end effect chains composed of three tasks with parameters $T_1 = 5, C_1 = 3, T_2 = 10, C_2 = 2, T_3 = 20$ and $C_3 = 3$. . . . 47
Figure 4.5: End-to-end effect chain with LET composed of three tasks with parameters $T_1 = 5, T_2 = 10, T_3 = 20$ and $C_1 = C_2 = C_3 = 1$. . . . 48
Figure 4.6: End-to-end effect chain with LET composed of three tasks with parameters $T_1 = 3, T_2 = 5, T_3 = 6$ and $C_1 = C_2 = C_3 = 1$. . . . 48
Figure 4.7: Task communication example. . . . 51
Figure 4.8: Explicit communication example. . . . 55

Figure 4.9: Implicit communication implementation. 57
Figure 4.10: Implicit communication example. 57
Figure 4.11: Publishing and reading points when the reader has a larger (a) or smaller (b) period than the writer. 59
Figure 4.12: LET harmonic communication. 61
Figure 4.13: NHSC: $2T_R = 5T_R$. 62
Figure 4.14: NHSC: $5T_R = 2T_R$. 63
Figure 4.15: Age semantics 65
Figure 4.16: Reaction semantics 65
Figure 4.17: Calculation of ϕ_i^r 66
Figure 4.18: Worst-case sub-chain age latency $\alpha_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$ (a) and $\phi_R^j < \phi_W^i$ (b). 67
Figure 4.19: Worst case sub-chain reaction latency $\delta_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$ (a) and $\phi_R^j < \phi_W^i$ (b). 68
Figure 4.20: Worst-case sub-chain reaction latency ($\phi_R^0 < \phi_W^{last}$) for the Implicit communication. 69
Figure 4.21: End-To-End latency characterization of the LET communication. . . 70
Figure 4.22: Proposed *EC*s. 72
Figure 4.23: Hardware model with task distribution. 72
Figure 4.24: Distribution of labels on runnables/cores. 73
Figure 4.25: Task response times considering Explicit and Implicit/LET communication patterns. 74
Figure 4.26: Fig. 13. Normalized End-To-End latencies: Age latency (a) Reaction latency (b). 76
Figure 4.27: Memory footprint considering the communication patterns. 78

Figure 5.1: HGT framework organization. 83
Figure 5.2: Example of RT-DOT and AMALTHEA files and models. 84
Figure 5.3: AMALTHEA task with 4 runnables. 85
Figure 5.4: Example of code generation process from AMALTHEA model. . . . 87
Figure 5.5: AMALTHEA communication models. 89
Figure 5.6: Memory performance test. 91
Figure 5.7: Task execution model. 93
Figure 5.8: Computation performance test. 94

1 Introduction

Real-time systems are computer-based systems that must react within tight time constraints to critical events in the external world [1]. Due to their specificity and complexity, these systems have been traditionally designed and implemented on top of a single computing core, especially in the embedded and industrial domain. In this context, modern industrial applications need to process a huge amount of data coming from multiple sensors with tight timing restrictions, thus requiring an increasing and considerable amount of computational power. To satisfy this increasing computational demand in the embedded domain, at first, silicon vendors invested in exploiting Moore's law/Dennard's scaling by increasing the processor clock speed and by substantially increasing the Instruction-Level parallelism (ILP). However, this approach led to power consumption and power dissipation issues, that ultimately make it impossible to power up even more than 75% of transistors in a single SoC (System On Chip) simultaneously. This phenomenon is called *Dark Silicon* [2]. For this reason, architectures based on a single-core processor reached a frequency ceiling limit more than a decade ago. As a consequence, vendors decided to take advantage of Thread-Level Parallelism (TLP) by integrating multiple processors onto the same die, delivering high computing performance while meeting size, weight and power (SWaP) constraints that are imposed in the embedded domain. The appearance of parallel programming models like OpenMP [1] or Cuda [2] that exploit the parallel nature of these new architectures, help programmers in addressing the considerable computational demand required in new-generation applications by increasing their productivity and making the exploitation of the hardware straightforward.

At the same time, the technological trend in embeddded real-time systems lead towards the integration of multiple applications with different criticality levels onto the same computing platform, thus reducing considerably production costs. Accordingly, many manufacturers in the automotive, avionic or even medical domain are moving towards multi-core platfoms for the integration of mixed-criticality systems. Nevertheless, the increasing complexity of today's multi-core architectures that have been designed primarily for general purposes, combined with the need to integrate applications with

[1] http://www.openmp.org/
[2] https://www.geforce.com/hardware/technology/cuda

real-time constraints, challenge the design of reliable and efficient solutions to unprecedented problems. In this sense, a common research problem in the multi-core domain is related to the concurrent access to shared resources. For instance, the concurrent access to the memory is one of the main sources of unpredictability in program execution.

Concurrently, the single-core to multi-core transition does not only bring a change of programming/design methodologies, but also in functional safety issues. A typical example of functional safety in the computer science domain is the Ariane 5 accident, where a rocket exploded during a test flight. The error was because of a floating point data conversion due to the fact that Ariane 5 contained part of the Ariane 4 code. While Ariane 5 worked with 64-bit floating points values, the old software required 16-bit signed integer. This example shows the importance of considering a safety assessment process in the architecture transition phase. Traditionally, many of the mechanisms and safety measures used to mitigate potential risks have been developed and certified for single-core platforms. Consequently, multi-core systems have to be implemented considering a number of unprecedented problems such as managing access to shared resources or fault propagation problems.

Achieving predictability by means of virtualization. Nowadays, embedded system providers are increasingly implementing software solutions on top of multi-core processors. Applications running on multi-core platforms can be allocated according to different approaches. Historically, classic real-time systems are scheduled with partitioned or global scheduling. Unfortunately, those approaches have significant limitations for the integration and implementation of mixed-critical software with real-time requirements. Recently, virtualization has been gaining popularity for the implementation of software with mixed-critical requirements. Therefore, based on the performance and predictability provided, we can classify multi-core task allocation in the above mentioned categories (see Figure 1.1):

- Global scheduling: According to the global scheduling approach [3], tasks can be scheduled in any processor, such that, tasks can migrate between cores, this can lead to increase the task execution times due to cache misses. For instance, delays related to moving tasks between cores with a different on-board cache. Generally, global scheduling provides lower average response times achieving higher utilization bounds with respect the partitioned approach because the workload can be better distributed over the multiple cores.
- Partitioned scheduling: While global scheduling allows to schedule dynamically tasks in all cores, according to the partitioned scheduling approach, tasks are statically assigned to cores. Thus it is possible to schedule tasks using state of the

art single-core scheduling algorithms, being possible to extrapolate single-core analysis. However, it is still an open problem in the community the optimal task-to-core allocation (task partitioning), that is essentially a bin-packing problem, which is known to be a combinatorial NP-hard problem.

- Virtualization: Virtualization is a technology that provides mechanisms for the emulation of multiple systems running on a single hardware. A hypervisor or virtual machine monitor (VMM) is a layer of abstraction that enables the execution of different operating systems in the same on-board platform. In this regard, virtualization allows the hard isolation of applications in virtual machines providing an environment that enables the execution of mixed-criticality applications in shared hardware.

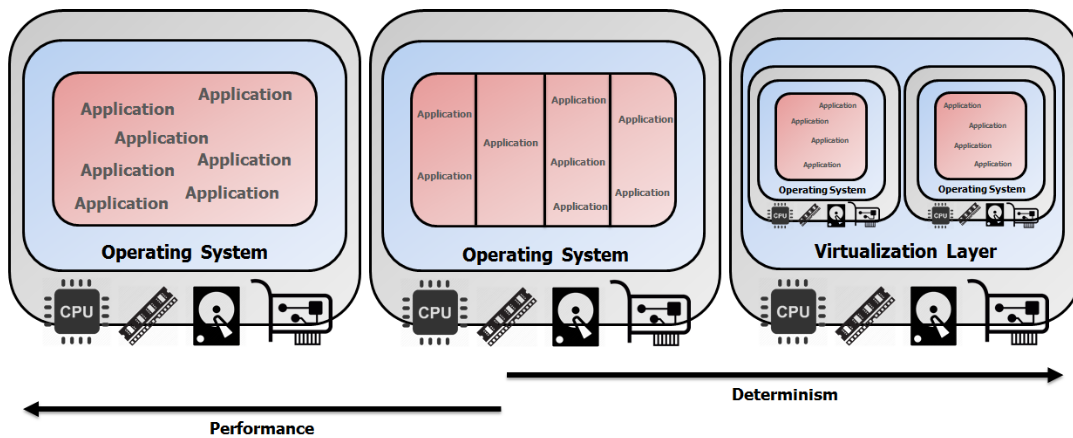


Figure 1.1: Global scheduling (left), Partitioned scheduling (middle), Virtualization (right).

Many software providers in the marketplace are moving towards the use of virtualization as a solution to execute complex mission critical applications and infotainment software within the same computing platform. This kind of "coexistence" brings problems in the interaction and management of the shared devices. Following this assumption, the main requirement imposed by many industrial standards is to achieve the so called *Freedom From Interference (FFI)* between partitions. Many software standards in the automotive (ISO-26262 [4] or Adaptive AUTOSAR in the near future [3]) and avionic domain (ARINC-653 [5], DO-178C [6]) define guidelines to evaluate and support systems executing applications with different criticality levels. Although those standards describe how to develop applications complying with software safety assessments, some

[3]https://www.autosar.org/standards/adaptive-platform/

of the safety recommendations in terms of software implementation are very ambiguous because they do not provide details on how to achieve spatial and temporal isolation between mixed-critical partitions.

Predictable Execution Models. While at hardware level virtualization provides mechanisms to isolate the hardware resources, at software level guaranteeing a predictable behavior in software components is still challenging. Although it is obvious, not only virtualization can provide "real-time" properties to applications. In the real-time context, many of the new innovations are driven by software. Accordingly, different task models have been proposed in the literature aiming to provide a predictable behavior. For instance, the AER (Acquisition, Execution, Restitution) [7] execution model, where each task is enforced to execute in its time slot, PREM (PRedictable Execution Model) [8] that decouples the task execution in different phases or even the *Logical Execution Time (LET)* model. LET [9] is a hard real-time programming abstraction that was introduced with the time triggered programming language Giotto [10] to provide determinism in the execution. In a nutshell, LET semantic fixes the time it takes from reading a particular input to writing program output, disregarding the temporal behavior of the application. In this way it is possible to achieve a high level of predictability and a strong consistency between the timing constraints (logical model) and the task execution (physical model), thus facilitating the design, implementation, test and certification proccess [11] (see Figure 1.2).

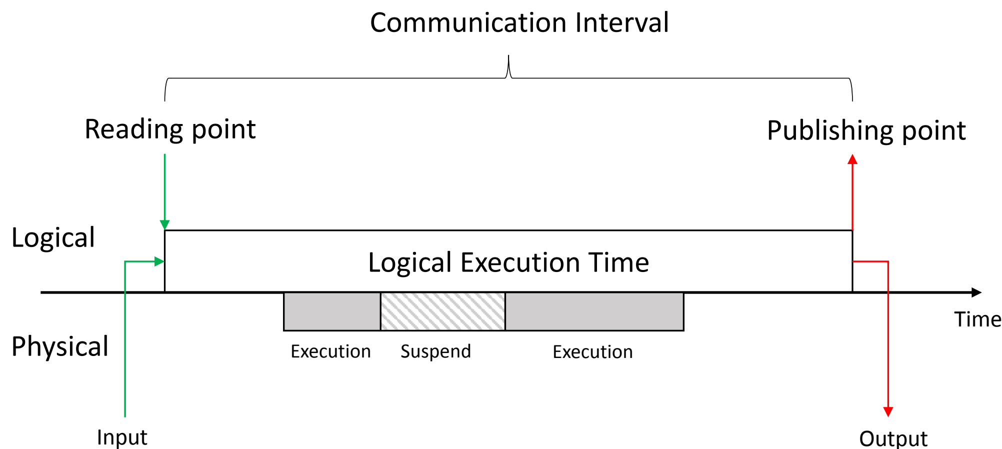


Figure 1.2: Logical Execution Time semantic.

In the automotive domain, automotive engineers are concerned with the application responsiveness. In particular, the propagation delay of an input stimulus, that triggers a chain reaction leading to a final actuation or control action, called *Effect Chain*. A parallel

concern arises in the integration of applications running on multi-core platforms, these applications usually communicate by means of shared memory. In a nutshell, this kind of communication can lead to possible data inconsistency issues due to concurrent access to shared data. Generally, concurrent access to shared resources need to be synchronized to avoid conflicts or data inconsistency problems between tasks that shared data. One possible solution is to limit the access using lock-based resource sharing protocols. Lock-based mechanisms enforce the exclusive mutual access to shared resources, consequently incurring in task high-blocking delays. In the automotive domain, different communication patterns are used for inter-task communication ensuring different levels of data consistency, namely *Explicit* and *Implicit* communication patterns, both approaches are presented in the AUTOSAR standard.

- *Explicit*: Reads and writes are performed directly in the memory disregarding potential data inconsistency issues. In order to avoid this pitfall, lock-based constructs are used.
- *Implicit*: In this case, tasks accessing shared labels work on local copies instead of the original labels. Specifically, data consistency is achieved prefetching all the shared variables into the local memory in the beginning of the task execution and publishing it at the end. In order to do so, lock-based, lock-free or wait-free mechanisms can be used.

Those approaches have diverse impact on the propagation delay of communication chains and also in terms of memory footprint. Logical Execution Time appears as a natural model for task execution that provides determinism and data consistency using wait-free mechanisms. Indeed, there is an increasing interest in LET in the automotive domain [12]. Unfortunately, LET is not currently supported by AUTOSAR and there are not many studies of LET and other communication patterns in the automotive context.

1.1 Organization of the thesis

This dissertation is organized as follows: Chapter 2 introduces the necessary background to understand and motivate the findings provided in this work. In particular, we introduce notions in functional safety and ISO-26262, giving a little perspective of the recommendations and requirements in the development of automotive software. In the same way, AUTOSAR and the RunTime Environment (RTE) are described. Then we introduce the principle of model based development, where a modeling framework used in the automotive domain called AMALTHEA is presented. Chapter 3 presents a survey on I/O management within virtualized platforms, providing a view of the limitations given in these technologies in terms of real-time performance at I/O level. How to overcome these limitations is an important aspect of the ISO-26262. Motivated by the need to explore isolation mechanisms for providing shared resource access management by one or more tasks, in Chapter 4 we present different novel and standard mechanisms for providing determinism and data consistency in inter-task communication at software level. Specifically, we propose a schedulability analysis for the AUTOSAR task model in which cooperative and preemptive tasks are concurrently scheduled on the same platform. Moreover, we present an end-to-end latency analysis and an implementation for three different inter-task communication patterns, namely, Explicit communication pattern, Implicit Communication pattern and Logical Execution Time. Theoretical results are derived using a widely adopted benchmark for automotive real-time systems. We derive this analysis in the context of the 'Formal Methods for Timing Verification' FMTV industrial challenge organized by Robert Bosch GmbH. The techniques exposed in this chapter can be used to solve shared resource management pitfalls. In Chapter 5, the Hipert Generator Tool (HGT) is presented. The code generator tool allows the generation of synthetic tasks from different modeling frameworks (RT-DOT and AMALTHEA) following a model based development approach. The tool was developed explicitly, and used to validate research outcomes in real industrial settings even when application details and code are unknown because they are covered by NDA (Non-Disclosure Agreement) or IPR (Intellectual Property Right). Finally, in Chapter 6, future work and the major findings of the thesis are summarized.

2 Background

In this chapter we summarize the background and provide a context for the concepts presented in this dissertation. In particular, Section 2.1 reviews the basic idea behind functional safety and software development for Electronic and Electrical (E/E) components according to ISO-26262, an international standard for the automotive industry. Since Chapter 4 is dedicated to derive an End-To-End latency analysis of automotive communication patterns, Section 2.2 presents an overview of software development following the AUTOSAR standard, paying special attention to the software communication mechanisms. Finally, Section 2.3 introduces notions of Model-Driven Development in order to understand how the code-generator tool presented in Chapter 5 works. Moreover, the AMALTHEA framework is presented, that in this dissertation is used to: abstract the functionality of automotive applications, compute the aforementioned End-To-End latency analysis, and generate the code of the corresponding application.

2.1 Functional safety

Lately, the automotive industry is facing a change in the way they build our vehicles. New passenger vehicles require a technological transition to satisfy the computational demand by new-generation automotive software, opening up a number of opportunities for innovation and research. All the big players in the automotive domain are spending a considerable amount of resources in this direction. Major OEMs (Original Equipment Manufacturer) like BMW, Volvo, Tesla, or General Motors and Tier-1s such as Bosch or Continental, are already developing the necessary know-how and technological background to build the next generation of embedded automotive systems.

According to Semicast studies [1], revenues for advanced driver-assistance systems (ADAS) electronics are estimated to grow to around 86$ billion in 2022, from 53$ billion in 2015. According to this trend, the complexity of the automotive software is growing very fast. Nowadays, a car manages up to 2500 signals [13]. Among these signals we find, for instance, information related to car speed, break control or just button-action requests that, for example, roll up a car window or unlock a car door. All these signals are exchanged through the car network and computed by an embedded system called

[1] http://semicast.net/wp-content/uploads/2016/09/120916.doc Last access 1/11/2017

Electronic Control Unit (ECU). Currently, a single car contains approximately from 50 to 100 ECUs. Each ECU manages a specific functionality of the car, for example, the combustion engine or the electronic valve control. In particular, the car's software contains close to 100 million of lines of code (LoC). It should be noted that, estimates indicate a growth of 150-300 millions of LoC in the near future [2]. To get a rough idea of this quantity, the mouse DNA can be written in around 120 millon of LoC [3].

In this line, safety is one of the key aspects of automobile development. Safety is the absence of unreasonable risk. According to IEC61508 [14], functional safety is: "Freedom from unacceptable risk of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment." Following this assumption, the idea behind functional safety is reducing risk in electronic systems. This concept is important also in many other domains, such as automotive, avionics, railroad or medical device.

In the automotive domain, while the development of new generation hardware and software components is growing very fast, not all the OEM and Tier-1 players are creating prototypes meeting the constraints required by the standards, potentially creating unsafe products. Probably the best-known case so far, is the Toyota unintended acceleration problem. According to the National Highway Traffic Safety Administration (NHTSA), unintended acceleration (UA) is "the occurrence of any degree of acceleration that the vehicle driver did not purposely cause to occur". In 2009, Toyota recalled several vehicles experiencing different unintended acceleration problems. As reported by NHTSA [15] the fault was related to pedal entrapment. As a consequence of this episode, there were reported 89 people dead. Moreover, the economical consequences for Toyota were significant: they spent approximately 1200M$ in legal costs for violating safety laws and more than 10 million vehicles were recalled worldwide. NASA was appointed to investigate the case. The report presented that only 11 of 127 rules of *MISRA C* were complied. MISRA (Motor Industry Software Reliability Association) is a consortium that supplies guidelines for the software development of E/E components, this guideline is adopted in the automotive field. MISRA C [16] is a guideline for the C programming language, originally conceived for the safety development of automotive embedded systems, currently it is used in many domains such as, defense, railway aerospace, telecommunications, among others. With regard to the Toyota unintended acceleration problem, experts in functional safety pointed that if they had followed the directives described in the automotive standards, the failure would have been mitigated. Therefore, it is easy to conclude that functional safety plays an important role in an area in which a system failure may causes physical injuries or even loss of life.

[2] http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code
[3] http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

There are many standards that contribute to the safety development of electronic and software components:

- IEC-61508 [14] (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems) specifies a complete safety life cycle for electronic components of different domains.
- ISO-26262 [4] standard, defines methodologies for the development and deployment of an electrical and/or electronic systems within a road vehicle. The standard is an adaptation of IEC-61508 for automotive systems.
- AUTOSAR (AUTomotive Open System ARchitecture) is an open standard for automotive system development whose main goals are: software independence from hardware, modularity, scalability, reusability of functions, and flexible maintenance. AUTOSAR looks at the different functionalities in a car network, splits them into logical clusters (Software Compositions), and finds functional atomic units (Software Components) that make up these clusters.

2.1.1 ISO-26262

ISO-26262 published in 2008, is the main functional safety standard for the development of electronic systems in passenger vehicles. The standard works as a guidance to avoid risks due to hazards caused by malfunctioning behavior of the vehicle. The content of the standard is not legally required, but it provides a methodical way for ensuring safe development of E/E components for vehicles. The standard defines a safety life cycle for E/E products that covers the entire life-cycle of an automotive component, from management and development to production, decommissioning and relation with suppliers. Moreover, it provides a V-model as a reference process model for the diverse phases of the product development.

The standard characterizes the importance to reduce a risk using a safety measure called *Automotive Safety Integrity Level (ASIL)*, that is the counterpart to the SIL, defined in IEC-61508. ASIL, is composed of five different levels, each level specifies the safety measures to apply for avoiding an unreasonable residual risk. These levels are ASIL D, ASIL C, ASIL B, ASIL A and QM, where ASIL D determines the maximum stringent level and QM (Quality Management) the least stringent level where there is no need to apply ISO-26262, i.e., a component without any safety requirement. The higher the ASIL level, the greater the importance to reduce the risk. In fact, ASIL D components require a hardware failure rate of $\leq 10^{-8}h^{-1}$ whereas ASIL B components have to comply with a failure rate of $\leq 10^{-7}h^{-1}$.

Table 2.1: ASIL risk matrix.

Controllability	Exposure	Severity			
		S0	S1	S2	S3
C1	E1	QM	QM	QM	QM
	E2	QM	QM	QM	QM
	E3	QM	QM	QM	A
	E4	QM	QM	A	B
C2	E1	QM	QM	QM	QM
	E2	QM	QM	QM	A
	E3	QM	QM	A	B
	E4	QM	A	B	C
C3	E1	QM	QM	QM	A
	E2	QM	QM	A	B
	E3	QM	A	B	C
	E4	QM	B	C	D

The ASIL level is evaluated based on a hazard and risk analysis. The analysis identifies and categorizes hazardous events to the prevention or mitigation of the associated hazards to avoid unreasonable risk. The analysis establishes the definition/allocation of functional and technical requirements into hardware and software components respectively. Accordingly, the criterion used to determine the ASIL level of an E/E automotive component is based on the following criteria:

1. Controllability: ability to avoid a specified harm or damage. The controllability is estimated based on a defined rationale for each hazardous event that can be caused by the driver or other persons potentially at risk. The controllability is designated to one of the following controllability classes: *C1 - Simply controllable, C2 - Normally controllable , C3 - Difficult to control or uncontrollable*, in accordance with Table 2.1.

2. Probability of exposure: likelihood of the occurrence of harm or malfunction. The exposure classes are: *E0 - Incredible, E1 - Very low probability, E2 - Low probability, E3 - Medium probability, E4 - High probability.*

3. Severity: measure or estimate of the extent of harm to the persons involved in a possible accident. The severity classes are: *SO - No injuries, fatal injuries, S1 - Light and moderate injuries , S2 - Severe and life-threatening injuries (survival probable), S3 - Life-threatening injuries (survival uncertain).*

In order to identify the possible hazards in the components, techniques such as brainstorming, checklists, quality history, failure mode and effect analysis (FMEA) and field studies can be used. In this work we mainly cover Part 6 of the ISO-26262 standard

(See Figure 2.1), i.e., "Product development at Software level". Part of dissertation is dedicated to unveil the details of the standard for the use of an hypervisor as a technology for develop mixed-criticality systems. In this context, one key concept established by the ISO-26262 standard, is the so called, *Freedom From interference (FFI)*. FFI is the absence of cascading failure between two or more components. In particular, Section 3.3 describes deeply the importance of Freedom From Interference for the execution of software with different criticality levels.

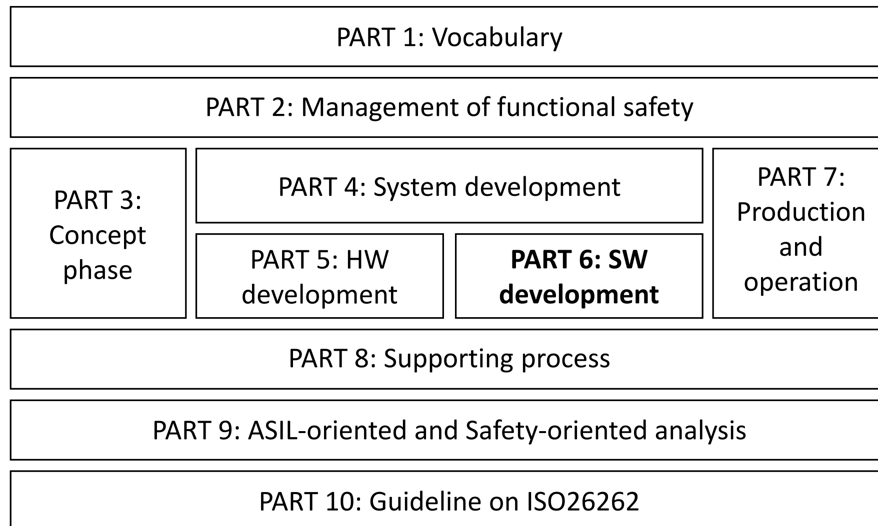


Figure 2.1: ISO-26262 structure.

2.2 AUTOSAR

A little over a decade ago, car software manufacturers used to integrate their own software solutions based on specific properties of the vehicle model. For instance, let us consider the software development of the power window. In this case, during the development phase, the Tier-1 had to consider several technical details of third-parties components, from the supplier of the door checker mechanism to the implementation of the power window motor. Such an approach made software suppliers dependent on the vehicle model, discouraging the reusability and scalability of the software components. Furthermore, the increasing complexity of E/E car components, the growth in software functionalities and the exponential complexity in car interconnections by means of ECUs, introduced many problems in terms of project management. All these factors led to Tier-1 and OEMs to establish the *AUTOSAR* partnership.

According to the standard [17], AUTOSAR (AUTomotive Open System ARchitecture) is a "worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry". The AUTOSAR consortium was constituted in 2003 by Continental, Bosch, BMW, Daimler, Volkswagen, Chrysler and Siemens in order to assemble the main principles for E/E vehicle software development [4]. AUTOSAR provides a standardization of the interfaces between the different software layers, giving a hierarchical organization of the SW/HW components integrated in the vehicle. As is depicted in Figure 2.2, this approach provides a powerful environment for reusability and scalability of SW/HW components, giving suppliers the chance to integrate the same software into different ECUs, reducing development times and minimizing maintenance efforts.

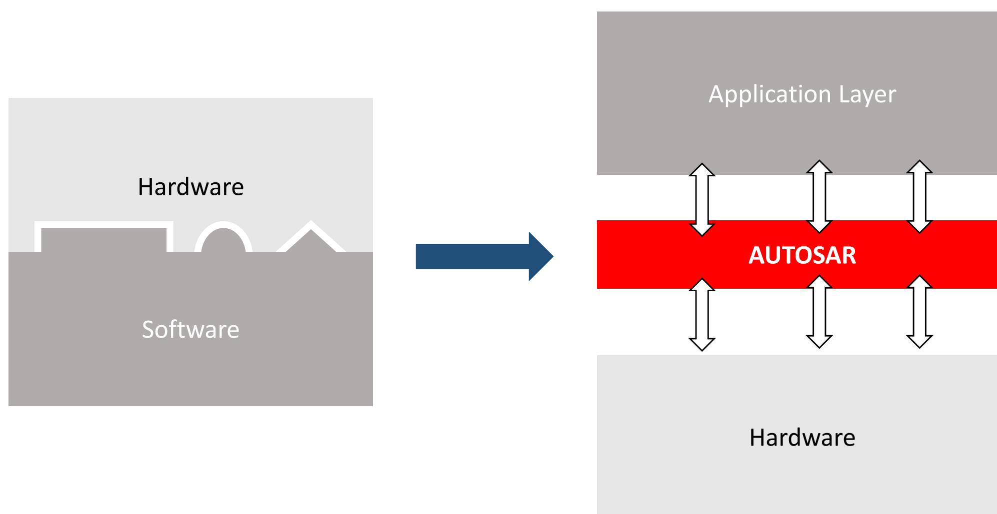


Figure 2.2: Old automotive paradigm & AUTOSAR paradigm.

The main principle of AUTOSAR is the division between *Application* and *Infrastructure*. To guarantee this principle, AUTOSAR defines three different software layers: (i) Basic Software (BSW), (ii) Application Software (ASW), and (iii) RunTime Environment (RTE) as detailed in Figure 2.3a:

- *Basic Software (BSW)*: is the standardized software layer that provides the infrastructural functionality for the ECU. The basic software defines through code and description files its functionality, in this way, it is possible to configure the different sub-layers grouped into the Basic Software. The BSW is composed of the following sub-layers (see Figure 2.3b): the Microcontroller Abstraction layer (MCAL) which provides hardware drivers making upper software layers independent from the

[4]Many other partners joined after the creation, like Toyota, Ford or Peugeot.

microcontroller; the ECU abstraction layer which provides APIs to access peripherals making upper software layers independent from the ECU hardware layout; and the Service Layer that provides operating system functionalities, memory services, diagnostic services, etc. Drivers that are not specified in AUTOSAR are to be found in the Complex Drivers layer.

- *Application Layer (ASW)*: This layer represents the implementation of the so called, *Software Components (SWCs)* and contains the functionality of the system. An obvious property of the SWCs is that they communicate with each other. According to the hierarchy established by the standard, the communication is done through the RunTime Environment (RTE).
- *RunTime Environment (RTE)*: The RunTime Environment provides a middleware to control the runtime behavior of the application layer, supporting the communication of the software components and the basic software at inter- and intra-ECU level. This layer makes SWCs independent from the mapping to a specific ECU.

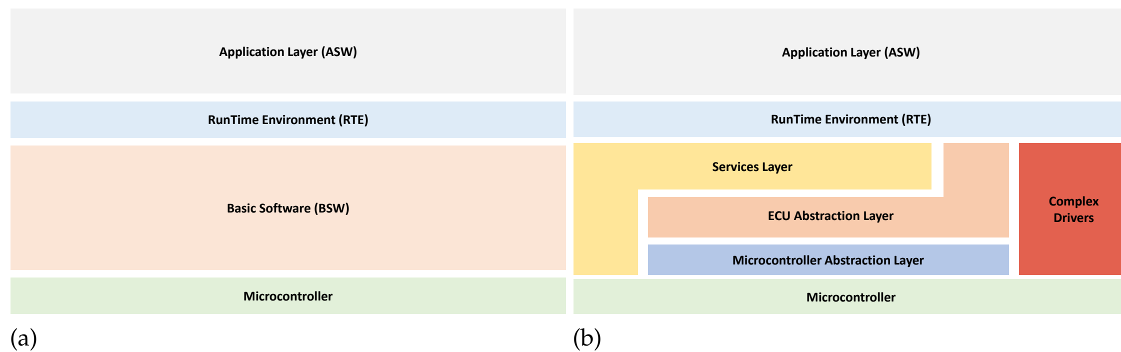


Figure 2.3: AUTOSAR architecture (a) BSW sublayers (b).

With regard to software scalability and portability, AUTOSAR provides two architectural mechanisms that facilitates independence of the software development. These mechanisms are called *Virtual Function Bus (VFB)* and the above-mentioned RunTime Environment. The VFB and RTE offer an abstract and concrete (respectively) layers for the exchange of information between software components. In the next sub-section we will introduce both concepts.

2.2.1 AUTOSAR Software Development

According to the standard: "Application software within AUTOSAR is organized in self-contained units called Software Components or, more formally, SwComponentTypes". In brief, SWCs contain the functionality of the application/system. The inherent

complexity of communication and interconnections by means of SWCs lead to model and design the communication in a very abstract form. SWCs are developed independently from the underlying hardware, thanks to the software abstraction layer provided by the Virtual Function Bus and the RunTime Environment. The VFB [18] is a virtual bus that models the communication among AUTOSAR SWCs and BSW modules. All SWCs are interconnected via the VFB. As is shown in Figure 2.4, the VFB provides a communication layer in which we can abstract communication details during design phase, strictly separating the Application Layer (i.e., SWCs) from the Infrastructure (i.e., BSW). Such an approach not only simplifies the communication between SWCs, but also provides a robust separation between Application and Infrastructure, thus allowing the reuse of the SWCs for different ECUs.

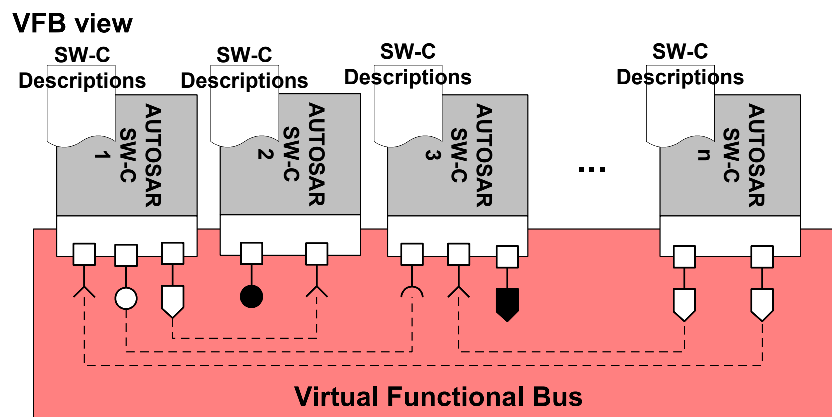


Figure 2.4: AUTOSAR Virtual Function Bus.

Specifically, software components communicate among each other and/or with the Basic Software modules through the RTE. While the VFB provides an abstraction that disregards the SWC-to-ECU mapping, the RTE represents the concrete implementation of the VFB (see Figure 2.5). In this context, the generation of the RTE for each ECU and the SWC-to-ECU allocation is performed by the so called RTE generator. The RTE generation proccess is divided into two different phases: *Contract Phase* and *Generation Phase* [19].

Accordingly, AUTOSAR offers an interface for the management of the SWCs called *Software Component Description*. In particular, this XML file provides information about the number of ports for the communication performed by the SWCs, a description of the runnable entities or scheduling information. This information will be used to generate the RTE interface that will contain functional aspects of the SWCs, working as a "contract" between the SWCs and the RTE. The contract phase does not create the code of the tasks, this is done by the definition of the functional behavior using tools like

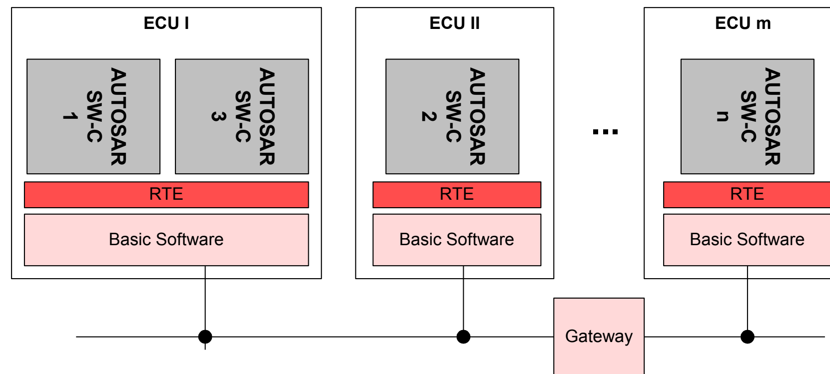


Figure 2.5: AUTOSAR RunTime Environment.

Simulink or just hand coding, but it creates the header files that contains the functions, data types and structures that will be used by the SWCs. For instance [13], let us suppose that we have a SWC with a send-port P that uses a data type D, the contract phase will generate the API function $RTE_Send_P_D$ that sends data D through the port P.

On the other hand, during the Generation phase, different documents are provided, for instance, ECU configuration parameters that will describe the mapping of the runnable to the tasks or the communication matrix. Summing up everything, the header files created during the contract phase are used to generate the concrete implementation of the RTE that will be executed in the ECU. Figure 2.6 depicts the process of the SWC tool deployment.

2.2.2 AUTOSAR Execution Model

The AUTOSAR execution model can be described in many forms. Generally, software engineers in the automotive domain use graph-based models to represent the task execution. In the literature, the notation shown to characterize the AUTOSAR execution model is based on *Directed Graph (D-graph)* or *Directed Acyclic Graph (DAG)* representation [20], [21]. In this dissertation, we proposed a DAG based approach to describe the execution model of the software components. In this sense, the proposed model is “compliant” with the AUTOSAR execution model. We will further explain the task model defined in Section 4.3.

SWCs are used to build the functionality of the system in a vehicle. In particular, the internal behavior of a SWC or application is characterized by tasks. According to AUTOSAR “a Task is the object which executes (user) code and which is managed by the OS”. A single SWC is composed of tasks and at the same time, each task is composed of different runnables. Runnable is the smallest functional entity in the automotive domain. A single runnable entity is mapped to a single task.

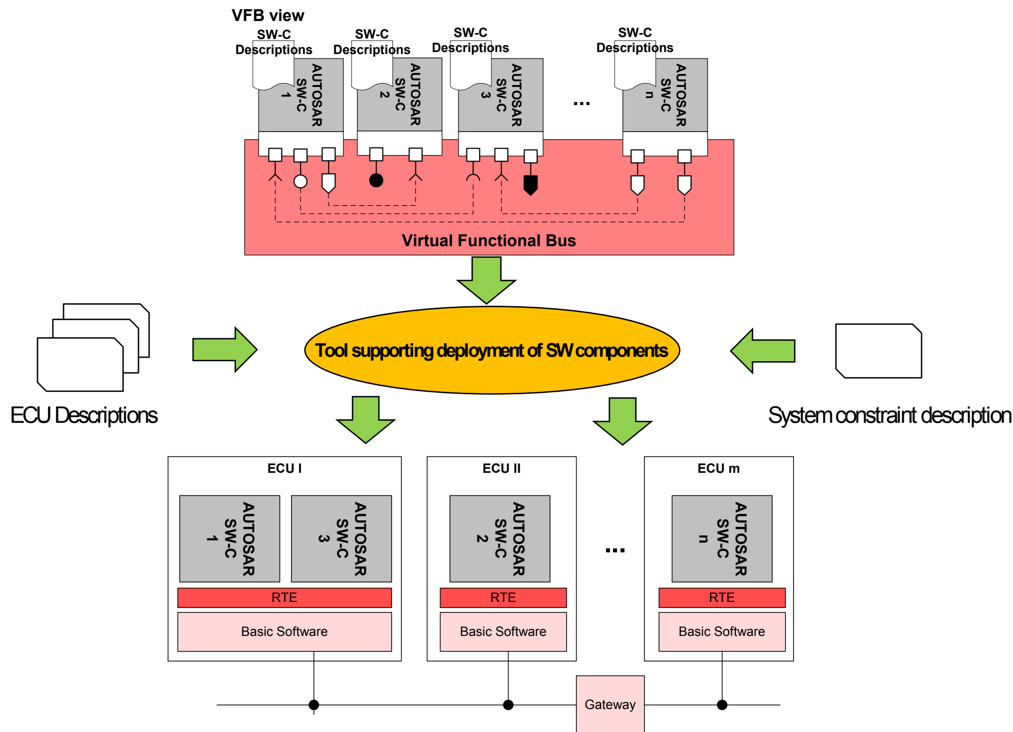


Figure 2.6: AUTOSAR ECU deployment.

In a multi-core platform different runnables can execute in different cores simultaneously. Typically, a task is the smallest schedulable entity, in this way, the operating system scheduler will decide in runtime which task will be scheduled according to the task priority and the scheduling policy. Many of the details provided in the AUTOSAR standard at operating system level, are based on the OSEK [5] OS (ISO 17356-3) standard. For instance, OSEK provides the definition of technical concepts like tasks, shared resource management or alarms and counters. AUTOSAR OS defines two different types of tasks, *Basic Tasks* and *Extended Tasks*. Without going into any further details, a basic task has three states (running, ready and suspended) and can not block itself. On the other hand, an extended task can suspend itself in order to wait for events, adding a Wait state in the flow state-machine design. The specification of the state-machine with the state transitions for both basic and extended tasks is depicted in Figure 2.7a.

Generally, tasks are scheduled according to partitioned fixed-priority scheduling. However, the priority can be change in runtime, it depends on the AUTOSAR OS. AUTOSAR operating systems supports three different scheduling algorithms, preemptive, non-preemptive, and cooperative scheduling [23]:

[5]www.osek-vdx.org

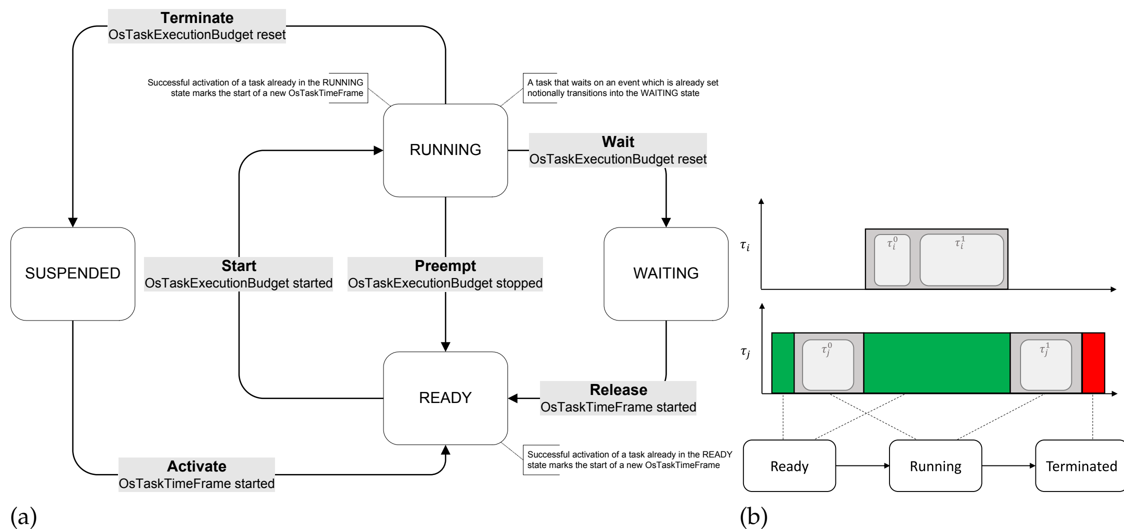


Figure 2.7: Task state transition model [22].

- Preemptive scheduling: a ready high priority task can preempt a lower priority one at any time during the execution. Although this approach provides very short response time and minimal jitter for highest priority tasks, under this scheduling algorithm, context switch overheads and data inconsistency issues need to be considered.
- Non-Preemptive scheduling: tasks can not be preempted. Scheduling decisions are taken at the end of the task execution. Following this assumption, response times are predictable for high priority tasks, however, non-preemptive tasks can incur to priority inversion problems.
- Cooperative or mixed preemptive scheduling: higher priority cooperative tasks can preempt lower priority cooperative tasks at predefined scheduling points or at the end of the execution. On the other hand, cooperative tasks can be preempted by higher priority preemptive tasks at any point in the execution. This approach allows to manage the task execution in a very flexible way.

To sum up, task and runnable entities run on top of the ECUs and are mapped onto SWCs thanks to the RTE.

The standard does not provide a task description based on activation requirements. However, strongly inspired in [24] we can classify tasks according to activation constraints:

- *Periodic and sporadic Tasks*: Periodic tasks are an infinite sequence of jobs, that are activated with a determined rate. On the other hand, sporadic tasks are characterized by a minimum interarrival time between consecutive jobs.

- *Single Activation Tasks*: In this case, these tasks are activated only once in the system, these tasks can either boot up/initialization or shutdown tasks.
- *Angle Synchronous Tasks*: These tasks are activated at prefixed rotation angles of the crankshaft. Following this assumption, the task rate depends on the engine speed, the higher the speed rotation, the higher the activation rate. This kind of tasks are also called angular or adaptive rate tasks [25].
- *Chained Tasks*: While the tasks mentioned-above do not have any precedence constraint in the activation, chained tasks are activated by a predecessor task. This activation can be triggered at intra- and inter-core level.
- *Interrupt Service Routines (ISR)*: An ISR is a process triggered by an interrupt request, it is triggered from a hardware device that manages a specific function of the associated device.
- *Modes and states*: Modes can be used to model the state of an ECU, i.e., a particular mode can trigger a task in a specific state. For instance, changing the mode of the ECU can trigger a task to prepare some resource before changing the state.

Many other properties of the classic real-time scheduling are defined in the AUTOSAR standard, like shared resources management policies or alarms and scheduling tables. However they are out of the scope of this dissertation.

2.2.3 AUTOSAR Communication Model

Physically, tasks communicate with each other through different communication interfaces, for instance, CAN [6], FlexRay [7], LIN [8] or Most [9]. In AUTOSAR, at design point of view, the communication between SWCs is realized through *Ports* and *Interfaces*. While the interface represents the structures and operations that are provided by a specific SWC through the RTE, a port is the software communication mechanism used to interconnect SWCs. We can distinguish between two kind of ports: *P-Port (Provider-Port)* and *R-Port (Receiver-Port)*. Ports and interfaces are interconnected, characterizing the structures and operations required or provided by a SWC through that port. AUTOSAR supports two kinds of communication Port-Interface:

- *Client-Server interface*: according to this communication interface, client components request server functions and server components provide the result to the

[6] http://www.bosch-semiconductors.de/en/automotive_electronics/ip_modules/can_literature_2.html
[7] http://www.mostcooperation.com/technology/most-network/
[8] https://elearning.vector.com/vl_lin_introduction_de.html
[9] http://www.mostcooperation.com

client. Following this assumption, Client-Server interfaces define the operations provided by the server that can be used by the client.

- *Sender-Receiver interface*: is the communication interface used for data exchange between SWCs. The interface defines the data type that will be exchanged from the sender to the receiver. In a nutshell, is like read/write variables between process, such as the C programming language.

Both communication interfaces are depicted in Figure 2.8. According to AUTOSAR, we

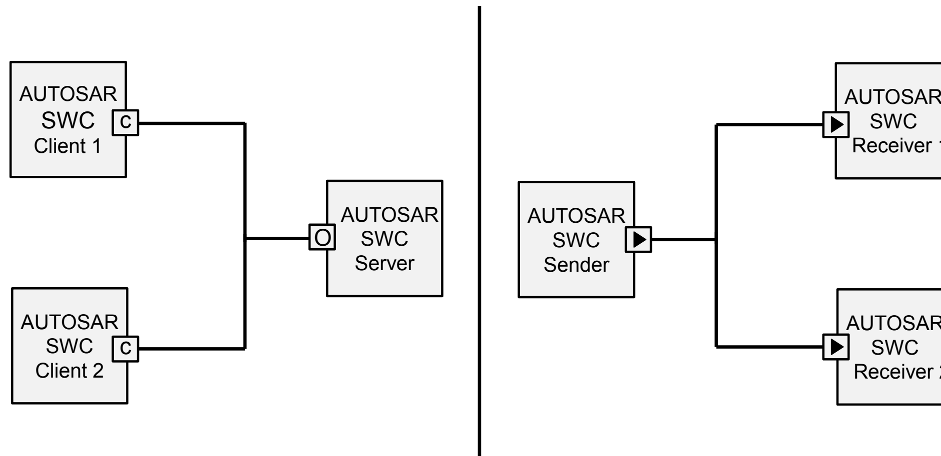


Figure 2.8: AUTOSAR Communication Patterns.

can further differentiate between two types of ECU communication:

- Intra-ECU communication: the exchange of information is realized among SWCs that are located in the same ECU.
- Inter-ECU communication: in this case, the communication is performed between SWCs that are located in different ECUs.

SWCs communicate through the RTE, but at implementation level, runnables communicate by means of variables, in AUTOSAR called inter-runnable variables. From now on we will call them shared labels. Data exchange between SWCs take place when runnables write or read in a variable. As described above, runnables may execute concurrently and very often different runnables are writing and reading simultaneously the same variable. This scenario can lead to potential inconsistency in the data exchanged during the communication.

The Sender-Receiver Communication schema uses a memory sharing mechanism allowing tasks to communicate by means of shared memory. In this sense, the RTE supports two different modes of communication, namely Explicit and Implicit:

- *Explicit*: According to the Explicit communication pattern, a task may directly access to shared variables at any point during its execution. This means that it is possible to produce inconsistencies in the data exchanged when accesses are not protected through proper synchronization or locking constructs.
- *Implicit*: With this communication pattern, tasks accessing shared labels work on task-local copies instead of the original labels. In this case, accesses to the global memory are defined at the beginning and at the end of the task boundaries. In this way, the communication pattern avoids concurrent access in the shared memory during the runnable execution, ensuring integrity in the shared data. At the end of the task execution the local copies are returned to the shared memory. The implicit communication model is implemented as a part of the code generated in the RTE.

As can be observed, both communication patterns provide different levels of data consistency. Accordingly, several solutions have been proposed to provide data consistency and flow preservation in AUTOSAR [24], [26].

Along the last years, there has been an increasing interest in the Logical Execution Time model in different domains, for instance, in automotive [12] or avionics [27] [28]. As we defined before, the main principle of LET is the definition of when the inputs are read and outputs are produced. This model provides independence between the hardware and the logical model, i.e., the application will behave in the same way disregarding where the application is running. The LET model, solves intrinsically some of the problems given typically in multi-core platforms in terms of predictability because the communication and task activities take place at fixed time instants.

In Chapter 4 we provide a single node Intra-ECU end-to-end latency analysis for the described AUTOSAR communication patterns and LET. We propose a formal implementation for the Explicit, Implicit and LET communication patterns, analyzing the impact introduced in terms of memory footprint and communication delay.

2.3 Model-Driven Development

The increasing complexity in software development combined with the need of improving the efficiency/productivity in the system implementation lifecycle, has stimulated the application of new techniques that help software developers in this activity. As stated by Atkinson in [29], since a long time ago researchers have been tried to raise the abstraction level of application software development. Model-Driven development (MDD) [30] is an engineering paradigm used to abstract the software development process. MDD has emerged as a good methodology to improve the efficiency in the system implementation activity. Since the traditional programming paradigm requires

to provide every detail of the system's implementation, MDD allows to abstract this step by just providing a model with the functionality needed by the system. In this context, a model is an abstract representation/description of the system. A model can be used for code generation, because it adequately characterizes the functionality/behavior of a given application.

The Model-Driven development process automates many of the routines in the software development. Moreover, the model of a system under analysis undergoes multiple refinement/optimization stages to obtain a final implementation, that is behaviorally identical, but with specific properties that are amenable to designers. Such a process is typically automated through the so-called Model-to-Model Transformation (M2M) and Model to Text (M2T) transformation. In a nutshell, the code generation process receives the model as input, this input is created in the requirement analysis phase by the software engineers. The model is tested in order to meet with all the constraints defined in the mentioned above phase. Then, if necessary, it converts the model into a different model with the M2M process. This can be, for instance, the conversion of a UML model into another model that better captures certain properties of the target system. Finally, the code is produced, it can be either hand coded or automatically generated in a phase in which the model is transformed into a lower level code representation (e.g., C/C++) through the M2T procedure. Figure 2.9 shows the process described above.

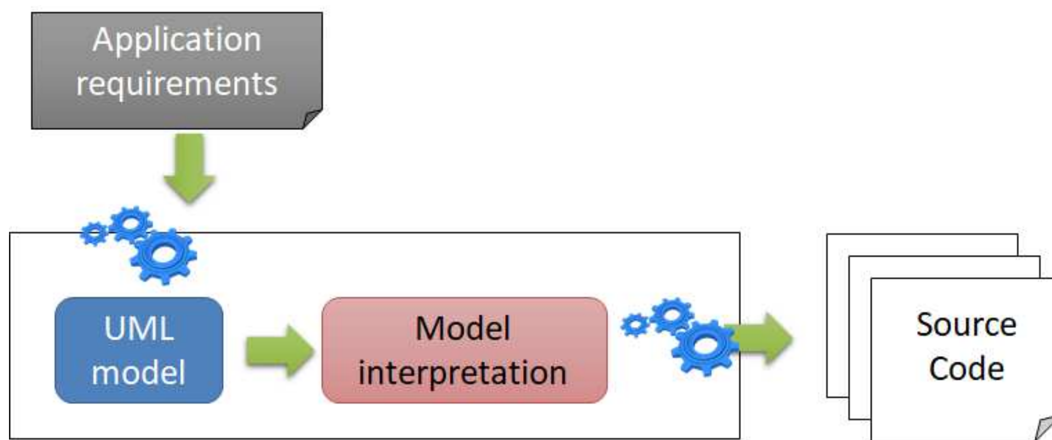


Figure 2.9: Code generation pipeline process.

A modeling language is composed of *syntax* and *semantic* rules [31] that define, namely, the set of language rules, constraints and the meaning of different language tokens. In order to create a model, engineers must first define the language rules, that is, the abstract syntax or *metamodel* [32]. A metamodel is the abstract model used for the

model constraint definition, i.e., the metamodel defines the constraints to comply by the model. In other words, in MDD design, the model is an instance of the metamodel.

Such an approach is extremely beneficial to software developers. First, in model-driven software development, several activities are automated, and the code creation process becomes faster and more efficient. Furthermore, automation, by definition reduces the error rate of the implementation with respect to a human developer. MDD also let developers work at higher levels of abstraction, so they can compose lower-level features (e.g., drivers) as "building blocks" from a higher level of design, this allows to test, find, and fix possible errors. This also lets engineers to automatically generate code in different languages and for different platforms, moreover it is possible to generate automatically the documentation along with the code, by only providing a single and solid model definition. Finally, and most importantly, this approach enables quick verification and certification of source code, enabling its adoption also in specific domains such as avionics and automotive where they are required, for example a model-based approach meets the specifications of IEC61508.

Traditionally, mathematical modelling has been widely used in many engineering domains such as aerospace and avionics [33], [34]. Recently, modeling has been gaining increasing attention in the automotive domain for simulating the real physical system behavior or to test the functionality of a car component. This paradigm is currently broadly used in the automotive domain. Specifically, the 6th part, Annex B of ISO-26262, describes the importance of MDD at software level. An important aspect of model-driven development is that a functional model not only specifies the function performed by the component, but also provides design and implementation information that is used for the code generation phase, i.e., functional model can represent specification aspects such as design and implementation. This approach is widely adopted using commercial off-the-self (COTS) modeling and simulation software tools, such as Matblab Simulink [10] or ETAS ASCET [11].

In Chapter 5 we presented a code generator tool that follows the MDD paradigm to generate the code that will be used to test the findings presented in Chapter 4.

2.3.1 AMALTHEA

AMALTHEA [35] is an open source framework that provides a XML-based document format for modeling embedded multi-/many-core systems, supporting the AUTOSAR [17] standard. It is based on model-driven methodology and adopted by several automotive companies. AMALTHEA is maintained by a consortium of academic and

[10]https://mathworks.com/products/simulink.html
[11]https://www.etas.com/en/products/ascet-developer.php

industrial entities from the automotive and embedded domains. Thanks to its simple and elegant XML-based layout, this model is extremely useful to automotive engineers in the Verification and Validation (V&V) process, and it is being adopted by leading Tier 1 companies such as Bosch Gmbh.

In this context, APP4MC [12] for embedded multi-/ and many-core software development, used by Bosch. The APP4MC platform provides an entire tool chain for tracing, partitioning, mapping and modeling. APP4MC is based on the Eclipse modelling framework (EMF) [36] to model software, hardware, timing behavior and constraints of the system.

The modeling tool (for convenience called "AMALTHEA framework") models both high-level requirements and low-level behavior of software components, e.g., task periods, deadlines, memory access patterns, etc. It also models the hardware platform where the application is targeted, and it supports the abstraction of multi-core architectures. Hiding the intrinsic application information and code, that is often under IPR (Intellectual Property Right) restrictions, AMALTHEA makes it possible for academics to directly interact with real system (SW/HW) models provided by industrial partners for research-related activities, with no licensing issues. Abstracting out functional implementation details, the timing behavior of concurrent tasks can be reproduced without exposing IP-critical internal details, limiting the risks to reverse the software engineering process.

The AMALTHEA tool chain process is depicted in Figure 2.10. The model covers the following aspects:

- *Constraints*. Provide definition for event chains, timing constraints, affinity constraints and runnable sequential constraints.
- *Events*. Definition of the type of event produced in the event chains {start, resume suspend, terminate}.
- *Hardware*. Abstract representation of the hardware target.
- *Mapping*. Mapping of the software components in the hardware task-core mapping.
- *Operating System*. Information about the adopted scheduler and scheduling algorithms.
- *Software*. Definition of the application software components, like runnables, tasks, and data items (labels).
- *Stimuli*. The activation instant of recurring tasks.

[12] Application Platform Project for MultiCore (APP4MC). Url: http://www.eclipse.org/app4mc/

Some of the metamodel features, such as peculiar scheduling policies, are not easily reproducible using a general purpose operating system, but they rather require RT-OSes (such as RTAI [37] or ERIKA [38]), or RT-oriented patches/extensions to generic OSes (e.g., SCHED_DEADLINE [39], PREEMPT-RT [40] or GRUB [41], for the GNU/Linux kernel) For example, the behavior of a set of tasks under a cooperative scheduling [13] policy is not easy to emulate on an operating system without limited preemptive support [42]. In this cases, some information of the metamodel like the above mentioned features are simply ignored.

Since AMALTHEA can be adopted to abstract the behavior of automotive applications, we used the model to derive the end-to-end latency analysis. In this way, it is possible to compute the analysis without having the application code.

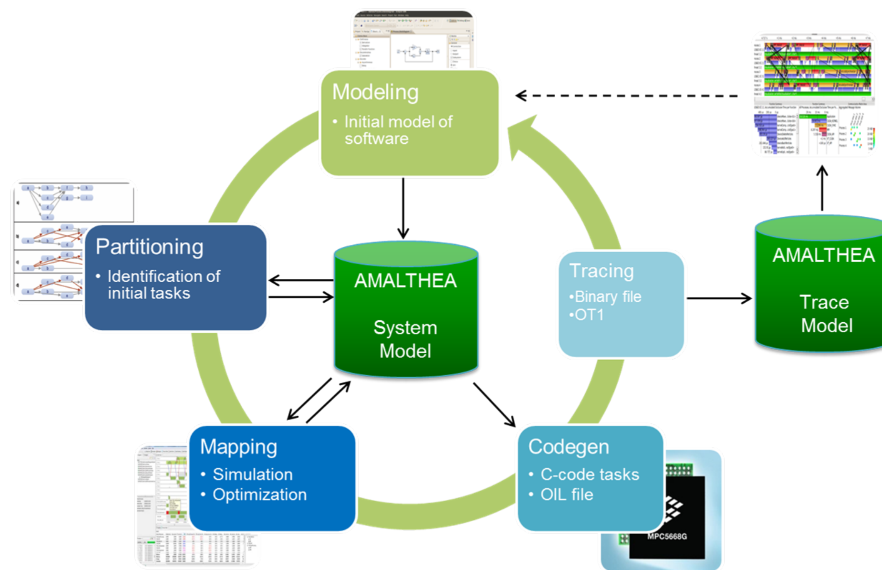


Figure 2.10: Structure of the AMALTHEA model.

[13] Once started, a task within a co-operative scheduling system will continue to run until it relinquishes control.

3 Towards predictability in virtualization platforms

In the embedded systems domain, hypervisors are increasingly being adopted to guarantee timing isolation and appropriate hardware resource sharing among different software components. However, managing concurrent and parallel requests to shared hardware resources in a predictable way still represents an open issue. We argue that hypervisors can be an effective means to achieve an efficient and predictable arbitration of competing requests to shared devices in order to satisfy real-time requirements. As a representative example, we consider the case for mass storage (I/O) devices like Hard Disk Drives (HDD) and Solid State Disks (SSD), whose access times are orders of magnitude higher than those of central memory and CPU caches, therefore having a greater impact on overall task delays. As we argue in Section 2.1.1, virtualization technologies can be used to provide spatial and temporal isolation (Freedom From Interference) thanks to the abstraction and capacity to isolate hardware resources.

In this chapter, we provide a survey of the literature on I/O management within virtualized environments, focusing on software solutions proposed in the open source community, discussing their main limitations in terms of real-time performance. Then, we discuss how the research in this subject may evolve in the future, highlighting the importance of techniques that are focused on scheduling not uniquely the processing bandwidth, but also the access to other important shared resources, like I/O devices. These issues are important to certification authorities in many domains, like: railway, avionics or automotive. The remainder of this chapter is organized as follows. The next section introduces basic virtualization concepts. In Section 3.2 we introduce the motivation behind this chapter. Then Section 3.3 presents a brief description of the safety assesment proccess as is required by the ISO-26262 standard for the development of hypervisors as a mechanism for partitioning mixed-criticality applications. Section 3.4 describes the existing solution based on the Xen hypervisor for I/O management. Section 3.5 discusses statically partitioned solutions for multi-core platforms. Section 3.6 highlights performance, predictability and security issues related to the layered scheduling systems implied by many virtualization techniques. Existing works introducing real-time parameters within the I/O scheduler are summarized in Section 3.7, while

Section 3.8 discusses the additional predictability problems incurred with current SSD devices. A final discussion is provided in Section 3.9 showing promising research lines to improve the predictable management of shared hardware resources by means of properly designed hypervisor mechanisms.

3.1 Introduction to Virtualization

A hypervisor, also called Virtual Machine Manager (VMM), is a combination of software and hardware components that allow emulating the execution of multiple virtual machines (VMs) upon the same computing platform by properly arbitrating the concurrent access to shared hardware resources (see Figure 3.1). Most of the available open source hypervisors are specifically tailored to server applications and cloud computing. In these areas, hypervisors are mainly designed to provide isolation, load balancing, server consolidation and desktop virtualization within the managed virtual machines. However, the emerging of new potential areas for VMMs, such as automotive applications and other embedded systems, and the possibility to exploit multi-core embedded processors are posing new challenges to real-time systems engineers. This is the case of next-generation automotive architectures, where cost-effective solutions ever more require sharing an on-board computing platform among different applications with heterogeneous safety and criticality levels, e.g., the infotainment part on one side, and a safety-critical image processing module on the other side. These domains are independent, with different period, deadline, safety and criticality requirements. Although hypervisors provide high level of isolation by definition, they need to be properly isolated with no mutual interference, or a misbehaving module may endanger the timely execution of a high-criticality domain, affecting safety qualification.

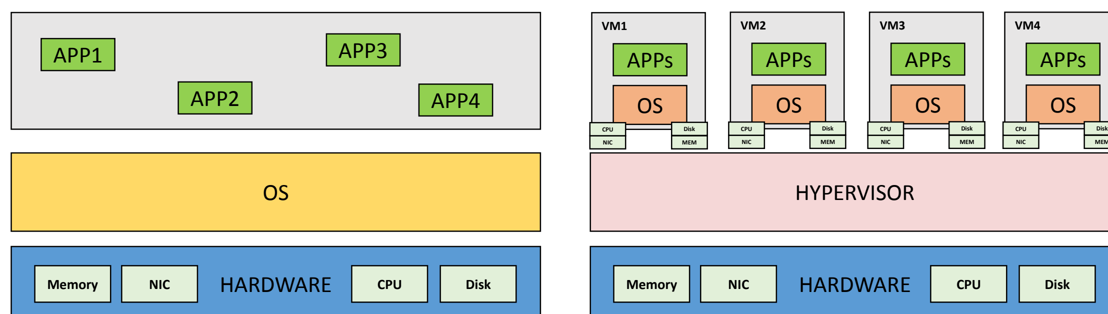


Figure 3.1: Physical and virtualization approaches.

Different approaches can be used to develop and implement an hypervisor with diverse influence in the throughput of the virtual machines under execution [43]. In particular, hypervisors can be classified as follows (see Figure 3.2):

- Type-1 or bare metal that runs directly on the system hardware, maximizing the efficiency with minimum overhead, close to the performance of bare metal operating systems.
- Type-2 or hosted hypervisor that runs on top of a host operating system. Guest hypervisors execute as another process in the operating system.

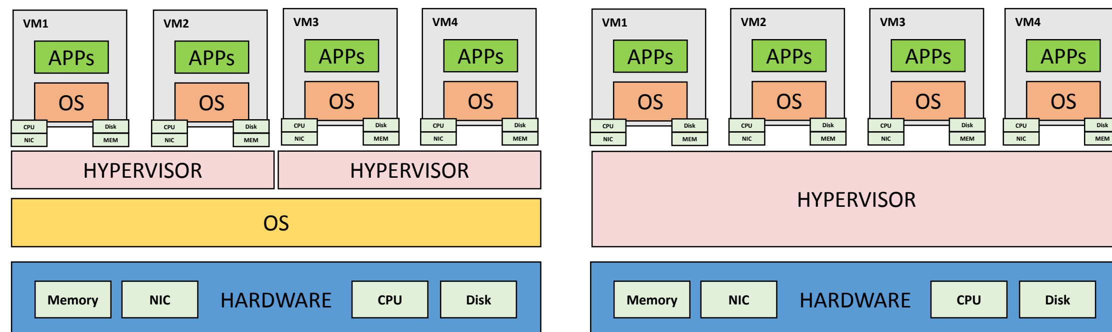


Figure 3.2: Hypervisor design, Type 1 (right) & Type 2 (left)

Mixed criticality systems designed without hypervisors implies the use of a plurality of devices; usually, a restricted number of them are running a general purpose operating system (like Linux or Windows), while other specific devices are running a Real-Time Operative System (RTOS) for managing highly critical tasks. A multi-device solution is therefore unacceptable due to very demanding energy consumption and amount of area required, making a single board managed through a hypervisor the most preferable approach.

In order to provide real-time guarantees, hypervisors either dynamically schedule virtual machines according to a given on-line policy, or they statically partition virtual machines to the available hardware resources. An example of the first category is RT-Xen [44] (now merged into mainline Xen [45]), which implements a hierarchical virtual machine scheduler managing both real-time and non-real-time workloads using the Global Earliest Deadline First (G-EDF) algorithm. On the other hand, statically partitioned solutions tend to isolate virtual machines onto dedicated cores, with an exclusive assignment of hardware resources. An example of this approach is given by Jailhouse [46]. Jailhouse, developed by Siemens, is a Linux-based hypervisor oriented to real-time, it isolates the virtual machines in small cells with few lines of code (13513

written in C), removing all of the unnecessary features (e.g., hooks for diagnostic tools), and allocates the virtual machines by pinning them to the computing cores. It also allows running bare-metal applications aside to Linux. Accordingly, Jailhouse does not allow multiple virtual machines to share the same core. An advantage of this latter approach is that the resulting hypervisors have a typically smaller code footprint, implying much lower certification costs. Indeed, reducing the code size is a prominent characteristic of other recent VMMs, like NOVA [47] and bhyve[1]. Indeed, NOVA OS Virtualization Architecture is even smaller (approx, 9K LoCs written in C++), and, similarly to Jailhouse, it is capable of running virtual machines and bare metal applications side-by-side. However, unlike Jailhouse, NOVA is not pinning physical cores, but it implements preemptive priority-driven round-robin scheduler.

No matter which virtualization approach is taken, I/O for storage devices might become a bottleneck. This is due to the added layer of complexity introduced by the hypervisor itself, as shown in Section 3.6. Most of the current literature on resource access arbitration for virtualized environments mainly focuses on CPU scheduling (see for example surveys [48] and [49]), neglecting the huge impact that the access to other shared hardware resources, like Hard Disk Drives (HDD) and Solid State Disks (SSD), may have on time-critical tasks. In view of this consideration, this chapter provides a survey on the state-of-the-art on I/O virtualization and concurrent HDD/SSD read/write operations. We will discuss the applicability of previously introduced solutions to I/O arbitration for enhancing the real-time guarantees that may be provided in a virtualized environment. Main limitations of classic fair provisioning schemes to resource sharing will be highlighted.

We are interested in software-based solutions that do not require customized device controllers and hardware mechanisms to obtain the desired behavior. Therefore, most of the addressed works deal with virtualized approaches that schedule the access to storage devices by means of a hypervisor or similar mechanisms, shaping the I/O requests to guarantee a given I/O bandwidth to multiple partitions/cores. For each of the presented works, we will highlight the main weaknesses and limitations, in order to stimulate the real-time research community to undertake a more rigorous and structured effort towards achieving the required predictability guarantees.

Contributions are divided by contexts. In this respect, a first coarse-grained distinction is made considering the technology used for data storage: rotational or non rotational. HDDs and flash-based devices, such as SSDs, may have similar issues when it comes to arbitration of concurrent accesses, but their radically different operating principles entail different problems to solve in order to ensure a predictable behavior. A finer-grained distinction is related to arbitration policies, examining how different I/O

[1] https://wiki.freebsd.org/bhyve

scheduling algorithms behave in a virtualized environment and whether they are able to satisfy hard/soft real-time guarantees. It is worth stressing that most of the work on virtualization in I/O environments is based on or concerned with the Xen hypervisor due to its popularity.

3.2 Motivation

There are multiple motivations under this chapter. The initial reason triggering this study relates to the problems encountered when trying to guarantee bounded shared resource access times to tasks concurrently executing on a multi-core platform. Even if often neglected by theoretical works on real-time scheduling, a great share of the predictability problems of modern multi-core platforms is due to potentially conflicting requests to shared hardware resources like caches, bus, main memory, I/O devices, network controllers, acceleration engines, etc. The arbitration of the access to the mentioned shared resources is often hardwired and cannot be easily controlled via software. The enforced policies are mostly tailored to improve average case performance and throughput, often conflicting with the predictability requirements of real-time applications. Finally, low level details on the arbitration policies are difficult to obtain and may significantly vary on different architectures. This makes it extremely difficult to develop a tight timing analysis even for simpler platforms. To sidestep these problems, we are studying scheduling solutions that aim at avoiding conflicts on the device arbiter, by properly shaping the device requests from the different cores. Hypervisors are natural candidates in this sense, providing a centralized decision point with a global view of the requests from the various partitions. This would allow taking the unpredictable arbiters out of the scheduling loop, leaving the resource management at hypervisor level. Before implementing such a solution, we examined the existing related approaches for managing shared resources in virtualized environments, taking storage devices as a representative example.

This choice is mainly due to the large interest in I/O scheduling within the open source community. Modifications to the current Linux schedulers are constantly being proposed and evaluated. For example, at the time of writing, a new scheduler denoted as BFQ (Budget Fair Queuing) [50] is undergoing evaluation for being merged into mainline Linux. This I/O scheduler is based on CFQ, the default I/O scheduler in most Linux systems. Among other goals, BFQ is designed to outperform CFQ in terms of the soft real-time requirements that can be guaranteed to multimedia applications. However, it remains unclear how the proposed modifications can deal with harder real-time constraints, given the unpredictable technical characteristics of storage devices.

A second motivation descends from the consideration that sub-optimal arbitration policies of an I/O storage device can be the primary cause of blocking delays and performance drops. This is due to the considerably worse latencies and bandwidths of storage devices with respect to other shared resources, such as central memory or CPU caches. As an example, the random access times to L1, L2, L3 and DDR main memory on an Intel® i7 architecture are in the order of 1ns, 10ns, 50ns and 100ns [2], respectively. In contrast, the random access times to SSDs and HDDS are considerably higher, in the order of 100us and 10ms [3], respectively. Despite the cost of SSDs random accesses is predicted to drop to 10/50us in the next years, the gap from the main memory access times would still be of at least two orders of magnitude. Due to this difference, it is of paramount importance to properly schedule and coordinate the access to storage peripherals.

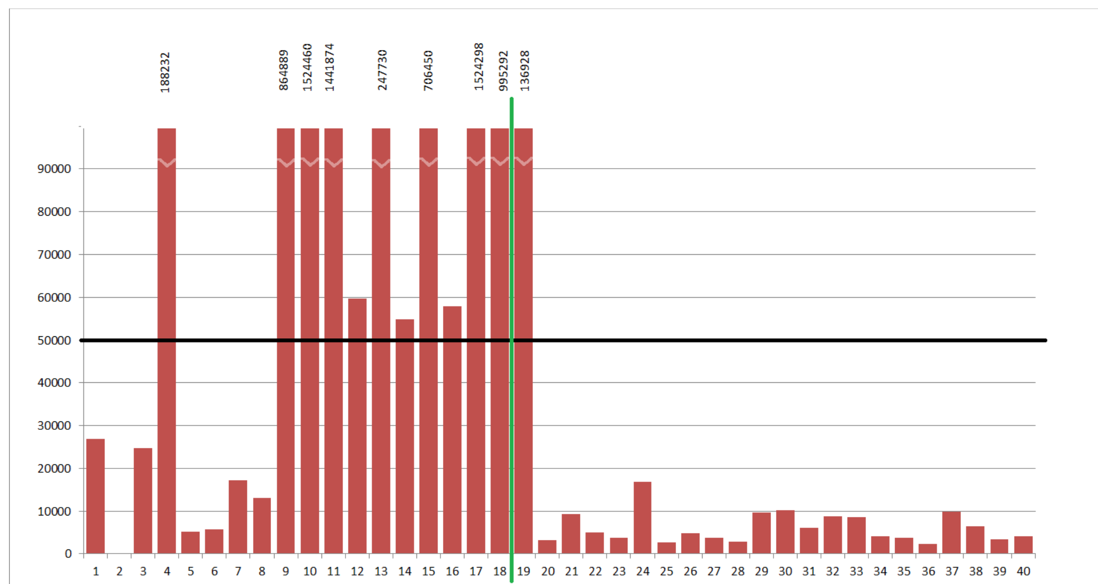


Figure 3.3: Summary of I/O experiments in Xen.

A third motivation is related to the abundant presence of I/O scheduling research in cloud and server virtualized scenarios. The major concerns in these fields are performance and fairness, rather than real-time constraints. Also, the concept of fairness is mainly applied to CPU scheduling, rather than on the access to shared resources. Consider the widely known Xen hypervisor [45]. Xen allows the system administrator to specify the policies that regulate how guests are scheduled on the various cores. We

[2]Intel Performance Analysis Guide http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf Last accessed on May 2017.

[3]HP Solid State Drives (SSDs) for Workstations http://h18000.www1.hp.com/products/quickspecs/13379_na/13379_na.html Last accessed on May 2017

can specify that a VM can be scheduled with RTDS (Real-Time Deferred Scheduling) and a different CPU with a Credit scheduler. By doing so, we are not going to arbitrate access to the CPU (as they are scheduled in different cores) but we want to specify requirements for the first VM and a non real-time domain for the credit scheduled VM. However, a Credit-scheduled virtual machine that runs an I/O intensive task can cause a priority inversion towards other RTDS-scheduled machines, even if the latter require much less I/O bandwidth. In order to further prove the validity of this motivation, we reproduced this priority inversion with a simple experiment in a Xen virtualized environment. The experiment involved a workstation managed with Xen 4.5.0 and equipped with a quad-core Intel® i7 processor, disabling *hyperthreading*. We setup two virtualized disk partitions using LVM (Logical Volume Manager) on a rotative HDD. The read peak rate of the HDD was ∼130MB/s. The device was paravirtualized. We created two guests pv1 and pv2, pinned to two different cores, each accessing one of the two partitions. The workload executed by these two virtual machines is as follows:

- pv1 is a Credit-scheduled virtual machine, associated with the Xen default scheduling weight (see Section 3.4 for a brief introduction of the Xen Credit scheduler and the description of its parameters). pv1 executes a non-critical, non-real-time, I/O-intensive application, sequentially reading a single 1GB-file. Such an application acts as an interfering workload to other real-time tasks on a different domain.
- pv2 is an RTDS-scheduled virtual machine that runs a single task with a 50ms period. The end of the period is assumed to coincide with its relative deadline. Within its period, this guest has to read a memory-page-sized (4KB) chunk of data, randomly chosen out of a 1GB-file in its partition. This setup allows reproducing the worst-case HDD access latency, which, for random reads, has a bandwidth of ∼0.63 MB/s, corresponding to an average latency of around 5ms for a 4KB memory-page read.

In order to rule out any performance bottlenecks into the privileged domain we assign the remaining memory and cores to Dom0. Despite the large slack of the RTDS guest (pv2) to complete its memory read and its higher priority, the I/O interference causes pv2 to experience a large number of deadline misses. Figure 3.3 shows the results of the experiment sampling 40 periodic I/O read accesses (x axis) by the RTDS domain. The y axis represents the time taken by each request in μs. Each bar represents the actual I/O request time. The horizontal black line indicates the period between subsequent requests, while the vertical green line corresponds to the instant when the interference operated by the Credit scheduled domain (pv1) is over. As can be easily seen, pv2's requests starve during the read process of the Credit guest, which almost monopolizes

the access to the HDD. In contrast, when the interference created by pv1 is over, pv2 does not experience any deadline miss. In retrospect, this behavior is not surprising, as the higher priority provided to an RDTS guest affects only the scheduling on different CPUs, but it has no effect on I/O scheduling. In other words, the priority is not transferred from CPU to I/O.

Last but not least, providing freedom from interference between ECUs in virtualization platforms, is an important issue in the automotive domain. In Section 2.1.1 we highlighted some of the constraints defined by the ISO-26262 in terms of spatial and temporal isolation. In this sense, the ISO-26262 establishes good directives for providing freedom from interference. In the next sections we focus on disk-I/O sources of interference, and existing solutions for providing temporal and spatial isolation in the state of the art in this context.

3.3 Virtualization and safety issues

As we argued in the beginning of this dissertation, the exponential increase in the vehicle's complexity and the integration of multiple functionalities in the same on-board platform have completely changed the way in which vendors designed it's vehicle-solutions. In this sense, car makers are not only concerned with safety and reliability but also with the costs of the components. Indeed, that is why most of the car makers are transitioning from single-core to multi-core architectures. However, the development of safety-related components is still the main concern in the automotive domain.

The integration of multiple applications in the same platform leads to implement safety-related and non-safety-related applications in the same MPSoC. In this context, applications can communicate among each other at inter- and intra-core level. However, an error in one partition can corrupt the other partitions triggering unwanted fault propagation problems. This is one easy example, but there are a lot of safety related problems that can be presented in a vehicle's software.

At certification level, it is still challenging how to design and validate a system that integrates mixed-criticality components. In order to ensure safeness in the development of safety-critical systems, the Part 6 of the ISO-26262 standard establishes a guidance document that should be considered during the development of software components. This part of the standard provides further recommendations for the development of safety critical software for automotive systems, from the planning of the functional safety activities for the software development, to the specification of software safety requirements. From a timing perspective, the standard specifies that the development process shall consider the "Execution or reaction time derived from the required re-

sponse time at the system level." [4]. It is worthwhile to mention that the standard includes a set of requirements for software design and implementation. For instance, it defines software constraints for the use of pointers and recursion. Both constraints are mandatory requirements for ASIL-D software components.

Turning to the integration of mixed-criticality components in the same on-board platform, software components implemented on a multi-core platform have to comply with a safety assessment process. Many of the software safety requirements are derived from technical safety requirements. In order to comply with these technical safety requirements, it is needed to achieve the so called *Freedom from Interference (FFI)*. In this context, considering that software components cannot be physically separated, providing freedom from interference is a very important issue. According to the annex D, "FFI is the absence of cascading failures between two or more elements that could lead to the violation of a safety requirement". To better clarify this concept, let us consider the following example: an ASIL C software component that controls periodically the acceleration of the vehicle, establishes a communication with an ASIL D component that is in charge of the lane keeping assist system (LKAS). In this case, the ASIL D component is considered free from interference with respect to the ASIL C component, if a failure of the component ASIL C component does not interfere ASIL D component and vice versa. Specifically, Freedom From Interference is provided if:

1. "Element 1 is free of interference from element 2 if no failure of element 2 can cause element 1 to fail" - Figure 3.4a
2. "Element 3 interferes with element 4 if there exists a failure of element 3 that causes element 4 to fail." - Figure 3.4b

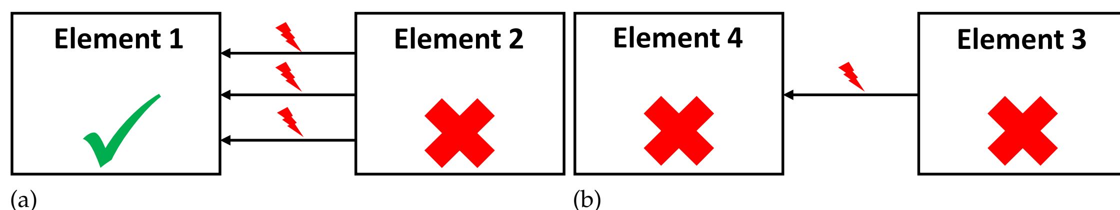


Figure 3.4: Freedom From Interference Example - ISO-26262. Definition 1.49.

Achieving freedom from interference between partitions allows deploying software components with different ASIL levels in the same system. In this way, lowest ASIL components involved in the certification process do not inherit the integrity level of highest ASIL software components, reducing in this way design and production costs.

[4] In Chapter 4 we derived a timing and schedulability analysis of an automotive application that is in charge of the engine management system (EMS)

At implementation level, software components can be interfered in many different ways. The purpose of the standard is to guarantee that there is no interference between components. ISO-26262 defines three types of freedom from interference:

- Temporal interference: it is necessary to ensure that one application does not interfere with another application in terms of time execution, for instance, consuming its CPU execution time or blocking a shared resource used by another application. This kind of interference can lead to blocking of execution, deadlocks or livelocks. According to the standard, the strategies that can be considered to avoid temporal interference are: cyclic execution scheduling, fixed priority based scheduling, time triggered scheduling, monitoring of processor execution time, program sequence monitoring and arrival rate monitoring. On the other hand, there are some novel mechanisms in the literature that aims to provide temporal isolation using cache coloring techniques. For instance in [51], [52], authors proposed page coloring and cache lockdown mechanisms to enforce a deterministic cache hit rate on the most frequently accessed memory pages. In some sense these techniques can be considered mechanisms to provide also spatial isolation since the data is "partitioned".

- Spatial interference: in this case, one partition cannot change data nor the code of another partition. For instance, the access to the shared memory by the software on one core can cause data inconsistency in the software allocated on a different core leading to such problems as race conditions, data starvation or deadlocks. Typically, freedom from spatial interference is avoided using mechanisms like MPU (Memory Protection Unit) to control access to the shared memory, or CRC (Cyclic Redundancy Check) for error detecting data or even the Intel's Cache Allocation Technology (CAT) [5] CAT helps managing the cache by providing mechanisms to control where the data is allocated in the last-level cache. In this way, it is possible to provide isolation between partitions and prioritization in the accesses at software level.

- Communication interference: generally, ECUs communicate among each other sending signals, data or messages through network or CAN bus. In this case, it is necessary to protect this communication against missing or corrupted data. The mechanism proposed to protect such a communication is called E2E (end-to-end) protection, it provides a protocol to detect and protect the effects of faults when exchanging data.

[5] https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology Last access 1/10/2017

Summing up, we believe that the partitioning techniques described above, combined with the isolation mechanisms provided by virtualization technologies (like type 1 hypervisors), fit perfectly to meet the requirements recommended by the ISO-26262. Indeed according to the standard: "Virtualization technologies can support the argument to guarantee freedom from interference between software elements running on a multi-core platform." Temporal and spatial isolation can be provided by statically partitioning virtual machines into the available hardware and improved using the mechanisms described above.

In this context, open source solutions like Automotive Grade Linux [6], Jailhouse [53] or commercial solutions like QNX [54] or PikeOS [55], aim at integrating hypervisors as a solution to provide strong isolation between software components.

3.4 I/O scheduling in virtualized environments

A significant number of contributions addressed I/O virtualization issues by modifying the existing Xen Credit scheduler. The Credit scheduler is the default CPU scheduler, and it works by distributing *credits* among virtual machines in proportion to their *weights*. Weights can be freely set on guest creation. A virtual machine consumes its credits while running on a physical CPU, and is in an *over* or *under* priority status depending on whether it has exceeded or not its share of CPU resource within a considered time window. Credits are redistributed for each virtual machine by a specifically designed system-wide thread. For a deeper explanation of the Xen Credit scheduler, please refer to the official documentation[7].

The part of the credit scheduler that relates to I/O scheduling is connected to what is known as *boosting mechanism*, i.e., an additional *boost* priority status that allows performance improvements for I/O-intensive guests [56]. A very demanding I/O task running on a guest causes the virtual machine to get blocked often, leading to a very limited credit consumption, with the guest always in the *under* state. When waking up after completing an I/O request, the VMM will grant the guest a boost priority, allowing it to preempt other running virtual machines to process the requested data.

Different works tried to improve I/O scheduling in virtualized environments by acting on the mechanisms used by Xen to assign priority statuses within guests and on the above described boosting mechanism. In [57], the authors developed a solution that extends the mechanisms of the Xen Credit scheduler. They introduced the notion of *task-aware (I/O) scheduling* arguing that a task-aware model is beneficial for scheduling purposes, especially in situations where mixed resource usage and I/O-intensive tasks

[6] https://www.automotivelinux.org/
[7] http://wiki.xen.org/wiki/Credit_Scheduler

are concentrated in specific domains. Once the VMM has knowledge of which guest has higher I/O bandwidth requirements or specific latency-related constraints, the hypervisor will use this information to decide how and when to assign the boost priority to those I/O-bound guests. In [58], the authors followed a somehow similar, but more complex, approach. They developed a technique for speeding up I/O virtualization using direct I/O with hardware IOMMU. To support a real-time response for high quality I/O virtualization, a new `REAL_TIME` priority state is added to the Xen Credit scheduler supporting preemption. Consider a latency-sensitive application running inside a guest to which the associated latency-critical pass-through device is assigned. Whenever the pass-through device fires an interrupt, the associated virtual machine is automatically promoted to the `REAL_TIME` state, triggering a preemption of any non `REAL_TIME` guest to schedule that particular machine right away. While the first contribution [57] mainly focuses on achieving a fair behavior among domains, the latter [58] shows promising results in terms of I/O throughput and latencies. Due to the low latency values obtained, the authors in [58] claim to have designed a real-time virtualized environment, but no experimental or analytical evidence has been provided to support these claims using classic real-time metrics, such as schedulability ratio, worst-case response times, deadline misses, etc.

Another interesting approach is presented in [59] and [60]. Both works are focused on adaptive time-slice sizing in Xen. In the first contribution, the authors modified the Xen Credit scheduler to guarantee Quality of Service (QoS) requirements for streaming audio applications in virtualized environments. They designed an adaptive modifier of the Xen Credit scheduler that allows flexible time-slices and real-time priority flags to be dynamically assigned to guests. According to their presented results, the authors were able to improve the responsiveness of latency-sensitive applications, achieving some kind of soft real-time guarantee. They tested their implementation by pinning multiple virtual CPUs (vCPUs) to the same physical core, hence testing concurrent I/O requests rather than parallel I/O operations. In [60], the authors adopt a similar mechanism for an on-the-fly adaptation of the time slices within the I/O scheduling policies (mainly CFQ and Anticipatory) of the Linux kernels that are executing within the Xen unprivileged domains. Here, parallel HDD requests are explicitly considered, showing an improved latency. However, even if improving latencies is an important step towards predictability, a system that dynamically adapts scheduling constraints, such as time slices, makes it very difficult to identify worst-case scheduling settings where to build a tight timing and schedulability analysis.

3.5 Multi-Core partitioning and virtualization

Another promising direction to obtain a predictable behavior is to exploit the multi-core nature of today's CPUs, assigning specific I/O handling functions to specific cores. This can be accomplished with CPU hardware extensions and/or a different virtualization paradigm using partitioning-based hypervisors. The work in [61] proposes a Xen implementation monitoring runtime information of the bandwidth requirements of the different guests. Specific functions related to I/O handling are pinned to specific cores, e.g., one core is used for driver-related aspects, another one to handle I/O events, another one for generic computations. Performance improvements are claimed in terms of bandwidth and latencies with a slight drop in the performance of compute-intensive tasks.

Another interesting contribution that relates to core specialization is presented in [62]. A VMM based on hardware resource partitioning is taken into account, proposing a hypervisor (SplitX) that resembles the operating mechanisms of Jailhouse[8]. Specialized cores can handle I/O related interrupt and hypervisor instructions. The authors claim that I/O level performance is expected to reach near bare-metal performance, by means of hardware extensions to allow directed inter-core signals for events notification but also for managing resources belonging to other cores. An example of this latter feature may allow a core to assign specific values to the registers of a different core. Unfortunately, this latter batch of related works mainly deals with performance improvements. Even if a considerable drop on latency values is a promising start for achieving real-time guarantees, these approaches are not concerned with obtaining worst-case delay bounds and a tight timing analysis.

In a recent publication [63], a scratch pad centric non-virtualized architecture is presented in which real-time requirements are explicitly taken into considerations. Similarly to the other approaches presented in this section, a specific core is delegated for I/O operations exploiting a dedicated I/O bus. Task executions are decoupled from instruction and data loading using a Time Division Multiplexing (TDM) approach. I/O operations are included in the same time slice used for task loading/unloading. While this latter contribution present a very rigorous and sound timing analysis, it does not explore I/O intrinsics threats to predictability in virtualized environments, nor it addresses the problems of having multiple I/O tasks hogging the dedicated core.

[8]Jailhouse does not yet support any mechanism to predictably manage the concurrent access to shared resources like I/O devices, each of which is statically pinned to a given partition/core having exclusive access to it.

3.6 Perfomance and security issues introduced by I/O virtualization

It is straightforward to observe that a hypervisor allowing its guests to run entire operating systems can easily introduce noticeable overheads due to the local CPU and I/O schedulers. Virtualized platforms, such as Xen, have their own CPU arbitration policies for scheduling guests, but also privileged domains have to go through their own block layer, while each guest runs its own kernel with different local scheduling policies for both CPU and I/O, hence providing an added level of complexity when accessing the storage device. This hierarchical structure is known to cause performance drops compared to bare-metal systems, but it also exposes a more complex architecture that dramatically increases the difficulties of deriving a sound timing analysis.

The performance overhead issue has been studied in different works [56], [64], [65]. In a recent paper [66] the authors measured the overhead of I/O stack duplication between host and virtual machines running KVM as VMM. It also provided a very complete survey on previous tests on different VMMs that eliminated a layer of the IO scheduler. A simple QEMU + virtIO solution is shown to outperform almost all tested scenarios.

The hierarchical organization of these kind of hypervisors also poses significant security threats. In [67], an untamed I/O intensive task running within a compromised/-malicious guest is used to slow down and interfere with other supposedly separated domains. For this reasons, the authors recommend to adopt a separate and unique I/O scheduler for virtualized environments.

We believe that such an I/O scheduler should be designed with the same guidelines considered when implementing efficient CPU real-time schedulers, ensuring a proper isolation among tasks that require disk access, while allowing them to complete their workload within given deadlines. On this latter consideration, it has to be pointed out that the Linux kernel provides features such as control groups (cgroups) that can be used to isolate, limit and control disk (rotational or SSD storage device) accesses of sets of processes. For example, cgroups can be set within privileged domains to limit resource usage by unprivileged guests. However, this feature does not provide for specific scheduling policies, rather it relies on the underlying I/O scheduler, and on its policy, for enforcement. In this respect, current Linux I/O schedulers implement too coarse policies for typical real-time requirements. In addition, the resource allocation scheme of cgroups is based only on weights, which is not sufficient scheme for most real-time applications.

3.7 Deadline-aware I/O scheduling

The need to provide tighter real-time guarantees to tasks accessing disk I/O has been a problem addressed since the early 90's. In [68], Reddy et al. presented a scheduling algorithm called SCAN-EDF that combined the Earliest Deadline First (EDF) [69] and SCAN schedulers for minimizing request latency in HDD while serving deadline constrained tasks. During the years, this algorithm has also been modified and improved by means of heuristics, such as batching and delaying requests, or aggregating multiple queues of requests. In [70], [71] and [72], the Xen I/O architecture has been modified to include deadline-based scheduling for the storage in a virtualized environment. In [70], [71], a two level scheduler called Flubber is introduced. The first level defines the throughput using a credit-rate controller to ensure performance isolation, while the second level applies Batch and Delay EDF (BD-EDF) to manage the request queues from the different guests. Even if the authors claim that Flubber improves Xen performance and allows the system administrator to specify deadlines, no results are provided to evaluate the worst-case delays and blocking times needed to establish a sound timing analysis. In [72], a similar approach is used, where the first level accumulates the amount of I/O requests in a fixed time slice while analyzing the disk bandwidth, and the second level exploits the deadline-modified SCAN algorithm reordering the requests according to the deadline group and to their location on the disk. While there is a performance enhancement for I/O intensive workloads, neither this work appears to lend itself to the analytical guarantees required in a real-time setting.

3.8 Real-time issues in SSDs

Solid State Drive based storage devices deliver from 5 to 10 times the bandwidth of a HDD, while maintaining a low power consumption and a much stronger resistance to shocks and vibrations. These features make SSDs the primary choice for applications in the automotive and avionics sectors, in which embedded platforms have to sustain prolonged vibrations while still delivering high performance. This makes it particularly interesting to understand whether the previous considerations coming from experiments executed on HDDs equivalently hold for guests sharing access to a SSD. In this respect, it has to be pointed out that the *intrinsic operating mechanisms* of SSDs pose significant problems towards the design of predictable hard real-time systems. This is due to the fact that flash memories are a write-once and bulk-erase medium, that implies that a flash translation layer (FTL) and a Garbage Collection (GC) mechanism are needed to provide applications a transparent storage service. A naïve best effort GC policy can unpredictably start segmentation operations causing tasks to wait for potentially long

blocking times. The authors in [73] focused on providing hard real-time guarantees for the GC phase in small NAND flash devices by proposing a token based garbage collection system. The presented results showed that the implemented system is predictable and robust to interferences introduced by non real-time tasks. A prototype is tested in a 16MB NAND-flash drive with two real-time tasks and one non real-time task in a manufacturing system scenario, with no deadline violations until high CPU utilization. A more recent contribution [74] observed that the previous solution does not scale well, making it impossible to apply to modern SSDs having a much larger storage capacity. An FTL implementation (KAST) is then proposed to allow the user to control the worst case blocking time by tuning some GC parameters.

3.9 Summary

Hypervisors represent a possible solution to bypass unpredictable scheduling policies enforced by off-the-shelf arbiters for the access to shared hardware resources. By taking informed decisions on the scheduling of the different requests coming from multiple tasks, a hypervisor may provide stronger timing guarantees to real-time tasks, predictably limiting the delays due to interfering requests on the shared devices. Taking I/O scheduling as a representative case for resource sharing, we highlighted the main results concerned with improving the delays due to competing accesses to storage devices in virtualized environments. We showed how I/O intensive tasks within non-critical virtual machines can easily cause more critical partitions to experience high blocking delays, leading to repeated deadline misses. This was the case with the Xen hypervisor, whose critical partitions are "privileged" only when assigning processing bandwidth, but not when arbitrating the access to other shared resources. We argued that smarter scheduling policies are needed, that take into account the timing requirements of the different tasks/partitions also when arbitrating the access to shared devices.

We showed that existing mechanisms help to improve the blocking delays are mainly tailored to obtain better average performance or achieve a fairer behavior, but cannot be leveraged to develop a sound timing and schedulability analysis. While it would be possible to manually tune the bandwidth allocated to each partition when accessing an I/O device, e.g., by playing with cgroups parameters in Xen, such a solution has clear limits in terms of flexibility, efficiency and responsiveness, preventing a tight timing analysis. Moreover, we pointed out that hypervisors like Xen and KVM add further layers of complexity to guest operating systems, with repeated scheduling and block layers coupled with para-virtualized driver architectures, making it very difficult to formalize the I/O scheduling model. Partitioned hypervisors seem more suitable in this sense, especially when the number of cores increases and each domain can be

statically assigned to one or more dedicated cores. Still, most of the available partitioned VMMs do not allow for a predictable and concurrent access to shared devices, but they either exclusively pin each resource to a selected domain, preventing tasks running on other partitions to access it, or they implement para-virtualized schemes that are not aware of the different real-time requirements.

4 Towards predictability in AUTOSAR

In the last chapter we highlighted the drawbacks concerning I/O shared resource management within virtualized environments. Applications typically communicate data among each other, as we discussed before, an error in one partition may lead to errors in other partitions producing undesired fault propagation problems (see Section 3.3). While virtualization is popular to isolate and subdivide the resources, concepts like data consistency or determinism in the partition or task communication are out of the virtualization scope. Different mechanisms can be used to provide determinism and data consistency. Those mechanisms are used to communicate tasks at software level. However, we believe that they are promising to guarantee determinism and data consistency if needed in the communication of distributed tasks, i.e., those mechanisms are useful not only to communicate tasks allocated in the same partition but also in different partitions (VMs or ECUs for instance). Nevertheless, they are usually used to manage the communication at inter- and intra-ECU level. In this chapter we analyze and propose the implementation of different techniques that are used for shared-memory inter-task communication in the automotive domain.

We will consider the automotive domain. In this context, modern automotive embedded systems are composed of multiple real-time tasks communicating by means of shared variables. The effect of an initial event is typically propagated to an actuation signal through sequences of tasks writing/reading shared variables, creating an "effect chain". The responsiveness, performance and stability of the control algorithms of an automotive application typically depend on the propagation delays of selected effect chains. Different types of communication have been proposed to ensure data consistency, with different impacts on the resulting propagation delays of effect chains, as well as in terms of overhead and memory footprint.

In this chapter, we explore the trade-offs between three communication patterns that are increasingly being adopted for industrial automotive systems, namely, Explicit, Implicit, and Logical Execution Time (LET). Furthermore, a novel timing and schedulability analysis is provided for tasks scheduled following a mixed preemptive configuration, as specified in the AUTOSAR model. Moreover, an end-to-end latency characterization is then proposed, deriving different latency metrics for effect chains under each one of the

considered patterns. Finally, the results are compared against an industrial case study consisting of an automotive engine control system provided by Bosch [75].

4.1 Introduction

In recent years, the amount of electronics in automotive vehicles has risen dramatically, constituting a significant share of the overall cost of the vehicle. The technological reason behind such a trend in the automotive industry is due to an increased number of safety and control functionalities that are being integrated in modern cars, as well as to the replacement of older hydraulic and mechanical direct actuation systems with modern by-wire counterparts, leading to an increased safety and comfort at a reduced unit cost. Well-known examples are electronic engine control, ABS, electronic stability program (ESP), active suspension, etc.

As we discussed at the beginning of this dissertation, the introduction of multi-core processors in the industry allow application providers to counter the thermal and power-related limitations to the Moore law demand for computing power without incurring thermal and power problems. In the automotive domain, multi-core platforms bring major improvements for some applications requiring high performance such as high-end engine controllers, electric and hybrid powertrains, advanced driver assistance systems, etc. Moreover, the increased computational power of multi-core platforms allows integrating into a single controller multiple functionalities that were spread around different electronic control units (ECUs), reducing the number of computing units as well as communication overhead. Some of the cores are dedicated to handling low-level services (AUTOSAR's Basic Software) or high-level services (AUTOSAR's Application Software), provided the necessary timing and safety constraints are satisfied, adapting existing design methods to the new multi-core paradigm.

However, distributing tasks and runnables over multiple cores may have a significant impact over the control performance of a given application, due to the concurrent execution of multiple tasks partitioned on different cores that communicate through shared memory. The typical way tasks communicate in the AUTOSAR model is through shared labels, that are written/read by two or more runnables. Different communication patterns have been proposed in the automotive industry to ensure a consistent management of shared labels, i.e., Explicit, Implicit and Logical Execution Time. Each of this patterns has a different impact over the communication latencies experienced by tasks accessing the same shared variable. In particular, automotive applications are particularly concerned with optimizing end-to-end propagation latencies of input events that trigger a chain of computations leading to a final actuation or control action. An "effect chain" (*EC*) is defined as a chain of tasks, where each task has a runnable

writing a shared label that is then read by a second task; this latter task processes the read variable, and then writes a different shared label, which is then read by a third task. And so on, until the end of the chain. The amount of time that elapses from the first input event until the end of the chain may significantly affect the control performance of the considered application.

In this chapter, we analyze in detail the propagation latencies of event chains composed of multiple tasks under different types of communication. We propose and characterize meaningful latency metrics to evaluate the control performance of selected event chains. For each considered communication pattern, we characterize wort-case scenarios that lead to the largest latency of the event chain, deriving analytical upper bounds of the worst-case propagation time of an input event. We also provide valid upper bounds of the response time of each runnable for tasks scheduled either under the preemptive or the cooperative scheduling policy supported in AUTOSAR. To our knowledge, this is the first work that provides such an analytical characterization and comparison of end-to-end latencies under different industrial-grade communication patterns, for task systems compliant with the AUTOSAR scheduling model.

The Chapter is organized as follows. The following section introduces the related work. Section 4.3 presents the scheduling model, as well as the preemptive and cooperative scheduling algorithms, and related response-time analysis, for AUTOSAR tasks. Section 4.4 describes the considered communication models, discussing the additional memory and communication overhead implied by each communication pattern. Section 4.5 derives analytical upper bounds of meaningful end-to-end latencies for each considered communication pattern. The analytical bounds are then instantiated in Section 4.6 to an automotive industrial case study, consisting of an engine control system by Bosch [12], based on concurrent AUTOSAR tasks partitioned onto a multi-core system. Finally, section 4.7 presents our conclusions and directions for future works.

4.1.1 Data consistency and time determinism issues

In this subsection we will discuss the importance of data consistency and time determinism.

Data consistency. Data consistency is a broad and ambiguous term. This concern is typically given in database management systems (ACID transactions). We will refer to it as the "data model" that do not lead to a critical state, i.e, a data model in which each entity expects always the correct data. In real-time systems data consistency is an important issue and in certain applications may be critical. For instance, consider Figure 4.1, we have two tasks that shares a variable. In the first case (the upper part of the figure) τ_2 starts the execution, then it initialize a variable x to 0, at some point in the

execution τ_1 preempts τ_2 that changes the value to -1 and turns the data inconsistent. When τ_2 resume the execution, it executes a square root of -1 that causes an error. We have a very similar situation in the second case (write conflict) where τ_2 produce an error when it tries to execute a division by zero.

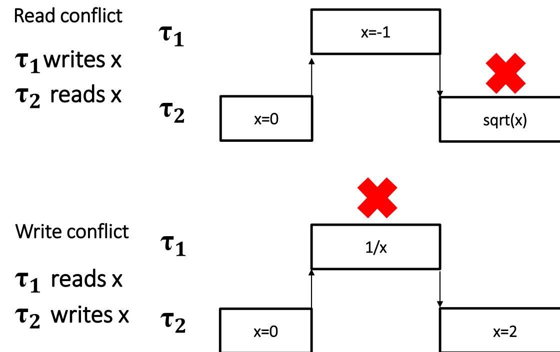


Figure 4.1: Write and read conflicts.

One way to maintain data consistency is restrain simultaneous access to shared resources. This can be achieved using locking mechanisms. A lock is a synchronization mechanism that may be used to enforce data consistency when accessing shared resources by blocking the resource during the execution. Another possible mechanism is based on the use of buffers. For instance in Figure 4.2, the read conflict is avoided by prefetching or buffering all shared data into local variables, in this way, τ_1 can not affect the data changed by τ_1 during the execution. At the end of the execution tau_1 "publish" it changes. The second part of the Figure denotes how the write conflicts can be avoided using locking mechanisms.

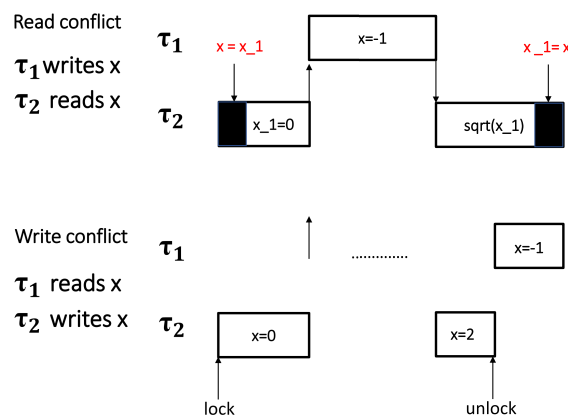


Figure 4.2: Buffer (Implicit) and lock mechanisms.

Both mechanisms have different impact in terms of blocking delay (i.e, the performance decreases) and memory footprint. However, we have to consider that for certain applications the performance or the memory footprint can be trade off for guarantee data consistency. In the next sections we will characterize this impact.

Time determinism. The above mentioned mechanisms (locking or buffering) disregard of time determinism. To better understand this concept, let's suppose that tasks communicates among each other in a producer-consumer fashion. Following this assumption, let us consider the example presented in Figure 4.3, where an effect chain composed of τ_1, τ_2 and τ_3 is shown. Task τ_1 has a runnable writing a shared label that is then read by τ_2; this latter task processes the read variable, and then writes a different shared label, which is then read by a runnable in τ_3. In the end, this runnable outputs an actuation signal that completes the previously introduced, effect chain. In this case, the amount of time that elapses from the first input event until the end of the chain, also known as the end-to-end latency, is 3. If the computation time of some runnables is modified, or more runnables are added as in Figure 4.4, the end-to-end latency may increase (19 for the case in the figure).

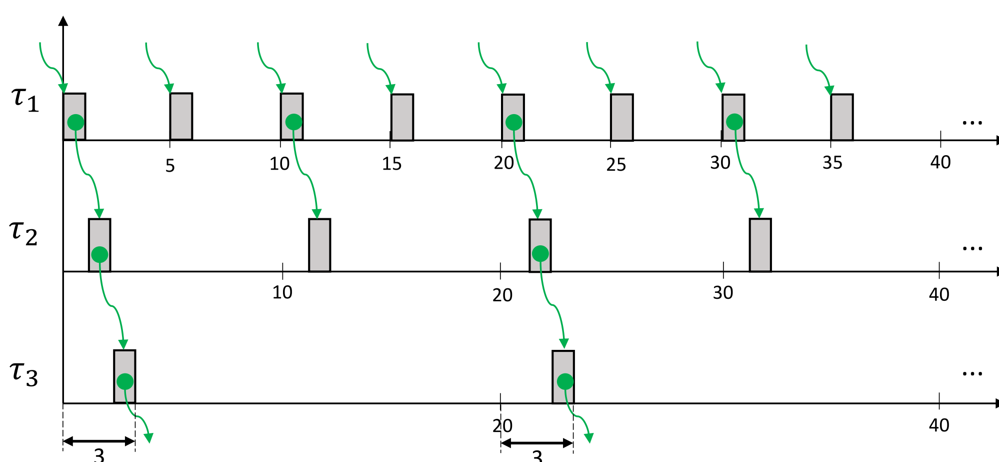


Figure 4.3: End-to-end effect chains composed of three tasks with parameters $T_1 = 5, T_2 = 10, T_3 = 20$ and $C_1 = C_2 = C_3 = 1$.

Control tasks are typically executed periodically, i.e., at a given sampling period. The resulting control performance is highly dependent on task jitter, task response times, scheduling policy and end-to-end latency of effect chains. Even a small change in one of these parameters might be detrimental to control performance, potentially requiring a system redesign, with related additional cost and time.

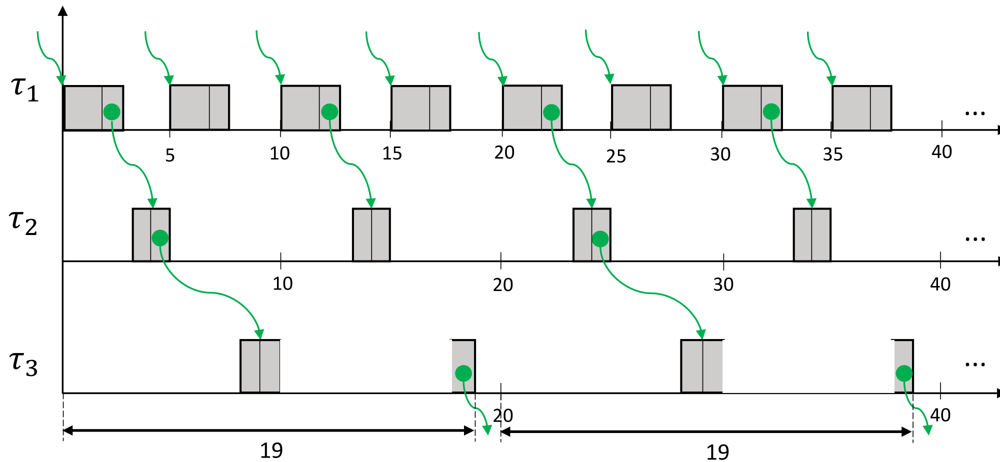


Figure 4.4: End-to-end effect chains composed of three tasks with parameters $T_1 = 5, C_1 = 3, T_2 = 10, C_2 = 2, T_3 = 20$ and $C_3 = 3$.

The LET concept has been introduced in the automotive industry to explicitly address this issue. The LET semantics decouples control algorithms from task jitter, task response times, scheduling policy and hardware dependence, enabling more robust algorithms and more deterministic and predictable systems. The LET model requires that inputs and outputs be logically updated at reading and publishing points, respectively. To see the effect of this paradigm on end-to-end latency, let's apply its semantics to the examples shown in Figure 4.3 and 4.4. The results are shown in Figure 4.11a and 4.11b, where it is easy to see that the age latency is the same in both cases. Clearly, this communication pattern allows not only deterministically setting publishing and reading points, but also setting the age latency of an effect chain to a fixed value, regardless of the actual execution time and core allocation of the involved communicating tasks.

4.2 Related Work

The characterization of end-to-end timing latencies of effect chains between communicating AUTOSAR tasks [1] is an important problem for many automotive applications with tight real-time requirements. An analysis of worst-case latencies along effect chains in critical avionic systems is presented in [76], proposing a mixed integer linear programming (MILP) formulation focusing on end-to-end latency and temporal consistency. In the automotive domain, multiple communication patterns have been proposed, affecting the resulting end-to-end latencies of event chains in different ways. Beside Explicit and

[1]To better understand the background where the considered communication patterns have been proposed, see Section 2.2

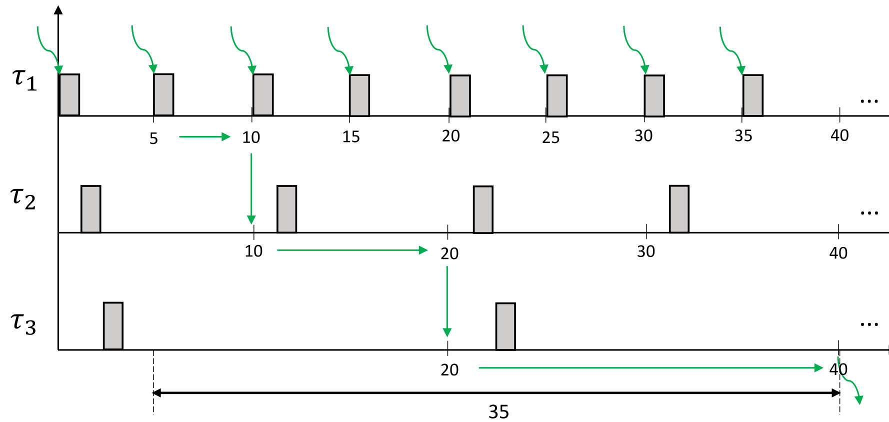


Figure 4.5: End-to-end effect chain with LET composed of three tasks with parameters $T_1 = 5, T_2 = 10, T_3 = 20$ and $C_1 = C_2 = C_3 = 1$.

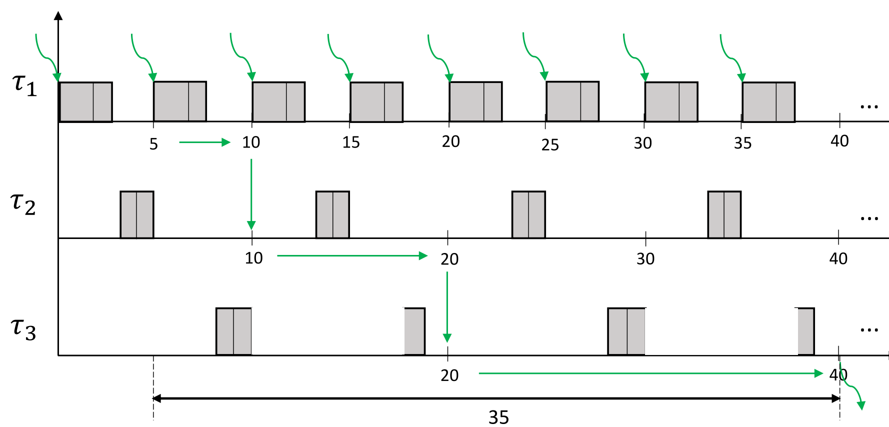


Figure 4.6: End-to-end effect chain with LET composed of three tasks with parameters $T_1 = 3, T_2 = 5, T_3 = 6$ and $C_1 = C_2 = C_3 = 1$.

Implicit communication modes, the Logical Execution Time (LET) paradigm has been proposed within the time-triggered programming language Giotto [10], [77]. This communication semantic allows determining the time it takes from reading program input to writing program output regardless of the actual execution time of a real-time program. As stated in [9], LET evolved from a highly controversial idea to a well-understood principle of real-time programming, motivated by the observation that the relevant behavior of real-time programs is determined by when inputs are read and outputs are written. This concept has been adopted by the automotive industry as a way of introducing determinism in their systems.

In [24], an overview of the different communication patterns used in the automotive domain is provided, highlighting the importance of the end-to-end latencies of effect

chains in an engine management system. Moreover, a method to transform LET and implicit communication into their corresponding direct communication analogues is presented. The impact on end-to-end latencies and communication overheads in terms of temporal determinism and data consistency is shown using the SymTA/S tool[2]. In [78], an end-to-end timing latency analysis for effect chains with specified age-constraints is presented. The analysis is based on deriving all possible data propagation paths. These paths are used to compute minimum and maximum end-to-end latencies of the cause-effect chains. In [79], the analysis is extended including the Implicit communication and the Logical Execution Time paradigms, providing techniques for deriving the maximum data age of cause-effect chains.

To the best of our knowledge, the work proposed in this chapter represents the first complete study that combines three communication patterns: Explicit, Implicit, and LET, with a concise mathematical end-to-end latency timing analysis that encompasses two end-to-end timing semantics, namely Age and Reaction latency, for automotive systems. To that end, we give a tight schedulability and timing analysis of a mixed-preemptive cooperative task setting, that will enable us to provide upper bounds on the end-to-end latency of effect chains in an automotive setting. Detailed considerations are provided concerning the implementation and mathematical models of the aforementioned communication patterns, comparing the resulting latencies of the different semantics against an industrial case study.

4.3 System model, terminology and notation

This section describes the terminology and notation used throughout the following sections considering the constraints defined in the AUTOSAR standard and AMALTHEA model. As we introduced in Section 2.2. The smallest functional entity in AUTOSAR is called *runnable*. A SWC is made up of one or more runnables. Runnables having the same functional period according to the control dynamics are grouped into the same task. In the simplest case, one functionality is realized by means of a single runnable. However, more complex functionalities are typically accomplished using several communicating runnables, possibly distributed over multiple tasks.

The model is assumed to comprise m identical cores, with tasks and runnables statically partitioned to the cores, and no migration support. Each task τ_i is specified by a tuple $(C_i, D_i, T_i, P_i, PT_i)$, where C_i stands for the worst-case execution time (WCET), D_i is the relative deadline, T_i is the period, P_i is the priority, and PT_i defines the type of preemption. Every period T_i, each task releases a job composed of γ_i subsequent runnables, where τ_i^r represents the rth runnable of τ_i, with $1 \leq r \leq \gamma_i$.

[2]https://auto.luxoft.com/uth/timing-analysis-tools/

The worst-case execution time of τ_i^r is denoted as C_i^r. Therefore,

$$C_i = \sum_{k \in [1, \gamma_i]} C_i^k. \tag{4.1}$$

We also denote as $\overline{C}_i^r$ the cumulative execution time of runnables $\tau_i' \ldots, \tau_i^r$, i.e.,

$$\overline{C}_i^r = \sum_{k \in [1, r]} C_i^k. \tag{4.2}$$

Moreover each task is characterized by a worst-case response time R_i as the longest possible time from the time release until the execution completion.

Tasks are scheduled by the operating system based on the assigned (fixed) priorities. The scheduling policy may be either preemptive or cooperative, as specified by PT_i. Preemptive tasks always preempt lower priority tasks, while cooperative tasks preempt a lower priority one only at runnable boundaries. Preemptive tasks are assumed to have always a higher priority than any cooperative task. The mixed cooperative-preemptive nature allows modeling automotive systems where hard real-time tasks (preemptive tasks) co-exist with soft and firm real-time tasks (cooperative tasks), providing the proper balance between preemption latency and context switch overhead according to the needs of each task.

Tasks communicate with one another through shared labels, that abstract a message-passing communication mechanism, typically implemented with a shared memory (see Figure 4.7). Regarding the type of access, a task can be either a sender or a receiver of a label. A sender is a task that writes a label. We assume there is only one sender per label, while there may be multiple receiving tasks reading that label. Even though the microcontroller used for this analysis, AURIX TC38X[3], allows more than one instruction-per-cycle (IPC), the complexity of autmotive software makes the actual average value less than that. We, therefore, consider $IPC = 1$. The execution time of a runnable τ_i^r, without taking memory computation into account, can then be computed as $C_i^r = E_i^r / f$, where E_i^r is a bound on the number of instructions for the considered runnable given by its Weibull distribution, and f is the core frequency. The computational phase is also characterized by a parameter $F_\ell^{R/W}$ that represents the number of times a runnable accesses a label (a.k.a. frequency access in the AMALTHEA model).

Considering the memory constraints of the AMALTHEA model, the time it takes to access a label depends on the memory the label is mapped on to. If the label is allocated to the local memory, the considered task may access it within 1 clock cycle. Otherwise, if the label is in the LRAM of a different core or it is in the GRAM, the task pays an access

[3]https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-safety-joins-performance/aurix-2nd-generation-tc3xx/#!documents

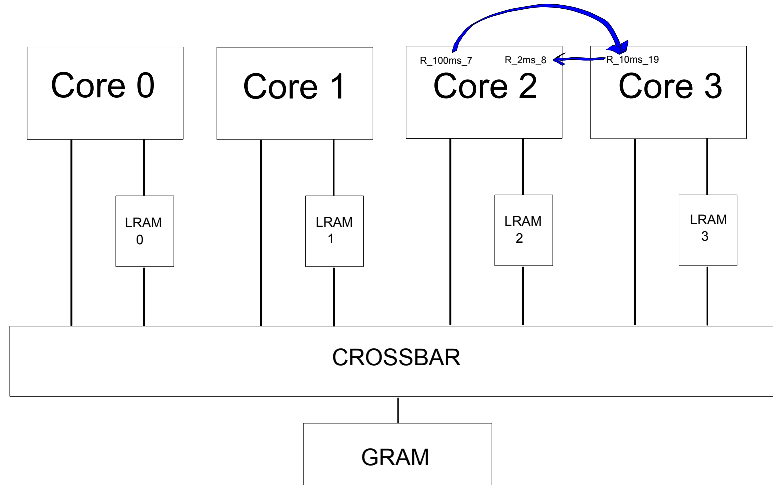


Figure 4.7: Task communication example.

penalty of 8 time units. Since multiple cores may concurrently access the same shared memory, an additional contention penalty is paid when accessing external memories due to the arbitration mechanism. The model assumes a FIFO arbitration, so that a core may wait up to m cycles to obtain access to the addressed memory, where $m = 4$ is the number of cores. Moreover, it is also necessary to obtain L_i^{GRAM}, $L_i^{LRAM_o}$ and $L_i^{LRAM_e}$, where these values represent the number of labels allocated to the global RAM, $GRAM$, local scratchpad, $LRAM_o$, and external scratchpad, $LRAM_e$. These parameters are used to compute the $WCET$ of each runnable. Since reading and writing times are assumed to be equivalent in the model, we denote as $\xi_\ell(x)$ the time it takes to access a shared label ℓ from memory x, where x may be $GRAM$, local LRAM ($LRAM_L$) or external LRAM ($LRAM_E$). In the considered model, we thus have $\xi_\ell(GRAM) = \xi_\ell(LRAM_E) = 8 + m$, and $\xi_\ell(LRAM_L) = 1$.

The overall worst-case execution time C_i^r of runnable τ_i^r can then be derived taking into account also the access time of each label ℓ of τ_i^r,

$$C_i^r = \frac{E_i^r}{f} + \sum_{\ell \in \tau_i^r} \{F_{i,\ell}^r * \xi_\ell\} \tag{4.3}$$

where $F_{i,\ell}^r$ represents the number of times the label ℓ is accessed by runnable τ_i^r, and ξ_ℓ is the time it takes to access ℓ.

In the same way, we can also obtain the best-case execution time (BCET) of runnable τ_i^r, b_i^r, by taking into consideration the minimum number of intructions, e_i^r, given by its Weibull distribution:

$$b_i^r = \frac{e_i^r}{f} + \sum_{\ell \in \tau_i^r} \{F_{i,\ell}^r * \xi_\ell\} \tag{4.4}$$

Thus the best-case start time of runnable τ_i^r, s_i^r, can be computed as the sum of all the BCET of the preceding runnables:

$$s_i^r = \sum_{k \in [1, r-1]} b_i^k. \tag{4.5}$$

4.3.1 Analysis for Preemptive Tasks

According to the considered model, preemptive runnables can only be preempted by higher priority preemptive runnables, and they can always preempt any lower priority task. Therefore, a preemptive task will never experience any blocking delay due to lower priority (preemptive or cooperative) tasks. Hence, the response time for preemptive tasks can be computed adapting the classic response time analysis for arbitrary deadlines presented in [80]. The arbitrary deadline model is used instead of the simpler analysis for constrained deadlines because there are configurations where the response time R_i of a task τ_i may be later than the activation of the subsequent job of the same task, i.e., $R_i > T_i$. Under these conditions, the maximum response time of a task is not necessarily given by the first instance released after the synchronous arrival of all higher priority tasks (also called critical instant), but may be due to later jobs.

For each task τ_i, the analysis requires checking multiple jobs until the end of the level-*i* busy period, i.e., the maximum consecutive amount of time for which a processor is continuously executing tasks of priority P_i or higher. The longest Level-*i* active period (L_i) can be calculated by fixed-point iteration of the following relation, starting with $L_i = C_i$:

$$L_i = \sum_{j: P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{4.6}$$

The number of τ_i's instances to check are therefore:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{4.7}$$

The finishing time of the k-th instance ($k \in [1, K_i]$) of runnable τ_i^r in the level-i busy period can be iteratively computed as

$$f_i^{r,k} = \sum_{j:P_j>P_i} \left\lceil \frac{f_i^{r,k}}{T_j} \right\rceil C_j + (k-1)C_i + \overline{C}_i^r, \quad (4.8)$$

where the first term in the sum accounts for the higher priority interference, the second term accounts for the $(k-1)$ preceding jobs of τ_i, and the last term considers the contribution of the k-th job limited to τ_i^r and its preceding runnables. The response time of the k-th instance of τ_i^r can then be easily found subtracting its arrival time:

$$R_i^{r,k} = f_i^{r,k} - (k-1)T_i. \quad (4.9)$$

The worst-case response time of runnable τ_i^r can be found by taking the maximum among all K_i jobs in the level-i busy period:

$$R_i^r = \max_{k\in[1,K_i]} \{R_i^{r,k}\}. \quad (4.10)$$

Finally, the worst-case response time of task τ_i is computed in the following way:

$$R_i = \sum_{k\in[1,\gamma_i]} R_i^k. \quad (4.11)$$

4.3.2 Analysis for Cooperative Tasks

While the main advantage of preemptive scheduling is real-time response, cooperative scheduling limits the number of preemptions between cooperative tasks, reducing the overhead due to context switches and simplifying re-entrance problems. Moreover, in classic automotive platforms based on single core technologies, cooperative scheduling was a way to provide implicit data consistency at runnable level, avoiding the need for mutual exclusion primitives.

The analysis for cooperative tasks is somewhat more complicated, since it needs to take into account (i) the blocking delays due to lower priority cooperative tasks that can be preempted only at runnable boundaries; (ii) the interference due to higher priority cooperative tasks that can preempt the considered task only at runnable boundaries; (iii) the interference of preemptive tasks that may preempt even within a runnable. To tackle this problem, we will modify and merge the analysis for limited-preemption systems with Fixed Preemption Points (FPP) and for Preemption Threshold Scheduling (PTS), both summarized in [81]. The outcome will be a necessary and sufficient response-time analysis for the considered mixed preemptive-cooperative task model.

Under this model, a preemption threshold is assigned to cooperative tasks. This priority is higher than that of any cooperative task, but lower than that of any preemptive tasks. When a cooperative task τ_i is executing one of its runnables, its nominal priority P_i is raised to the threshold θ_i, so that cooperative tasks cannot preempt it. The nominal priority is restored when the runnable is completed, allowing cooperative preemptions from higher priority tasks.

As with preemptive tasks, it is also necessary to consider multiple jobs within a busy period. However, the busy period must also include the blocking due to lower priority tasks. The longest Level-*i* active period can be calculated adding a blocking factor to the recurring relation of Equation (4.6):

$$L_i = B_i + \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{4.12}$$

Since a task can only be blocked once by lower priority instances, B_i corresponds to the largest execution time among lower priority runnables[4]:

$$B_i = \max_{j,r:P_j < P_i} \{C_j^r\}. \tag{4.13}$$

Equation (4.7) can then be used to compute the number of instances to check in the busy period.

The worst-case starting time $\hat{s}_i^{r,k}$ of the *k*-th instance of runnable τ_i^r can be computed taking into consideration the blocking time B_i, the interference produced by higher priority tasks before $\tau_{i,r}$ can start, the preceding (k-1) instances of τ_i, and the execution time of the preceding runnables of $\tau_{i,r}$:

$$\hat{s}_i^{r,k} = B_i + \sum_{j:P_j > P_i} \left(\left\lfloor \frac{s_i^{r,k}}{T_j} \right\rfloor + 1 \right) C_j + (k-1)C_i + \overline{C}_i^{r-1}. \tag{4.14}$$

The best-case starting time can instead be easily computed as $s_i^{r,k} = \overline{C}_i^{r-1}$. The formula is similar to Equation (4.8), adding the blocking term and subtracting the execution time of the considered runnable since we are considering its starting time.

The worst-case finishing time $f_i^{r,k}$ is calculated by adding to the worst-case starting time $\hat{s}_i^{r,k}$, the execution time of the considered runnable C_i^k, along with the interference of the tasks that can preempt τ_i^r, i.e., the preemptive tasks which have a nominal priority higher than the preemption threshold of any cooperative task. To compute this last

[4]Since the lower priority task must have already arrived before the critical instant, the actual blocking term is actually an infinitesimal amount smaller. We neglect infinitesimal amounts to simplify the formula.

interfering term, we compute the higher priority instances that arrive from the critical instant until the finishing time, and subtract those that arrived before the starting time:

$$f_i^{r,k} = \hat{s}_i^{r,k} + C_i^r + \sum_{j:P_j>\theta_i} \left(\left\lceil \frac{f_i^{r,k}}{T_j} \right\rceil - \left(\left\lfloor \frac{\hat{s}_i^{r,k}}{T_j} \right\rfloor + 1 \right) \right) C_j. \tag{4.15}$$

Equation (4.10) and (4.11) can then be identically used to compute the worst-case response time of the considered runnable and task, respectively.

4.4 Inter task communication

In line with the multi-core complexity trend, automotive applications are evolving towards more complicated task and runnable settings. As tasks communicate across the memory hierarchy, data consistency problems may arise. On the other hand, as control algorithms need deterministic timing, non-deterministic behavior, such as task jitter, might cause different levels of control performance degradation that might even lead to system instability. Thus, distinct communication patterns have been proposed in order to provide different levels of determinism and consistency: (i) Explicit, (ii) Implicit and (iii) Logical Execution Time (LET).

1. Explicit communication means that a runnable makes an explicit RTE API call in order to directly write or read labels, i.e. a runnable or a task may have unrestricted access to variables at any point during its execution (see Figure 4.8). To avoid data inconsistency issues, accesses must be protected through explicit synchronization or locking constructs. A memory-aware analysis of Explicit communication is proposed in [82].

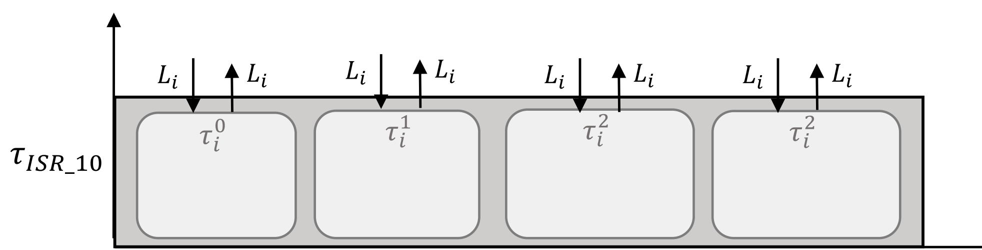


Figure 4.8: Explicit communication example.

2. Implicit communication aims at data consistency and defines two kinds of operations: Implicit Read and Implicit Write. The former implies that if a runnable reads a label, a copy of this label, instead of the original, should become available

for the runnable, when it starts at the latest. The RTE ensures that this copy does not change during the execution of the runnable. Implicit write means that a label modified by a runnable should be made available to other runnables at the earliest when the runnable execution is over. The RTE makes sure that this update is done by means of a copy mechanism. One way to implement this is that tasks accessing shared labels work on task-local copies instead of the original labels. To avoid data inconsistency, each task instance performs a copy of the required labels at the beginning of its execution. After working on local copies in an exclusive way, it then publishes its results at the end of its execution. If needed, extra buffers or locking support might be used. The latter is only required at the beginning of a reading task, and at the end of a writing task, which signifies a much lighter synchronization overhead than in the explicit model.

3. As mentioned above, Logical Execution Time (LET) is a hard real-time programming abstraction that was introduced by Giotto programming model. As the relevant behavior of real-time tasks is determined by when inputs are read and outputs are written, the LET semantics requires that inputs and outputs be logically updated at the beginning and at the end of the so called *communication interval*, i.e., in correspondence to the release times of the communicating tasks. This allows deterministically fixing the time it takes from reading an input to writing an output regardless of the actual response time of the involved communicating tasks.

The LET implementation we consider in this dissertation adopts a lock-free paradigm that tries to closely resemble the above-mentioned behavior at the cost of a slightly higher use of local buffers, as will be shown in Section 4.4.2. An implementation of the Implicit communication pattern that tries to duplicate the behavior of this communication is presented in the follwoing section.

4.4.1 Implicit communication

Let I_i and O_i be the set of all shared labels read and written by tasks τ_i, respectively. I_i and O_i therefore represent the inputs and outputs of the considered task. Our implementation for Implicit communication assumes any task τ_i accessing a shared label works on a copy instead of the original label. Copies are created, statically allocated to the task-local scratchpad and inserted in runnables at compile time. Furthermore, two task-specific runnables τ_i^0 and $\tau_i^{(\gamma_i+1)}$ (also called τ_i^{last} for simplicity) are to be inserted at the beginning and at the end of the task. Runnable τ_i^0 is responsible of reading shared labels to the local copies, while τ_i^{last} will write the local copies to the corresponding shared variables. If $\dot{I}_i$ and $\dot{O}_i$ represent the set of τ_i-local copies of the labels contained in I_i and

O_i, respectively, runnable τ_i^0 updates $\dot{I}_i$, whereas runnable τ_i^{last} publishes its updates by writing $\dot{O}_i$ to the corresponding shared variables in O_i. See Figure 4.9. Observe that if a task writes and reads the same label, only one copy is created.

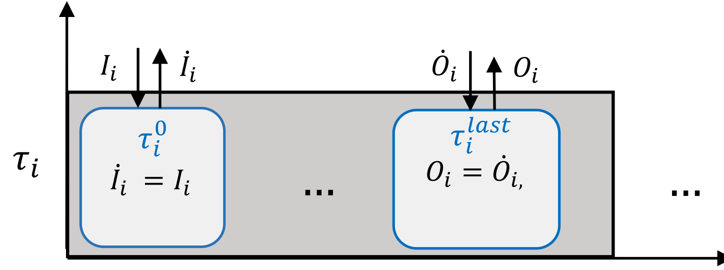


Figure 4.9: Implicit communication implementation.

For example, suppose a task τ_i reads shared label L_1 and writes to shared label L_2. Let $L_{i,1}$ and $L_{i,2}$ represent the τ_i-local copies of L_1 and L_2 respectively. This model dictates that $L_{i,1}$ is to be updated by runnable τ_i^0 at the beginning of task τ_i. After that, τ_i reads $L_{i,1}$ and writes to $L_{i,2}$, never accessing the original labels L_1 and L_2. In the end, runnable τ_i^{last} writes the latest value of $L_{i,2}$ to L_2. It does not need to publish L_1, since it did not modify it. See Figure 4.10.

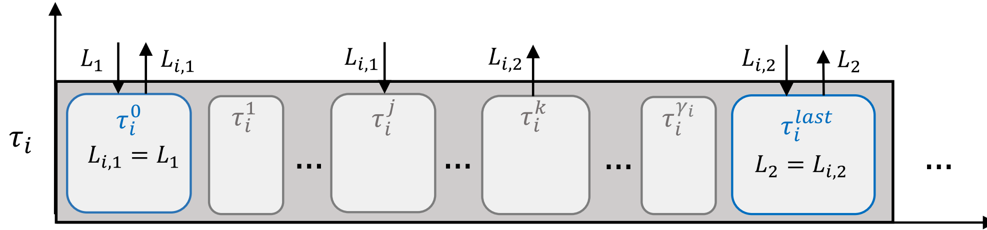


Figure 4.10: Implicit communication example.

An upper bound on the overhead introduced by the copy-in (τ_i^0) and copy-out (τ_i^{last}) runnables can be easily computed as

$$C_i^0 = \sum_{\ell \in I_i} \xi_\ell, \tag{4.16}$$

and

$$C_i^{last} = \sum_{\ell \in O_i} \xi_\ell, \tag{4.17}$$

where the sum is extended over all shared labels read (resp. written) by the considered task τ_i. The total execution time of τ_i is computed as

$$C_i = C_i^0 + C_i^{last} + \sum_{r \in [1,\gamma_i]} C_i^r, \tag{4.18}$$

where the execution time of a runnable can be expressed as

$$C_i^r = \frac{E_i^r}{f} + \xi \sum_{\ell \in \tau_i} F_{i,\ell}^r, \tag{4.19}$$

considering a fixed cost ξ for each one of the $F_{i,\ell}^r$ accesses by the considered runnable to the local memory, whether they be to a label only accessed by this task, or to a local copy of a shared label.

The additional memory occupancy in the Implicit model is given by the local copies created for shared labels, i.e., all labels in $I_i \cup O_i$ for all tasks τ_i.

4.4.2 LET communication

Differently from the Implicit case, LET enforces task communications at deterministic times, corresponding to task activation times. In our implementation, each reader creates one or more local copies of the shared label. Since the considered model allows just one writer task for each label, the writer task is allowed to directly modify the original label, updating the readers copies at well-determined times.

We hereafter consider the communication between the writer and one of the readers. Assume the writer has period $T_W = 2$ and the reader $T_R = 5$, as in Figure 4.11a: while τ_W may repeatedly write the considered label L, these updates are not visible to the concurrently executing reader, until a *publishing point* $P_{W,R}^n$, where the value is updated for the next reader instance. This point corresponds to the first upcoming writer release that directly precedes a reader release, i.e., where no other write release appears before the arrival of the following reader instance. We call *publishing instance* the writing instance that updates the shared value for the next reading instance, i.e., the writer's job that directly precedes a publishing point. Note that not all writing instance are publishing instances. See Figure 4.11a, where publishing instances are marked in bold red.

It is also convenient to define *reading points* $Q_{R,W}^n$, which correspond to the arrival of the reading instance that will first use the new data published in the preceding publishing point $P_{R,W}^n$. Figure 4.11a & 4.11b shows publishing and reading points for a case where the writer task has a smaller (a) or larger (b) period than the reader task.

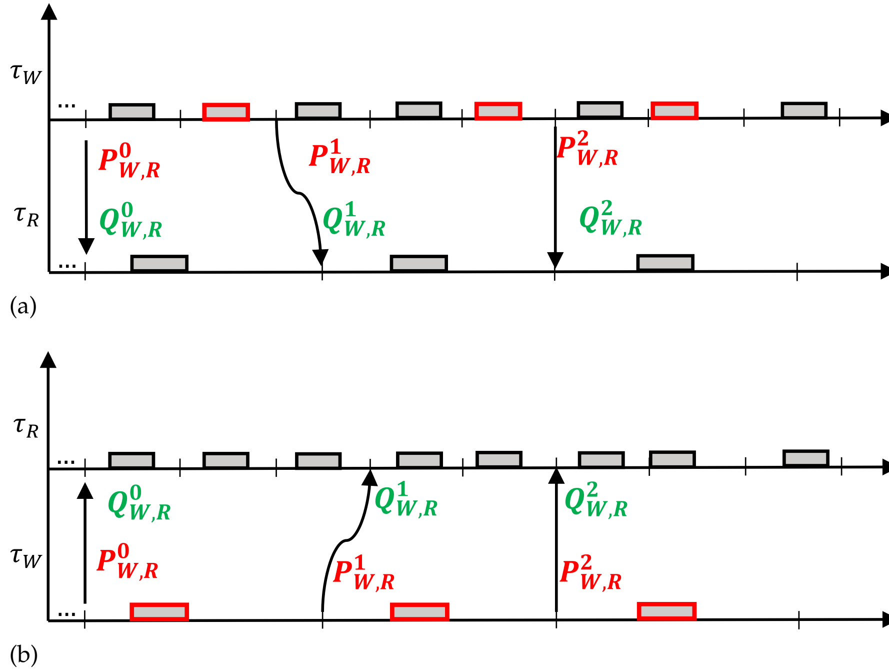


Figure 4.11: Publishing and reading points when the reader has a larger (a) or smaller (b) period than the writer.

We define the *hyperperiod* of two communicating tasks as the least common multiple *LCM* of their periods. The publishing and reading points of two communicating tasks can be computed as a function of their periods, as shown in the next theorem.

Theorem 1. *Given two communicating tasks τ_W and τ_R, the publishing and the reading points can be computed as*

$$P^n_{W,R} = \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W, \quad \forall n \in [0, n_{W,R}], \tag{4.20}$$

$$Q^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R, \quad \forall n \in [0, n_{W,R}], \tag{4.21}$$

where $T_{\max} = \max(T_W, T_R)$ and $n_{W,R}$ is the number of jobs released in a hyperperiod by the task with the longest period, i.e.,

$$n_{W,R} = \frac{LCM(T_W, T_R)}{T_{\max}} = n_{R,W}. \tag{4.22}$$

Proof. If the writer τ_W has a smaller or equal period than the reader τ_R, i.e., $T_W \leq T_R$ as in Figure 4.11a, there is one publishing and one reading point for each *reading* instance.

There are $LCM(\tau_W, \tau_R)/\tau_R = n_{W,R}$ such instances. Reading points trivially correspond to each reading task release, i.e.,

$$Q^n_{W,R} = nT_R, \quad \forall n \in [0, n_{W,R}],$$

while publishing points correspond to the last writer release before such a reading instance, i.e.,

$$P^n_{W,R} = \left\lfloor \frac{nT_R}{T_W} \right\rfloor T_W, \quad \forall n \in [0, n_{W,R}].$$

Otherwise, when the writer τ_W has a larger period than the reader τ_R, i.e., $T_W \geq T_R$ as in Figure 4.11b, there is one publishing and one reading point for each *writing* instance. Again, there are $\frac{LCM(T_W, T_R)}{T_W} = n_{w,r}$ such instances. Publishing points trivially correspond to each writing task release, i.e.,

$$P^n_{W,R} = nT_W, \quad \forall n \in [0, n_{W,R}],$$

while reading points correspond to the last reader release before such a writing instance, i.e.,

$$Q^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R, \quad \forall n \in [0, n_{W,R}],$$

It is easy to see that, in both cases $T_W \leq T_R$ and $T_W \geq T_R$, the formula for $P^n_{W,R}$ and $Q^n_{W,R}$ are generalized by Equations (4.20) and (4.21). Note that, when $T_W = T_R$, $P^n_{W,R} = Q^n_{W,R} = nT_W$. □

Let $I_{W,R}$ denote the set of labels written by τ_W and read by τ_R. For each of these labels, the reading task τ_R creates a local copy to which it has exclusive access. Let $\dot{I}_{W,R}$ denote the set of τ_R-local copies of the labels contained in $I_{W,R}$. A communication-specific runnable is to be inserted to update $\dot{I}_{i,j}$ at the end of the communication period, i.e., by the latest completing task before a publishing point.

In the automotive domain we can easily make a distinction between different kinds of tasks that co-exists in the same ECU following different activation patterns, for instance, periodic tasks (classified into harmonic and non harmonic tasks), sporadic tasks, Interrupt Service Routine (ISR) tasks or adaptive variable-rate (AVR) tasks [83]. In this work we considered the cases of (i) harmonic synchronous communication and (ii) non-harmonic synchronous communication. The asynchronous case will be considered as future work.

Harmonic Synchronous Communication (HSC)

Two communicating tasks τ_W and τ_R have harmonic periods if the period of one of them is an integer multiple of the other. When a harmonic synchronous communication (HSC) is established, the following relations hold: $LCM(T_W, T_R) = T_{\max}$, $n_{W,R} = n_{R,W} = 1$ and $P^n_{W,R} = Q^n_{W,R} = nT_{\max}$, i.e., publishing and reading points are integer multiples of the largest period of the communicating tasks.

Consider the example in Figure 4.12, where two tasks τ_l and τ_s, with $T_l/T_s = 2$, both read shared labels L_1 and L_2. Moreover, τ_l writes to L_1, while τ_s writes to L_2. The proposal suggests that τ_s and τ_l are to read $L_{s,1}$ and $L_{l,2}$ instead of the original labels. Notice that τ_l and τ_s directly write to L_1 and L_2. These copies are to be updated by either runnable τ_s^{last} or runnable τ_l^{last} depending on whichever job finishes last before the next publishing point. In other words, the responsibility to update the copies is given either to the reader or to the writer, depending on which one completes last in the communication interval. The first reader instance after the publishing point is the first one that accesses the updated value. Such a value will be used by all reading instances until the next reading point.

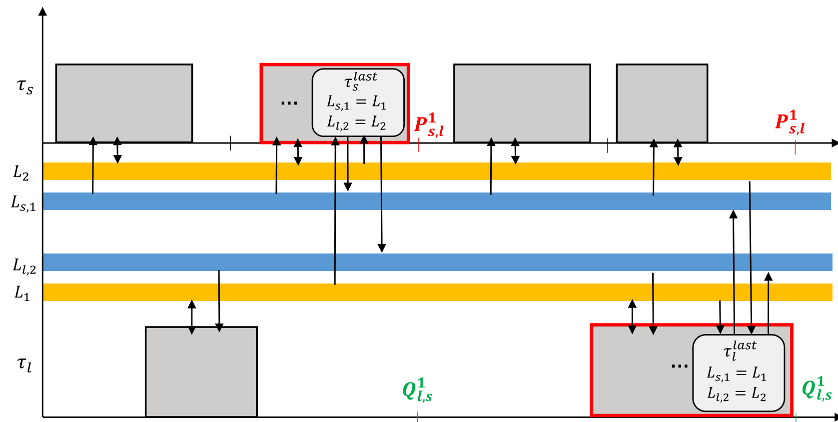


Figure 4.12: LET harmonic communication.

Unlike the Implicit communication, only one task pays the overhead for maintaining the determinism in the communication. Assuming such a task is τ_i, its worst-case execution time can be computed as

$$C_i^{total} = \sum_{k\in[1,\gamma_i]} C_i^k + \sum_{\ell\in I_i\cup O_i} \xi_\ell(x), \tag{4.23}$$

where τ_i is assumed to update all its shared labels. Better estimations are possible considering which task effectively finishes last in each communication period, making

the analysis significantly more complex. The additional memory occupancy is given by the local copies created for shared labels, i.e., all labels in I_i for all tasks τ_i.

Non-Harmonic Synchronous Communication (NHSC)

When two communicating tasks do not have harmonic periods, a non-harmonic synchronous communication (NHSC) is established. The general formulas of Section 4.4.2 apply.

Like in the HSC case, the reading task of a shared label accesses a local copy instead of the original label. However, due to the misaligned activations of the communicating tasks, at least two copies of the same shared label are needed in a NHSC. A task-specific runnable is to be inserted at the end of the writer in order to update the copies of $I_{W,R}$ before the publishing point. If only one copy was used, a task could be writing it while the reader is reading it, leading to an inconsistent state. With two copies, instead, a reader reads a local copy, while the writer may freely write a new value for the next reading instance in a different buffer.

For example, consider a reading task τ_R and a writing task τ_W communicating through a shared variable L_2, with $2T_R = 5T_W$ as in Figure 4.13. There are two τ_R-local copies, $L_{R,2,1}$ and $L_{R,2,2}$, of the shared label L_2. The reading task τ_R reads from one of these copies instead of the original label. These copies are to be updated by the last runnable τ_W^{last} of the writing task. Note that τ_W directly writes to L_2 instead of a local copy.

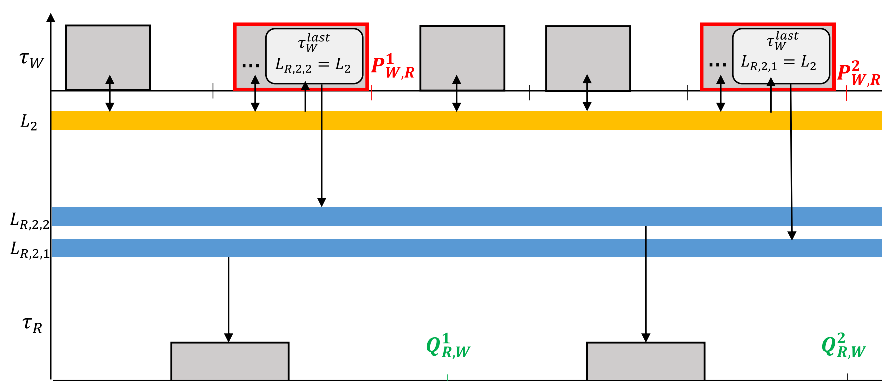


Figure 4.13: NHSC: $2T_R = 5T_R$.

There might also be cases where three copies per labels are needed in order to fulfill the LET determinism. Consider Figure 4.14 where $5T_R = 2T_W$. Note that τ_W may directly access L_1, while τ_R reads from one of the three copies $L_{R,1,1}$, $L_{R,1,2}$ or $L_{R,1,3}$, which are to be updated by runnable τ_W^{last}. An extra copy of L_1 is needed because the value

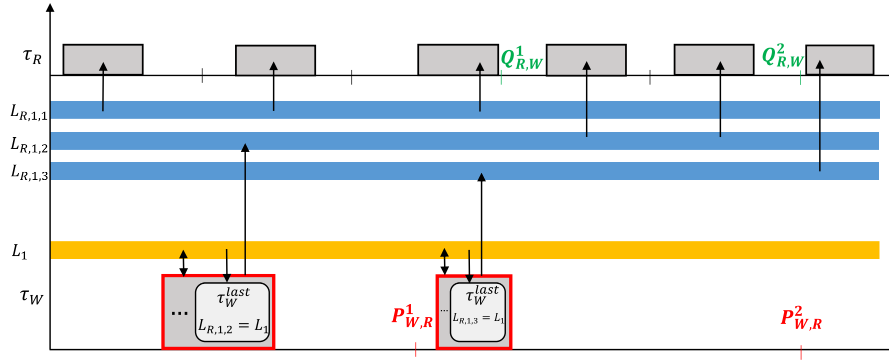


Figure 4.14: NHSC: $5T_R = 2T_R$.

computed by the second writing instance may be available either before or after the next reading point $Q^1_{R,W}$, depending on the response time of τ_W. If the second instance of τ_W finishes before (resp. after) $Q^1_{R,W}$, the reading instance after $Q^1_{R,W}$ would read the data of the second (resp. first) writing instance. Therefore, the value read at $Q^1_{R,W}$ is not deterministic, as it might correspond either to the first or to the second writing instance. Introducing a third buffer allows obtaining a deterministic behavior, as desired with the LET semantics, where the values published by the first and second writing instances are always read at $Q^1_{R,W}$ and $Q^2_{R,W}$, respectively.

In general, this happens when a publishing instance has a best-case finishing time that precedes the next reading point. Let us define $w^n_{W,R}$ as the window of time between a publishing point $P^n_{W,R}$ and the next reading point $Q^n_{W,R}$. Then, using Equations (4.20) and (4.21),

$$w^n_{W,R} = Q^n_{W,R} - P^n_{W,R} = \left\lceil \frac{nT_{\max}}{T_R} \right\rceil T_R - \left\lfloor \frac{nT_{\max}}{T_W} \right\rfloor T_W, \quad \forall n \in [0, n_{W,R}]. \tag{4.24}$$

It is worth pointing out that if a *HSC* is established, then $w^n_{W,R} = 0$. Furthermore, if the best-case response time of a publishing instance is smaller than the corresponding $w^n_{W,R}$, a third buffer is needed to store the new value. Depending on the above condition, the additional memory occupancy due to the local copies is two or three times the size of the labels in I_i for all tasks τ_i.

4.5 End-To-End latency characterization

In this section, we propose a method for computing the end-to-end propagation delay of effect chains taking into consideration different communication patterns.

An effect chain is a producer/consumer relationship between runnables working on shared labels. Effects chains are assumed to be triggered by an event or a task release. The first task in the chain produces an output (i.e., writes to a shared label) for another task following in the event chain. The second task reads the shared label to write an output to a different shared label, which may be then read by a third task, and so on. When the last task produces its final output, the event chain is over.

In [84], four different end-to-end timing semantics are described to characterize the timing delays of effect chains given by multi-rate tasks communicating by means of shared variables. Depending on the application requirements, different end-to-end delay metrics can be of interest. Control systems driving external actuators are interested in the "age" of an input data, i.e., for how long a given sensor data will be used to take actuation decisions. For example, how long a radar or camera frame will be used as a valid reference by a localization or object detection system to perceive the environment: the older the frame, the less precise is the system. Similar considerations are valid for an engine control or a fuel injection system, where correct actuation decisions depend on the "freshness" of sensed data.

Another metric of interest is the "reaction" latency to a change of the input, i.e., how long will it take for the system to react on a new sensed data. Multiple body and chassis automotive applications are concerned with this metric. For example, for a door locking system, it is important to know the time it takes to effectively lock the doors after receiving the corresponding signal.

To more formally characterize age and reaction latencies, consider Figure 4.15, showing an event chain triggered by a periodic sensor (upwards black arrows). The upper task reads the sensor data, elaborates it, and shares the result with the next task. And so on, until the end of the event chain. Green arrows denote when an input is propagated to the next task. In this case, we call it a *valid* input. Red arrows correspond to elaborations that are not propagated, also called *invalid* inputs, because they are overwritten before being read by the next task in the chain. The *age latency* is defined as the delay between a valid sensor input until the last output related to this input in the event chain. The *reaction latency* is defined as the delay between a valid sensor input until the first output of the event chain that reflects such an input. It measures how much time it takes for a new event to propagate to the end of the event chain. Depending on tasks alignments, the reaction latency may significantly vary. In Figure 4.16, the first sensor input arrives just a bit after the runnable that is responsible of elaborating it (marked as a green dot in the first job of τ_i). This causes the reaction latency to increase substantially, as the output task τ_k will continue working with an older input for three further jobs (marked as A, B and C in the figure).

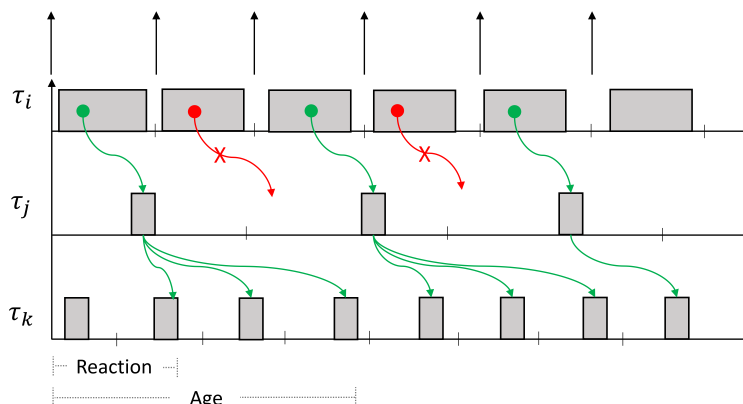


Figure 4.15: Age semantics

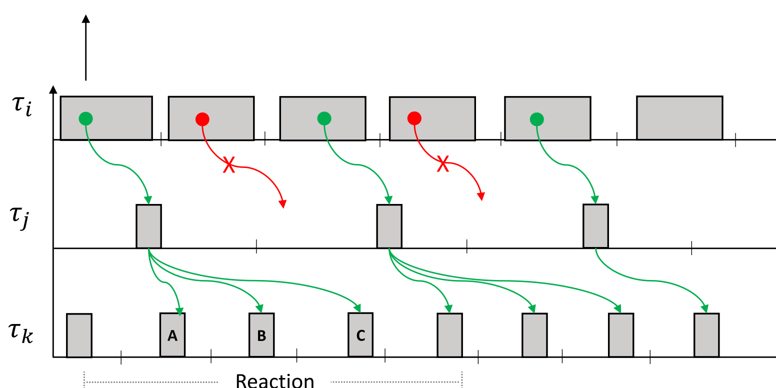


Figure 4.16: Reaction semantics

In [84], age and reaction latencies are also referred to as last-to-last (L2L) and first-to-first (F2F) delay, respectively. However, no method is presented to formally compute these metrics.

Before computing end-to-end age and reaction latencies of an effect chain, we first compute the maximum delay ϕ_i^r between two operations on the same variable executed by two consecutive instances of the same runnable τ_i^r. In Figure 4.17, ϕ_i^r is derived as a function of the best-case start time s_i^r and the worst-case response time R_i^r of runnable τ_i^r:

$$\phi_i^r = T_i - s_i^r + R_i^r - (\varepsilon_1 + \varepsilon_2). \tag{4.25}$$

Where ε_1 (resp. ε_2) stands for the time between s_1^r (resp. R_i^r) and the first (resp. last) operation on the label performed by the runnable. Assuming the first and the second

runnable instance access the shared label at the beginning and at the end of their execution, respectively, it follows $\varepsilon_1 = \varepsilon_2 = 0$, and

$$\phi_i^r = T_i - s_i^r + R_i^r. \tag{4.26}$$

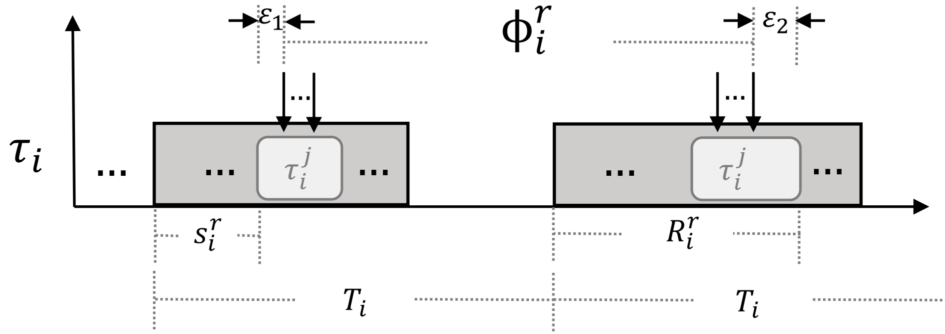


Figure 4.17: Calculation of ϕ_i^r

In the following, we compute age and reaction latencies for the considered communication patterns. From now on, downwards (resp. upwards) black arrows mean read (resp. write) operations. We first examine the Explicit communication in detail, since it establishes the basis for the latency characterization of its Implicit counterpart.

4.5.1 Explicit Communication

Consider an effect sub-chain, where a runnable τ_W^i writes to a label L, which is in turn read by another runnable τ_R^j. We hereafter compute the worst-case sub-chain age latency $\alpha_{W,R}^{i,j}$ and worst-case sub-chain reaction latency $\delta_{W,R}^{i,j}$. To do this, we consider different worst-case settings where the following conditions hold:

C1. τ_W^i stores L right after τ_R^j started loading it.

C2. Two subsequent read operations are ϕ_R^j time-units apart.

C3. Two subsequent write operations are ϕ_W^i time-units apart.

Theorem 2. *The worst-case sub-chain age latency of two communicating tasks τ_W and τ_R is*

$$\alpha_{W,R}^{i,j} = \phi_W^i \tag{4.27}$$

Proof. To compute the age latency $\alpha_{W,R}^{i,j}$, we separately consider the cases with $\phi_R^j \geq \phi_W^i$ and $\phi_R^j < \phi_W^i$. When $\phi_R^j \geq \phi_W^i$, the worst-case situation is that of Figure 4.18a, where $\alpha_{W,R}^{i,j} = \phi_W^i$. Shifting the reading instance of τ_R earlier would cause a proportional

decrement in the age latency, while postponing it right after the second update of τ_W would cause a sudden drop of the age latency to zero, as the read would refer to the new writing update. When instead $\phi_R^j < \phi_W^i$, the worst-case scenario is that of Figure 4.18b, where the latest instance of τ_R reads just before the next update of τ_W. In this case, the age latency is ϕ_W^i. Shifting the reading instance to the left would proportionally decrement the age latency, whereas postponing it right after the update would decrease the age latency by one reading period.

In both considered cases, the age latency is $\alpha_{W,R}^{i,j} = \phi_W^i$, proving the theorem. □

Theorem 3. *The worst-case sub-chain reaction latency of two communicating tasks τ_W and τ_R is*

$$\delta_{W,R}^{i,j} = \phi_R^j \tag{4.28}$$

Proof. To compute the reaction latency $\delta_{W,R}^{i,j}$, we again separately consider the cases with $\phi_R^j \geq \phi_W^i$ and $\phi_R^j < \phi_W^i$. When $\phi_R^j \geq \phi_W^i$, the worst-case situation is shown in Figure 4.19a, where the reaction latency is equal to ϕ_R^j. Shifting earlier the reading instance would cause a proportional decrease of the reaction latency, while moving it later wold make it refer to the last write update, leading to a null reaction latency. Note that earlier writing instances within the considered window do not need to be considered for the reaction latency because they are overwritten, i.e., they do not cause any "reaction" in the system. When instead $\phi_R^j < \phi_W^i$, the worst-case scenario is that of Figure 4.19b, where $\delta_{W,R}^{i,j} = \phi_R^j$. Shifting the writing instance to the right would cause a proportional decrement in the reaction latency, while moving it a bit earlier would cause a sudden drop of the reaction latency to zero.

In both considered cases, the reaction latency is $\delta_{W,R}^{i,j} = \phi_R^j$, proving the theorem. □

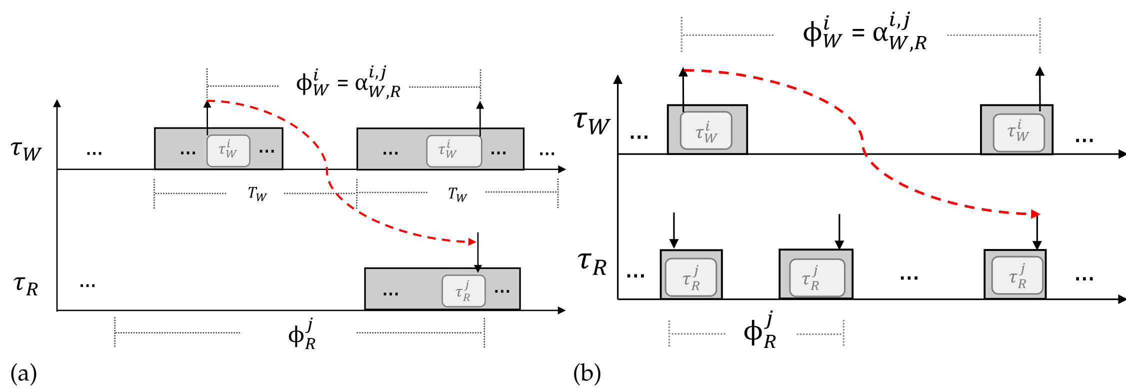


Figure 4.18: Worst-case sub-chain age latency $\alpha_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$ (a) and $\phi_R^j < \phi_W^i$ (b).

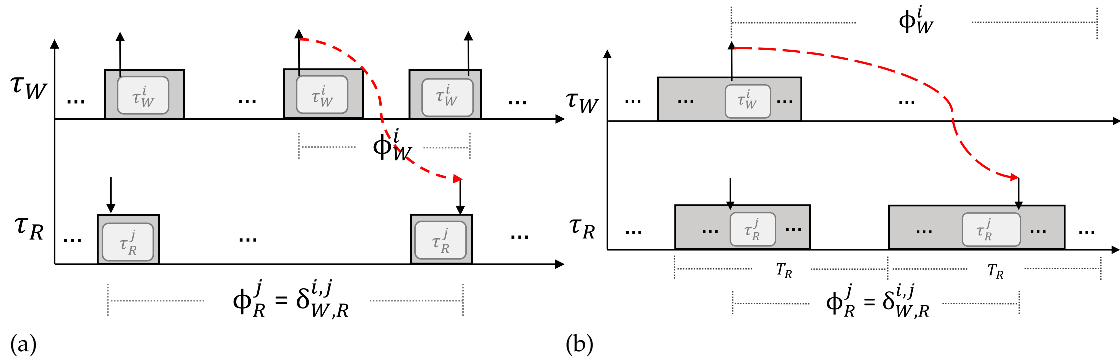


Figure 4.19: Worst case sub-chain reaction latency $\delta_{W,R}^{i,j}$ when $\phi_R^j \geq \phi_W^i$ (a) and $\phi_R^j < \phi_W^i$ (b).

For simplicity, we will drop the apexes on $\delta_{W,R}^{i,j}$, $\alpha_{W,R}^{i,j}$, ϕ_W^i and ϕ_R^j when we do not need to explicitly refer to the communicating tasks. An upper bound on the overall end-to-end age latency of an effect chain $\alpha(EC)$ can therefore be computed as

$$\alpha(EC) = \sum_{h=0}^{\eta-1} \alpha_{h,h+1} = \sum_{h=0}^{\eta-1} \phi_h, \tag{4.29}$$

where η is the number of tasks constituting the effect chain EC.

Similarly, an upper bound on the overall end-to-end reaction latency of an effect chain $\delta(EC)$ can be computed as:

$$\delta(EC) = \sum_{h=0}^{\eta-1} \delta_{h,h+1} = \sum_{h=1}^{\eta} \phi_h. \tag{4.30}$$

4.5.2 Implicit Communication

As explained in Section 4.4.1, our Implicit communication model introduces two extras runnables at task boundaries in charge of reading and publishing the shared labels. From an end-to-end latency perspective, the Implicit communication can be considered as a particular case of its Explicit counterpart, considering τ_W^{last} and τ_R^0 as writing and reading runnables, respectively. For instance, the worst-case sub-chain propagation delay $\delta_{W,R}^{i,j}$ for any pair of communicating runnables τ_W^i and τ_R^j is equal to $\delta_{W,R}^{last,0}$, plus an extra delay Δ_R due to the fact that τ_R publishes all its shared labels at the end of its execution. For any task τ_i, ϕ_i^0 and ϕ_i^{last} can be calculated as

$$\phi_i^0 = T_i + R_i^0. \tag{4.31}$$

$$\phi_i^{last} = T_i - s_i^{last} + R_i \tag{4.32}$$

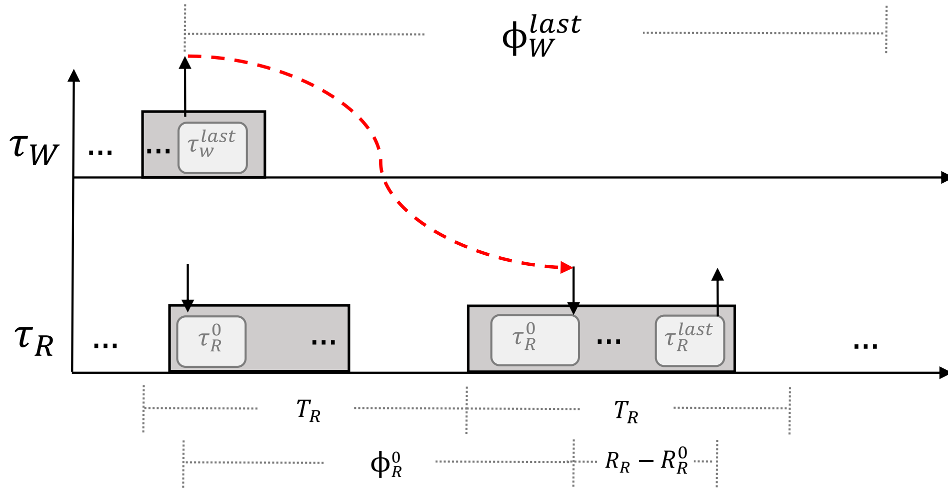


Figure 4.20: Worst-case sub-chain reaction latency ($\phi_R^0 < \phi_W^{last}$) for the Implicit communication.

Figure 4.20 shows the worst-case sub-chain reaction latency with $\phi_R^0 < \phi_W^{last}$. It is easy to see that $\Delta_R = R_R - R_R^0$. A similar situation has been verified to happen in all other possible settings. Sub-chain age and reaction latencies in the implicit case can then be simply computed adding Δ_R to the corresponding explicit counterparts given by Equation (4.27) and (4.28):

$$\alpha_{W,R}^{i,j} = \alpha_{W,R}^{last,0} + \Delta_R = \phi_W^{last} + \Delta_R \tag{4.33}$$

$$\delta_{W,R}^{i,j} = \delta_{W,R}^{last,0} + \Delta_R = \phi_R^0 + \Delta_R. \tag{4.34}$$

An upper bound on the overall end-to-end age and reaction latency can then be computed as

$$\alpha(EC) = \sum_{h=0}^{\eta-1} \alpha_{h,h+1} = \sum_{h=0}^{\eta-1} (\phi_h^{last} + \Delta_{h+1}) \tag{4.35}$$

$$\delta(EC) = \sum_{h=0}^{\eta-1} \delta_{h,h+1} = \sum_{h=1}^{\eta} (\phi_h^0 + \Delta_h). \tag{4.36}$$

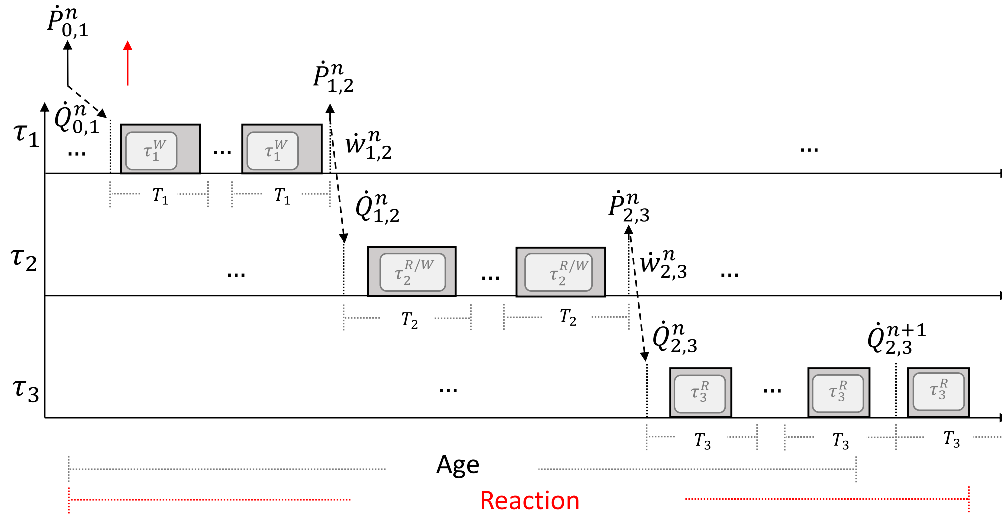


Figure 4.21: End-To-End latency characterization of the LET communication.

4.5.3 LET Communication

If we define the hyperperiod of an *EC*, H_{EC}, as the *LCM* of the periods of the tasks composing the chain, i.e. $H_{EC} = LCM^{\eta}_{i=1}(T_i)$, then there is a fixed number of possible communication paths in a hyperperiod, starting from the end of the period of the first task (the starting points must be different) and finishing with the release of the last one in the *EC*. We call these chains *basic paths*. Note that if all tasks in the *EC* have harmonic periods then there is only one basic path. We also extend the domain of equations (4.20), (4.21), and (4.24) to $\mathbb{N}$. In order to calculate the length of the *n-th basic path* we need to obtain the first publishing point, $\dot{P}^n_{1,2}$, and last reading point, $\dot{Q}^n_{\eta-1,\eta}$, composing this path (Note that $\dot{P}^n_{1,2} = P^n_{1,2}$ and $\dot{Q}^n_{\eta-1,\eta} = Q^n_{\eta-1,\eta}$ is not necessarily true, see Section 4.6). For this purpose, given two communicating tasks, τ_i and τ_j, that form part of an *EC*, let $n_{i,j}$ denote the number of finished jobs released in the hyperperiod of the *EC* by the task with the longest period in the pair, i.e. $n_{i,j} = \frac{H_{EC}}{max(T_i,T_j)}$. Let us also define P and Q as the arrays containing the publishing points $< P^0_{j,k}, P^1_{j,k}, ..., P^{n_{j,k}}_{j,k} >$ and the reading points $< Q^0_{i,j}, Q^1_{i,j}, ..., Q^{n_{i,j}}_{i,j} >$ respectively. Thus, ***find-publishing-point*** returns the publishing point in P, $\dot{P}^m_{j,k}$, that corresponds to a given reading point, $\dot{Q}^m_{i,j}$. Similarly, ***find-reading-point*** returns the reading point in Q, $\dot{Q}^m_{i,j}$, that corresponds to a given publishing point, $\dot{P}^m_{j,k}$. See Figure 4.21 for an example of a 3-task EC. By applying the combination of both algorithms 0 to every pair of consecutive tasks composing the *EC* and then getting rid of paths having the same starting points, the length θ^n_{EC} of the n-th basic path of the *EC* can be computed as:

$$\theta^n_{EC} = \dot{Q}^n_{\eta-1,\eta} - \dot{P}^n_{0,1} \tag{4.37}$$

Table 4.1: *Find-Publishing-Point* Algorithm (left) *Find-Reading-Point* Algorithm (right)

Find-Publishing-Point	Find-Reading-Point
1: **procedure** FIND-PUBLISHING-POINT($\dot{Q}^m_{j,k}$, P)	1: **procedure** FIND-READING-POINT($\dot{P}^m_{j,k}$, Q)
2: $d \leftarrow \dot{Q}^m_{j,k} - P[l]$	2: $d \leftarrow \dot{P}^m_{j,k} - Q[l]$
3: $l \leftarrow 0$	3: $l \leftarrow 0$
4: **while** $l \leq n_{j,k}$ and $d \geq 0$ **do**	4: **while** $l \leq n_{i,j}$ and $d > 0$ **do**
5: $\dot{P}^m_{j,k} \leftarrow P[l]$	5: $\dot{Q}^m_{i,j} \leftarrow Q[l]$
6: $l \leftarrow l + 1$	6: $l \leftarrow l + 1$
7: **end while**	7: **end while**
8: **return** $\dot{P}^m_{j,k}$	8: **return** $\dot{Q}^m_{i,j}$
9: **end procedure**	9: **end procedure**

If $\mathbb{G}$ denotes the set of lengths of the basic paths, i.e. $\mathbb{G} = \{\theta^1_{EC}, \theta^2_{EC}, ...\}$, then the worst-case end-to-end age latency $\alpha(EC)$, can be computed as:

$$\alpha(EC) = \max_{u \in \mathbb{G}} \left\{ \theta^u_{EC} + \dot{Q}^{u+1}_{\eta-1,\eta} - \dot{Q}^u_{\eta-1,\eta} \right\} \tag{4.38}$$

With regard to the other semantic, the worst-case scenario occurs when the sensor data of interest arrives right after the first reading point of the EC as shown by the upwards red arrow in Figure 4.21. Therefore, the worst-case end-to-end reaction latency, $\delta(EC)$, is given by:

$$\delta(EC) = \max_{u \in \mathbb{G}} \left\{ \theta^u_{EC} + \dot{Q}^{u+1}_{\eta-1,\eta} - \dot{Q}^u_{\eta-1,\eta} \right\} + T_\eta = \alpha(EC) + T_\eta, \tag{4.39}$$

4.6 System analysis

In this section we carry out an entire analysis of the system. We derived a response time analysis and an end-to-end worst-case latency analysis describing the advantages and disadvantages of the communication patterns analyzed before. In this context, we characterized the end-to-end latency analysis for the effect chains depicted in Figure 4.22, those chains are taken from the Formal Methods for Timing Verification (FMTV) challenge [85][12]. Specifically, the task set is produced from a real engine application and generated in the AMALTHEA model.

4.6.1 System and model constraints

The hardware described in the given AMALTHEA model consists of 4 cores, running at 300 MHz, 4 core-local RAMs, and one global DRAM. Non-local RAMs and the GRAM are accessible via a cross-bar interconnection network. We assume that all labels can be

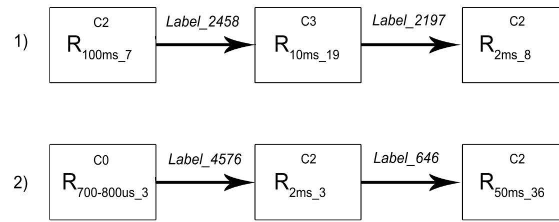


Figure 4.22: Proposed *ECs*.

accessed with a single memory read, neglecting the fact that there are labels which are larger than the bus width (i.e., occupy 64 or 128 bits against a 32-bit bus), hence more consecutive memory accesses may be required for a label transfer. Tasks are distributed among the four cores with different preemption schemes and types of activations. Notice that all cooperative tasks run on the same core. See Figure 4.23. As it can be easily seen

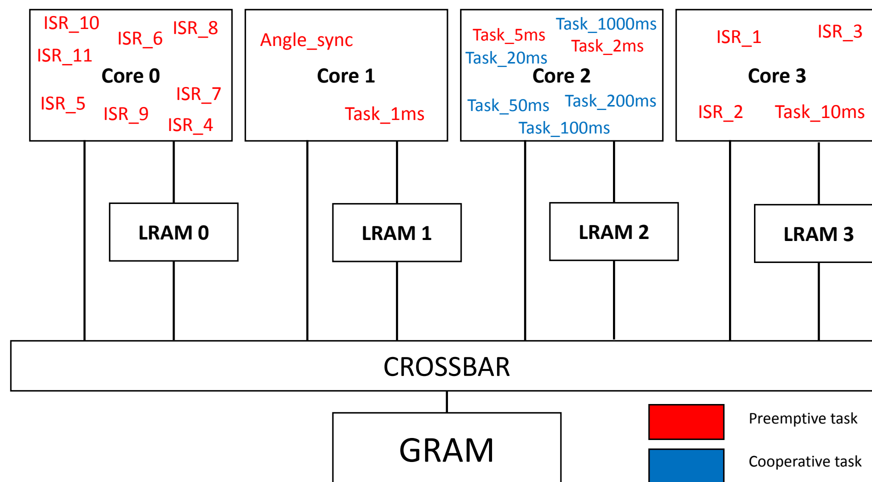


Figure 4.23: Hardware model with task distribution.

in the figure, the effect chain represents a producer consumer relationship between runnables mapped onto different tasks, which in turn, are allocated to different cores.

In order to get a clear vision of the model and to find out possible ways to optimize the end-to-end characterization, we performed a preliminary analysis of the memory accesses realized by all runnables in the given AMALTHEA use-case. We categorized the data items (labels) in three sets:

1. *PRIVATE* labels, which are exclusively accessed by one runnable;

2. *SHARED* labels, which are accessed by multiple runnables (e.g., in a producer-consumer fashion);

3. *UNUSED* labels, which we ignore.

Table 4.2 shows the number of labels in the proposed model, and their total memory occupation in KBytes, while Figure 4.24 shows how many (PRIVATE and SHARED) labels are accessed by (runnables assigned to) each core, and their size in bytes (right).

	#	Size (KB)
PRIVATE	8293	22.1
SHARED	1690	9.50
UNUSED	17	-

Table 4.2: System memory usage

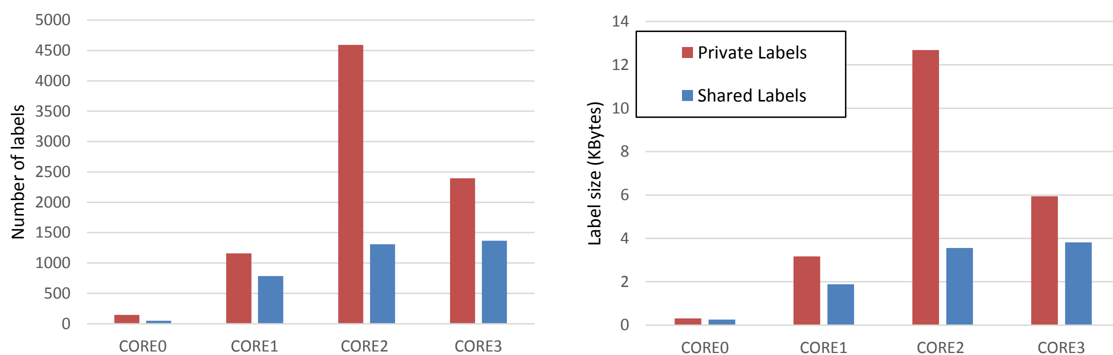


Figure 4.24: Distribution of labels on runnables/cores.

Then, we performed an experiment to measure the response time of the tasks given in the AMALTHEA model. We derived it for the Explicit, Implicit and LET communication patterns. In this context, we have to clarify that LET is a concept introduced to model the logical execution of an application, disregarding the physical execution. However, from an analytical perspective, it is necessary to guarantee that the physical execution respect the logical execution. Since the implementations proposed for the Implicit and LET communication only differ in when the inputs and outputs are produced, i.e., only affects the end-to-end latency, the response times for both communication patterns can be considered the same, because the worst case scenario for a LET task is when a given task read and publish the data in the same instance, that is the same behavior as the Implicit communication pattern. The respective results are shown in Figure 4.25. A first consideration that can be drawn from the previous analysis is that unfortunately the benefits of the communication mechanisms proposed can barely be observed because the tasks given in the model are not memory-intensive. In the future we consider the analysis of different task sets in order to characterize the positive effect of the aforementioned

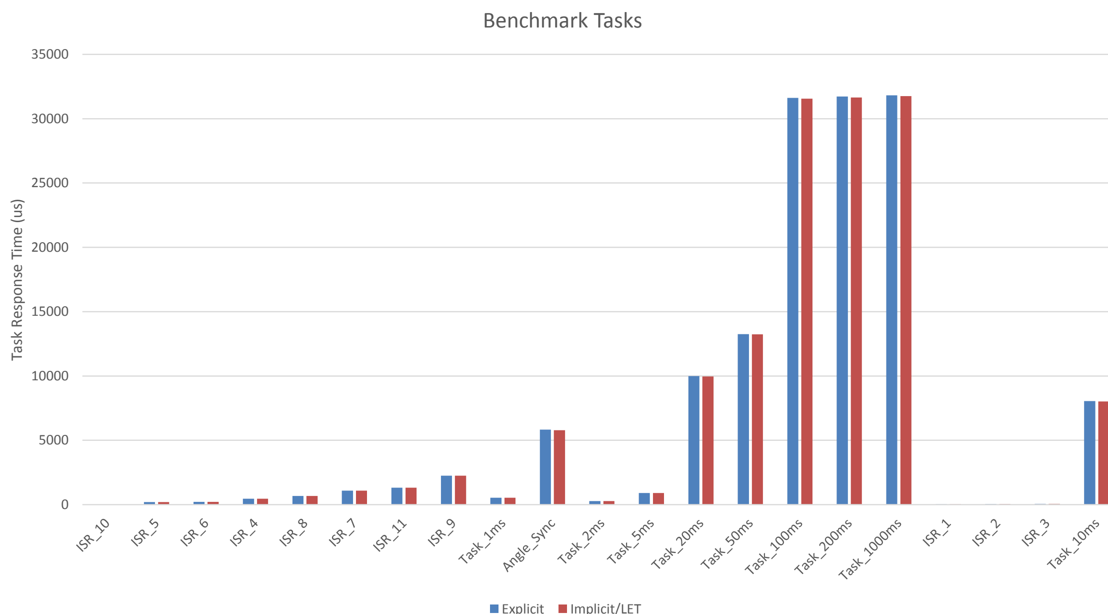


Figure 4.25: Task response times considering Explicit and Implicit/LET communication patterns.

mechanisms. On the other hand, we can observe that there is enough space to store all labels in any of the memories of the system, either in LRAMs (size 128 KB, according to the specifications) or GRAM (256 KB). Hence, it is possible to devise label-mapping strategies for minimizing a given constraint, for instance, design a methodology for shared label mapping which "optimizes" end-to-end latencies.

4.6.2 End-To-End latency analysis

The first effect chain under analysis ($EC1$ in Figure 4.22) is composed of three runnables mapped on to three ($\eta = 3$) different tasks τ_1, τ_2 and τ_3 with the following harmonic periods: 100ms, 10ms, and 2ms respectively. The second effect chain ($EC2$ in Figure 4.22) is also composed of three runnables mapped on to three tasks, however, while the last two tasks have periods of 2ms and 50ms respectively, the first task is sporadic with an interarrival time between 700μs and 800μs.

In the following we characterize the end-to-end latency of the first effect chain for the three communication patterns discussed in this chapter. Results of the characterization of both ECs are summarized in Table 4.3a. Even though the characterization of $EC2$ is similar to that of the other EC, it is worth mentioning that for the Explicit and Implicit patterns, the worst-case scenario occurs when the sporadic task releases jobs that are spaced 800μs apart since that enlarges the results obtained through (4.29), (4.30), (4.35)

and (4.36). For the LET pattern, however, 799μs lengthens the effect chain the most as it can be corroborated by means of (4.38). Since no sensor information is given, we assume that an EC starts at the release time of the task that initiates the chain. Therefore, in order to compute the worst-case end-to-end age latency for the Explicit and Implicit communication pattern, we append the best-case start time of the runnable that initiates the effect chain EC, s_1^I, and the best-case start time of its copy-out runnable, s_1^{last}, to (4.29) and (4.35) respectively.

Explicit Communication

From (4.5), (4.10), (4.11), and (4.29), $\alpha(EC) = s_1^I + \phi_1^{R_{100ms_7}} + \phi_2^{R_{10ms_19}} = 70{,}333\mu s + \phi_1^{R_{100ms_7}} + \phi_2^{R_{10ms_19}}$. From (4.26) $\phi_1^{R_{100ms_7}} = T_1 - s_1^{R_{100ms_7}} + R_1^{R_{100ms_7}} = 100000\mu s - 70{,}333\mu s + 13294{,}876\mu s = 113225\mu s$, and $\phi_2^{R_{10ms_19}} = T_2 - s_2^{R_{10ms_19}} + R_2^{R_{10ms_19}} = 10000\mu s - 196{,}366\mu s + 619{,}43\mu s = 10423\mu s$. Thus, $\alpha(EC) = 70{,}333\mu s + 113225\mu s + 10423\mu s =$ **123,718ms**. Similarly, from (4.30), we obtain $\delta(EC) = \phi_1^{R_{100ms_7}} + \phi_2^{R_{10ms_19}} + \phi_3^{R_{2ms_8}} = 123648\mu s + \phi_3^{R_{2ms_8}}$. From (4.26), $\phi_3^{R_{2ms_8}} = T_3 - s_3^{R_{2ms_8}} + R_3^{R_{2ms_8}} = 2000\mu s - 36{,}053\mu s + 99\mu s = 2062\mu s$. Then, $\delta(EC) =$ **125,710ms**

Implicit Communication

From (4.35) we know $\alpha(EC) = s_1^{last} + \phi_1^{last} + \Delta_2 + \phi_2^{last} + \Delta_3$ and from (4.5), (4.10), (4.11), and (4.32), $s_1^{last} = 2191{,}530\mu s$, $\phi_1^{last} = T_1 - s_1^{last} + R_1 = 100000\mu s - 2191{,}530\mu s + 31556{,}579\mu s = 129365{,}049\mu s$, $\Delta_2 = R_2 - R_2^0 = 8019{,}393\mu s - 73{,}523\mu s = 7945{,}87\mu s$, $\phi_2^{last} = T_2 - s_2^{last} + R_2 = 10000\mu s - 2812{,}369\mu s + 8019{,}393\mu s = 15207{,}024\mu s$, and $\Delta_3 = R_3 - R_3^0 = 279{,}596\mu s - 0{,}3\mu s = 279{,}296\mu s$. Then, $\alpha(EC) = 2191{,}530\mu s + 129365{,}049\mu s + 7945{,}87\mu s + 15207{,}024\mu s + 279{,}296\mu s =$ **154,988ms**. In a similar way, from (4.5), (4.10), (4.11), and (4.31), $\phi_1^0 = T_1 + R_1^0 = 100000\mu s + 13043{,}313\mu s = 113043{,}313\mu s$, $\phi_2^0 = T_2 + R_2^0 = 10000\mu s + 73{,}523\mu s = 10073{,}523\mu s$, $\phi_3^0 = T_3 + R_3^0 = 2000\mu s + 0{,}3\mu s = 2000{,}3\mu s$ and $\Delta_1 = R_1 - R_1^0 = 31556{,}579\mu s - 13043{,}313\mu s = 18513{,}266\mu s$. Finally $\delta(EC) = \phi_1^0 + \Delta_1 + \phi_2^0 + \Delta_2 + \phi_3^0 + \Delta_3 = 113043{,}313\mu s + 18513{,}266\mu s + 10073{,}523\mu s + 7945{,}87\mu s + 2000{,}3\mu s + 279{,}296\mu s =$ **151,855ms**

LET Communication

Due to the lack of sensor information, we assume $P_{0,1}^1 = 0ms$ and $P_{1,2}^1 = 100ms$. Since the three tasks composing the EC have harmonic periods, then there is only one basic path. By using *find-publishing-point* and *find-reading-point* in conjunction with (4.37) we get $\theta_{EC}^1 = Q_{2,3}^1 - P_{0,1}^1 = 110ms - 0ms = 110ms$. Moreover, from (4.38) we have $\alpha(EC) = \theta_{EC}^1 + \dot{Q}_{2,3}^2 - \dot{Q}_{2,3}^1 = \theta_{EC}^1 + Q_{2,3}^{21} - Q_{2,3}^{11} = 110ms + 210ms - 110ms =$ **210ms** and

from (4.39) $\rho(EC) = \alpha(EC) + T_3 = 210ms + 2ms = \mathbf{212ms}$.

Semantics	Explicit	Implicit	LET
Age	123,718ms	154,988ms	210ms
Reaction	125,710ms	151,855ms	212ms

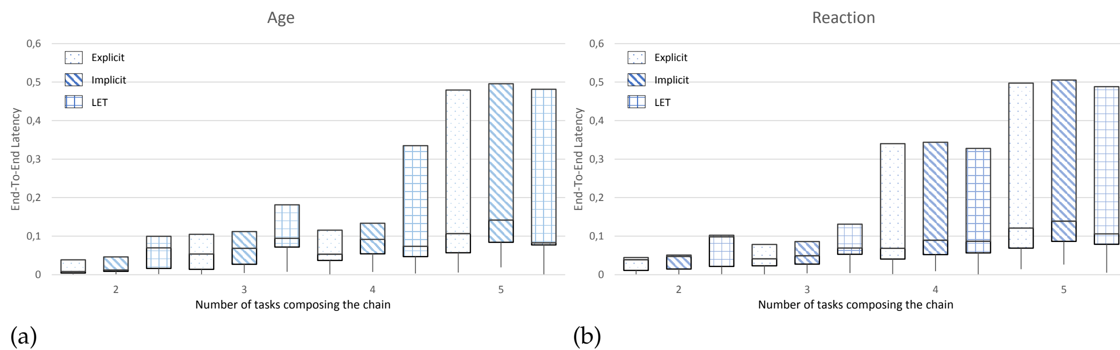
(a)

Semantics	Explicit	Implicit	LET
Age	2,844ms	6,54ms	53,597ms
Reaction	64,894ms	66,33ms	103,597ms

(b)

Table 4.3: End-to-End latency characterization of $EC1$ (top) and $EC2$ (bottom).

Conclusions of the End-To-End analysis



(a) (b)

Figure 4.26: Fig. 13. Normalized End-To-End latencies: Age latency (a) Reaction latency (b).

In order to get a better idea of the behavior of our three communication patterns, we need a more representative sample of effect chains, so with this end in view and due to the fact that very little has been published concerning effect chain benchmarks, we built our own effect chains based on the AMALTHEA model under discussion. As previously mentioned, an effect chain is a producer/consumer relationship between runnables, therefore, given a runnable that produces an output, we explore all the possible runnables, mapped on to a different task, that consume this output. By iterating over the whole model, our new effect chain set consists of over 1000 ECs composed of up to 5 tasks. Figure 4.26a & 4.26b depict two box plots of the normalized age and reaction end-to-end latency of our effect chain set respectively. In order to better appreciate the results obtained through this analysis, the data between the third quartile

and the maximum is omitted. In spite of the pessimism over the upper bounds of the end-to-end latency calculated for the Explicit and Implicit communication patterns and the lack of representative samples of large *ECs* (composed of more than three tasks), we can conclude from the plot that the end-to-end latency of the LET communication pattern tends to be larger than their counterparts. On the other hand, not only does the LET communication guarantee a deterministic inter-task communication but also data consistency. Yet this pattern introduces more copies, in the case of *NHSC*, and therefore more overhead with regard to the other two types of task-communication.

One of the advantages of the Implicit Communication is the fact that it might reduce the response time of a task since the penalty for accessing shared labels mapped onto the GRAM and/or non-local RAMs is paid by the copy-in and copy-out runnables only, a similar effect is also shown by [8]. With this sort of communication, runnables working on copies of shared labels only access their local RAM. The main advantage of this type of communication is data consistency but its downside is the extra footprint introduced by the copies and the longer end-to-end latency characterization in comparison with its Explicit counterpart. See Table 4.3a. The additional memory footprint introduced by the use of the communication patterns aforementioned, is depicted in Figure 4.27. From the results, we can consider the Explicit model as "baseline" since the implementation proposed does not introduce an additional overhead in terms of memory. However in the case of Implicit and LET, the memory footprint doubles. Specifically, LET consumes a little more memory footprint since there is a non-harmonic communication (*NHSC*) between two tasks (τ_{20ms} and τ_{50ms}), in that case, the task involved in the non harmonic communication, triple the copies performed (for further information see Section 4.4.2). 4.27 Basically, the Implicit implementation presented in this dissertation, which is derived from AUTOSAR, does not make use of locks; therefore, if data consistency between two (or more) variables is needed, e.g. between two labels representing the coordinates of an object, the operations on these labels performed by the copy-in and copy-out runnables should be surrounded by locks, when and where necessary, or should work on extra copies. It is also worth mentioning that in the automotive domain race conditions arise when two tasks, with different priorities or running in parallel to one another, modify the same label(s). This so-called multiple-writer scenario is discouraged since data consistency cannot be guaranteed through copies. In such a case Implicit communication should not be used. The use of Explicit communication with locking mechanism is one way to cope with this issue. Needless to say, the Explicit communication has the shortest end-to-end latency characterization, does not introduce extra memory footprint and relies on atomic operations, provided by the hardware, or other mechanisms, such as locks, semaphores, etc. in order to guarantee data consistency. On the other hand, neither the Explicit nor the Implicit communication take the concept

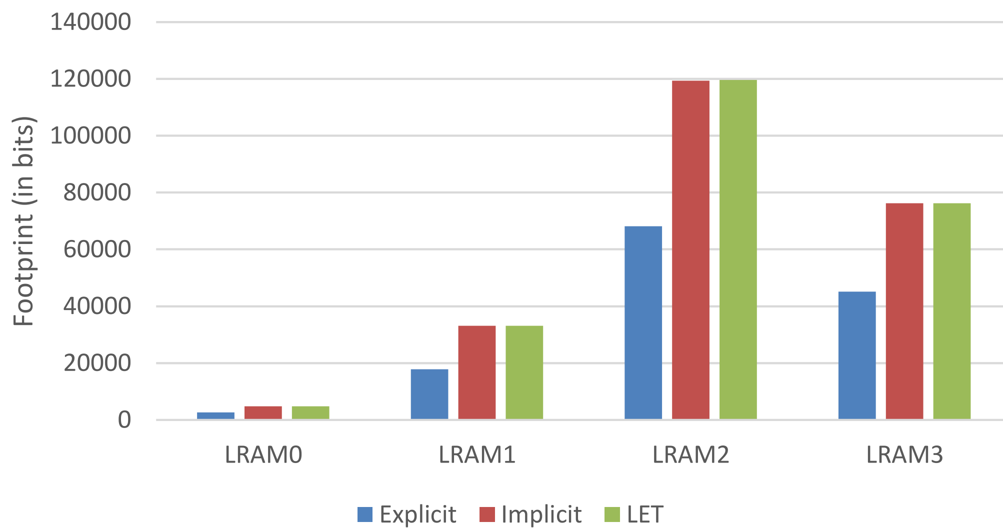


Figure 4.27: Memory footprint considering the communication patterns.

of determinism into account. This concept is very important for control algorithms as shown by [86] and [87]. Not only does the LET communication guarantee a deterministic inter-task communication but also data consistency. Indeed, since all the shared labels read by a given task are updated before and made available at the activation time of the task, there is no need to use locks in order to cope with data inconsistency, which makes the implementation lock-free. Nevertheless this pattern introduces more copies, in the case of *NHSC*, and therefore more overhead and lengthens the end-to-end latency of a given *EC*, as shown in Table 4.3a, with regard to the other two types of task-communication.

4.7 Summary

This chapter presented a study motivated by the industrial need to characterize the end-to-end latencies of effect chains of automotive real-time tasks communicating through shared variables in a multi-core system. As we discussed determinism in automotive is very important, especially for control applications. Different communication patterns adopted to ensure a consistent task communication were analyzed from a memory and timing perspective, characterizing the overhead introduced. A formal implementation has been proposed for two of them, namely Implicit communication and LET, analyzing the impact introduced in terms of memory footprint and communication delay. Moreover, an analytical characterization has been presented to compute valid upper bounds of end-to-end propagation delays of age and reaction latencies for a selected effect-chain taking into account the three communication patterns mentioned

in this work. The pros and cons of each of them were also discussed in order to assess which type(s) might suit best an application. The results have then been applied to an automotive industrial use case composed of multiple real-time tasks partitioned on a four-core setting.

While in Chaper 3 we covered issues like time predictability or isolation at virtualization level, in the future we intend to explore the concept of Logical Execution Time on top of virtualization platforms.

A Java implementation is available for the algorithms described in this chapter, available in[5].

[5]https://github.com/nachoSO/ChallengeWaters

5 Code generation support for automotive and general purpose platforms

The analysis derived in the last chapter characterizes the end-to-end latency of automotive task chains. Since it is no possible to test the real behavior of the application (because the representation used to perform the analysis is an abstraction of the real application), we developed a code-generator tool that automatically generates ready-to-use synthetic code that correctly mimics it.

As we described in Section 2.3, model-driven software development is a well-known paradigm in the automotive industry, where modularity and isolation between components are key to build safe, secure and certifiable systems. In this sense, the introduction of the AUTOSAR standard greatly improved the software development cycle, and the subsequent safety qualification process. Unfortunately, IPR limitations at industrial level often prevent academic researchers to apply their findings to real application settings. In this chapter, we introduce the HiPeRT Generator Tool that helps researchers creating synthetic yet realistic test cases, using a variety of techniques based on the model-driven development approach. The result is an open-source framework, that generates ready to use ANSI C code from different high-level modeling languages that are represented with a Directed acyclic graph (DAG). These modeling abstractions are: *RT-DOT* that is the real-time representation of *DOT (graph description language)* and the previously presented *AMALTHEA* framework.

This chapter is organized as follows. The next section describes the motivation behind this work and reviews the related work. Then, the code generator is presented in Section 5.3. Section 5.3.3 presents the AMALTHEA model interpretation and the code mapping process, showing a validation benchmark in Section 5.4, before a concluding discussion.

5.1 Motivation

Combining predictability and performance (i.e., worst-case vs. average-case performance) in systems with more than two/four cores is one of the most challenging tasks that automotive software engineers are facing today. The architectural complexity

of tightly-coupled computing engines makes timing and schedulability analysis significantly more complex, often leading to an over-estimation of the worst-case timing behavior. Multiple research projects aim at investigating the predictability vs. efficiency trade-off of multi-/many-core embedded platforms, proposing original solutions to the real-time scheduling problem [88], [89]. However, it is difficult to validate the effectiveness of these research findings to real industrial settings. The main reason is that even companies that are interested in cooperating with academia cannot share much information on their software architectures, ecosystems, neither the source code of applications, due to IPR restrictions.

This issue can be tackled leveraging well-known software design paradigms already adopted in industry, such as Model-Driven Development (MDD) (see Section 2.3) This common practice supports the design of complex software architectures and the Verification and Validation (V&V) process of real-time, safety-critical systems. Using a MDD approach, it is possible to abstract and represent high-level software components without the need to share "IP-critical" source code.

Currently, researchers from academia struggle to find a way for effectively characterizing the behavior real application code when running on multi- and many-cores platforms. To do so, they typically run benchmarks that stress and analyze the distinct components of the computing platforms, such as memory[1], disk[2] or CPU[3], or they use benchmarking suites to effectively validate their novel methodologies and techniques on platforms that are as similar as possible to the real ones. Significant examples from the real-time community are the Malardalen benchmark [90] and TACLeBench [91]. These benchmarks provide a collection of open-source programs to effectively validate tools and methodologies, but, unfortunately, they cannot capture the exact dynamics of real industrial applications.

We developed the *HiPeRT Generator Tool* – HGT[4], an open source tool to generate synthetic task code that is representative of the timing and precedence relations of concurrent industrial applications. Thanks to its scalable and easy-to-extend structure, HGT allows mimicking the timing behavior of parallel real-time applications represented as (i) Directed Acyclic Graphs (DAG) model that we called RT-DOT representation, or (ii) using the AMALTHEA model. HGT receives as input a set of task dependencies, timing and memory constraints of the modeled application. These constraints are parsed into an internal model (RT-DAG) and then transformed into ANSI C code. The generated code is then executed on top of a runtime that currently has PThreads as a parallel execution engine. The HGT runtime (HGR) is optimized for behavioral code emulation,

[1] http://www.bitmover.com/lmbench/
[2] https://www.coker.com.au/bonnie++/
[3] https://www.spec.org/cpu2006/
[4] Source code is released under GPL at: https://github.com/HiPeRT/HGT

including the replication of meaningful memory access patterns of the modeled tasks. This allows a tighter and more representative emulation of the timing behavior and contention sources of the considered application than existing approaches.

5.2 Related Work

There are several frameworks and tools offering the possibility to generate code from a metamodel instance. A widely used commercial tool is Simulink, which allows the generation of ANSI C and C++ code for real-time and non real-time applications from Matlab and Simulink diagrams. Another commercial tool is E4Coder [92]. E4Coder provides a set of tools used to simulate control algorithms and to generate code for embedded micro-controllers. The code generator tool translates ScicosLab and XCos diagrams into C language. In [93], the authors present edROOM, a graphical environment to edit ROOM models that automatically generates real-time C++ code. edROOM uses the model to describe the structure, communication topology and behavior of the system, automatically generating the application code. In [94], a MDD framework is proposed based on Java and XSLT called JComposer, for the automatic generation of real-time C-code for safety-critical embedded systems. JComposer runs on top of Linux-RTAI. In [95], a code generation framework is proposed to generate code for different target platforms modeled using AADL. In [96], another framework is presented for generating real-time code based on ADA. In this work, the functional behavior is characterized using UML2 adding the real time constraints using MARTE.

The main differences between these tools and the work presented in this dissertation are: 1) HGT generates code written in C (and soon OpenMP) that can seamlessly run on single-, multi- or many-core embedded platforms; 2) HGT hardware independent, i.e., we generate code for Intel and ARM architectures, replicating the timing behavior and precedence constraints in a very precisely way; 3) HGT is an open source tool, designed to be scalable in a very simple way, in contrast with the rest of works, that are closed or commercial. We are developing support for OpenMP and CUDA as potential code generation backend to target a wide set of commercial platforms.

5.3 Hipert Generator Tool - HGT

In this section, we describe the design of the HiPeRT Generator Tool (HGT). Figure 5.1 describes the overall system architecture and the three main components.

HGT is structured in three different layers.

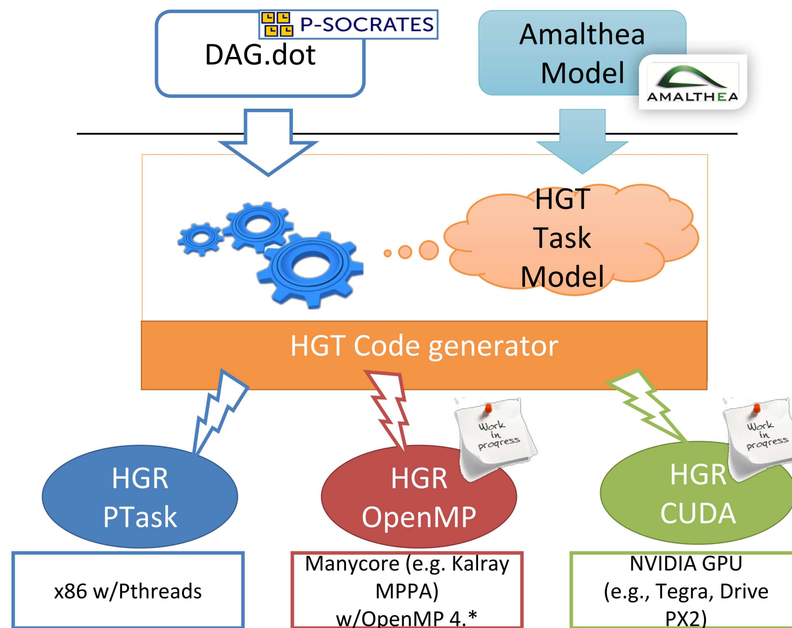


Figure 5.1: HGT framework organization.

- The *front-end* parses input files containing the task semantics to be translated into code. These files can be of different formats.
- The *core* layer translates the model semantic into the HGT task model, which is described in Section 5.3.2.
- The *back-end* generates code that mimics the behavior of the model on one or multiple platforms. It currently runs on top of PTask/Posix threads library[97], that provides RT tasking service.

5.3.1 Front-end

The goal of the front-end layer is to read the task/system representation that expresses the task constraints, e.g, period, deadline, worst-case execution time, etc. Intuitively, the more behavioral information is provided, the more accurate is the HGT system representation.

As shown in Figure 5.1, the front-end layer supports the AMALTHEA model, as well as an enhanced DOT representation[5] that allows expressing parallel task structures in the form of Directed Acyclic Graphs (DAG). DOT is a graph description language used for graph representation. A DOT file is basically composed of the definition of nodes and edges, where a node is the basic element of a graph and an edge represents

[5] https://en.wikipedia.org/wiki/DOT_(graph_description_language)

the precedence relationship between nodes. Inspired by the work done in [89], the DOT format has been specialized to include additional features that are typical of real-time systems, such as task periods and deadlines. We called such a specialized format RT-DOT, and it also inspired the HGT internal system representation, that is discussed in the next section. It is inspired by the P-SOCRATES FP7 project [89]. This HGT internal representation of DAGs, directly come from this. The RT-DOT file can be either written by-hand, or generated by a compiler, as it was done using the Mercurium source-to-source compiler[6]. For the sake of simplicity, we will not discuss in detail the RT-DOT semantics here, and we will focus only on how we generate code from the AMALTHEA model, because it inherently covers the complexity of the RT-DOT task model with small differences. Figure 5.2 summarizes the difference between the supported AMALTHEA and RT-DOT task model.

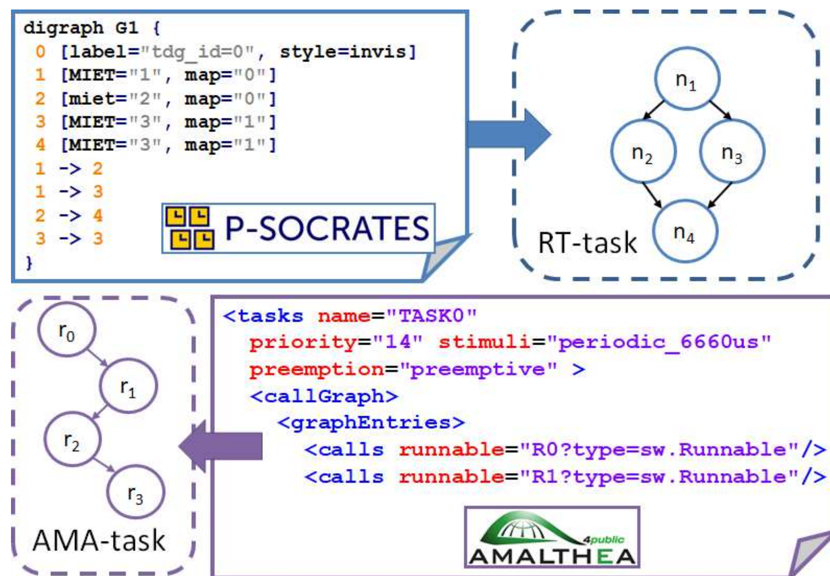


Figure 5.2: Example of RT-DOT and AMALTHEA files and models.

Accordingly with the task model defined in 4.3 and the AMALTHEA model, an AMALTHEA task τ is described using a Directed Acyclic Graph (DAG) $G(T) = (V; E)$, where V correspond to serial sequences of nodes and E the edges that denotes the precedence relationship between nodes. Each task τ_i is characterized by a period T_i, a priority P_i and a scheduling policy specified by PT_i. The scheduling policy may be either preemptive or cooperative, as Each node $\tau_{i,j}$ represent a different runnable, where j stands for the j-th task of the i-th task, with an associated worst-case computation time

[6]https://pm.bsc.es/mcxx

$C_{i,j}$ and a worst-case memory access size $M_{i,j}$. At runtime, a task τ_i releases a sequence of runnables. Each task needs to finish the execution before the deadline.

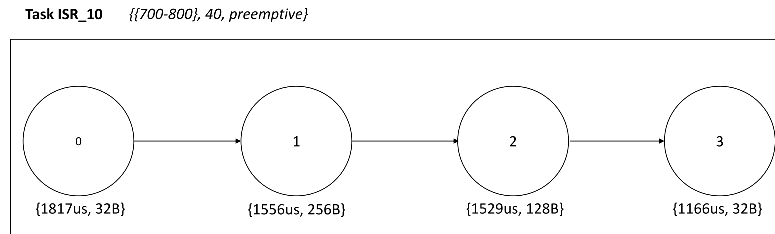


Figure 5.3: AMALTHEA task with 4 runnables.

The memory access models supported by the AMALTHEA model and implemented into our code generator can be summarized as follows:

- *Sparse*. The "traditional" execution model, where computation and memory accesses are interleaved.
- *Predictable execution model (PREM)* [8]. Under this model, the execution is decoupled into three different phases: a read phase, during which the task pre-fetches the required data into local memory; an execution phase, where the task performs pure computation workload, with local memory access; and a write phase, when the task publishes the labels it modified. Indeed, it is very similar to the Implicit task model defined previously. This technique allows improving both average and worst-case execution time, ensuring a more predictable behavior in a shared memory multi-core system, avoiding cache misses and memory interference [98]. Because it employs ordered prefetch and post-store phases from the global to the per-core local memory, that ultimately removed concurrent memory accesses due to uncontrolled cache misses. This model is very similar to the model proposed in [7] in the avionic domain.

5.3.2 Core

The HGT core layer processes the parsed DAG representation, through the AMALTHEA or the RT-DOT model, and does the following:

1. it determines a task grammar, where it precisely defines the task constraints;
2. it maps the input constraints into the defined previously task grammar; and
3. it defines the rules adopted to translate the model constraints into code.

An overview of the core layer process is shown in the Figure 5.4.

The first step consisted in creating a metamodel compliant with the constraints specified in the input file. In this case, we will consider the metamodel defined in AMALTHEA as a metalanguage to generate code (See Section 2). Several metamodeling frameworks exist to support this activity, like Ecore or MOF [99]. Eclipse Modeling Framework (EMF) [36] provides a modeling environment and runtime support for the model language manipulation. EMF provides a metamodel language called Ecore used for the model description. It also allows the specification of attributes contained in the language and their relation through class diagrams very similar to UML diagrams.

In a second step, when the metamodel is defined it is possible to create a model that complies with the metamodel created previously. In this case, the AMALTHEA model is generated based on an engine control application provide by Bosch. When the model is defined, it is translated onto the HGT Task Model, as shown in Table 5.1 and discussed in Section 5.3.3. The internal HGT model is based on XMI (XML Metadata Interchange), a subset of XML used for the model representation, enhanced with parameters to accurately capture the semantics of typical real-time tasks. The adoption of standard XML files as internal representation lets us completely decouple the frontend and backend representation, and makes the overall tool completely scalable. Yet, our custom XMI format is complete and scalable enough to capture all RT-related system properties such as task periods, deadlines, etc. The most important features of the system model are:

1. *Job execution model*. It represents the memory access model, either PREM or Sparse. If in Sparse mode, it is possible to define whether the memory accesses are sequential or random, as well as the access granularity, i.e., the minimum memory-computation unit defined in bytes. Note that the use of PREM implies that memory accesses are pre-fetched "in block" into the last level cache or in a local scratchpad memory [8]. In the AMALTHEA case, different memory access granularities may be specified following different semantics (see Section 5.3.3): at task boundaries (implicit model) or at task release times (Logical Execution Time, LET).

2. *Task scheduling policy*. Inspired by Posix Threads standard, it can be FIFO or Round Robin. In the future we plan to add support for scheduling policies used in the automotive industry such as the co-operative scheme.

With these rules and the DAG representation, task or nodes of the HGT model are translated, e.g., into a *PTask* [97], and edges are translated into *POSIX mutex* for the ANSI C code (see Table 5.1).

The front-end and core layers are implemented using the Epsilon framework [100]. Epsilon is a family of languages that provide an infrastructure for code generation,

model-to-model and model-to-text transformation, for the Eclipse platform. Specifically, the language used for translating the model into code is Epsilon Generation Language (EGL). EGL [101] is a template-based model-to-text language for generating code, documentation and other textual artifacts from models. It provides a template-based framework with so-called *translation rules* for the different attributes defined in the model. To port AMALTHEA and RT-DOT on the internal HGT representation, we

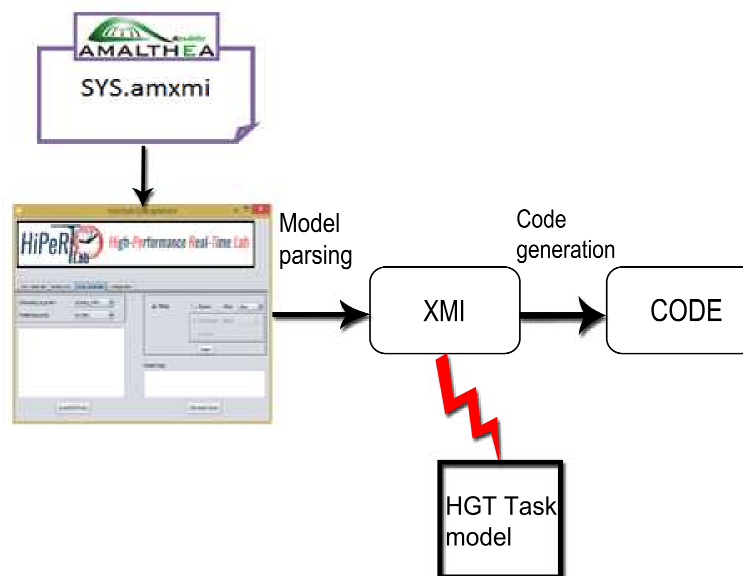


Figure 5.4: Example of code generation process from AMALTHEA model.

developed a lightweight parser for the input file, and the necessary code (written in Java) for HGT. This process is explained in the following subsection.

5.3.3 AMALTHEA mapping onto HGT

In this subsection, we provide a description of the AMALTHEA software constraints and how they are mapped into the HGT task model (RT-DAG) and, ultimately, into C code complying with these constraints. Similar considerations are valid for applications specified under the RT-DOT model.

As mentioned in Section 2.3.1, some elements of the metamodel are not mappable into code. For instance, OS-level scheduling artifacts such as limited preemption support [42] or SCHED_DEADLINE [39]) are implicitly assumed to be present in the backend emulation platform. Similarly, hardware properties' like the number of cores or the memory hierarchy cannot be "enforced" in a given machine, even though AMALTHEA model can express them (for instance, you cannot simulate an application running on 8 cores if there are only 2 available on a target machine). For this reason, and for the

sake of simplicity, in this dissertation we only cover the properties related to software components, such as, e.g., the task-runnable relation and the effect chain mapping.

In the AMALTHEA model, the main software component is a task. The metamodel defines a task as a sequential directed acyclic graph whose nodes are called runnables. A task can be seen as a "container" of runnables, represented with a call graph. Therefore, according to the AMALTHEA model, runnables within the same task are executed sequentially (See also Figure 5.2). There is a direct correspondence between AMALTHEA runnables and AUTOSAR runnables. I While AUTOSAR tasks may have functional dependencies between them (implemented, for instance, with mutexes, semaphores or locks), the AMALTHEA model does not capture this feature. This greatly simplifies the AMALTHEA-to-HGT mapping, avoiding the need of implementing additional synchronization constructs within the task code.

In a first step, we map those elements into code. Tasks and runnables could be seen as "classic" POSIX threads, which are also assigned a period, a deadline and a priority. We leverage the PTask library to implement this additional features [97]. Each AMALTHEA task is mapped onto a PTasks, while runnables are implemented as PThreads. AMALTHEA tasks have a period, a deadline and a priority. These features are not natively supported in any "standard" programming API, such as POSIX threads. Instead of re-writing them by-hand, we leverage the open-source PTask library to support them. In this library, every PTask is composed of multiple PThreads. As a consequence, we map AMALTHEA tasks into PTasks, and runnables as PThreads.

In AMALTHEA, tasks communicate through shared labels. A task can be either a sender or a receiver, namely a writer or a reader of a shared label. While there may be multiple receivers per label, each label has at most one writer. AMALTHEA defines the concept of effect chains to model asynchronous inter-/intra-task communication with a producer/consumer relationship between runnables.

As we shown in the last chapter, the end-to-end latency of effect chains is the main performance metric for engineers, and it is defined as the maximum propagation delay for an event affecting the first runnable of the chain (i.e., when an input label is modified) to the final one (when it writes its output label). It is worth noting that effect chains do not have a blocking semantic, i.e., runnables are always active and periodically activated, independently on other runnables and/or external events. In HGT, event chains can be translated merely as an asynchronous communication between threads, where one thread writes in the reader memory region. No additional mechanisms are needed to support task communication, other than read/write accesses to shared labels.

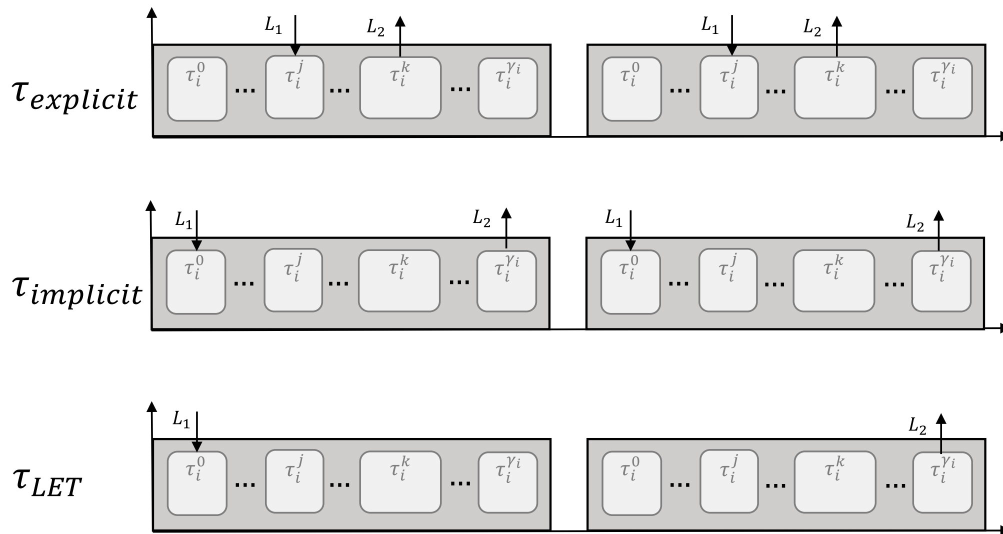


Figure 5.5: AMALTHEA communication models.

As we argued, different patterns are proposed to support a deterministic communication that maintains data consistency[7]. These patterns can be summarized as follows:

1. The Explicit model represents a "traditional" communication mode, where no pre-fetching is performed, but tasks freely access the shared labels in an unsynchronized way. When the same label may be accessed by different tasks, this model may lead to race conditions and data inconsistencies.

2. With the Implicit communication model, the communication is performed at task boundaries. In order to avoid data inconsistency, tasks accessing shared labels work on task-local copies created at the beginning of the job execution. After working on these local copies, the task writes back its local copies to the original labels, i.e., it publishes its results at the end of its execution.

3. The Logical Execution Time (LET) model LET requires that the input and output variables are updated at the beginning and at the end of the communication interval respectively. The communication interval is determined by the release times of the tasks involved in the communication.

From a design perspective, Figure 5.5 depicts these patterns. At implementation level, the Explicit communication pattern can be simply implemented with a direct communication through shared memory banks (e.g., mailboxes). The model ignores the potential

[7]Implicit and LET communication patterns are not available in the current AMALTHEA specifications, but they will be included in upcoming versions.

performance and data consistency issues that may arise following this approach. On the other hand, to implement the Implicit and the LET communication pattern, runnables create local working copies for the shared labels before the task starts in order to maintain the data consistency. Then, the difference between the Implicit and LET models is when the publishing time or write-back of the modified labels is defined. In this sense, replacing the shared labels with the modified local copies, happens at task completion in the implicit case, and at the end of the communication interval in the LET case. In LET, When tasks have harmonic periods, the communication interval is the hyperperiod of the communicating tasks. See 4.3 for more details about the conceptual implementation of the LET model. Table 5.1 depicts the equivalence between the RT-DOT and the AMALTHEA model.

RT-DOT		AMALTHEA		PTask	
Name	Properties	Name	Properties	Name	Properties
RT-Task	Period, deadline	Task	Period, deadline, concurrent, {map}	Real-time PTask	Period, deadline, concurrent
Job/Thread	Core map, concurrent	Runnable	**mapping{Task, no mapping}**	PThread	Concurrent, sched. policy
Dependency	Exec. dependency, block	*Dependency*	*Implicit in runnable*	PThread mutex	Blocking, preempt. point
Comm. channel	Thread and shared labels	Event chain	Runnable and shared labels	Shared vars	–

Table 5.1: AMALTHEA and RT-DOT model mapping.

5.3.4 Back-end

Finally, to allow HGT to be portable onto different platforms, we wanted to provide the necessary software abstraction to transparently support execution of the same code across different architectures. For this reason, our tool generate code that runs on top of a simple API layer, called *HG-Runtime* (HGR). The HGR API allows:

1. generating RT-tasks with their own period, deadline, priority, etc;
2. spawning concurrent RT-threads/jobs within a single task, and managing their dependencies;
3. simulating the execution of ’C’ clock cycles on a platform core, with or without performing memory accesses (simulating cache misses);
4. accessing ’M’ bytes in the main platform memory.

We implemented a first version of HGR based on the PTask library [97], a research API that enhances PThreads with real-time characteristics, like task periods, deadlines, priorities, and OS scheduling policy. Tasks in the HGT model are translated into *PTasks*, nodes into PThreads, and edges are implemented as *POSIX mutex* (see Table 5.1). This backend is used in Section 5.4 to validate our tool.

We are currently developing backends for (i) the Kalray MPPA many-core platform [102], by implementing HGT on top of the OpenMP runtime developed within the P-SOCRATES project[89]; (ii) automotive-grade heterogeneous SoCs based on tightly-coupled NVIDIA GPUs [103], [104], porting the HGR API on top of CUDA [105], and (iii) Infineon Aurix Tricore [106], a well-known platform that is already ASIL-D certified and shipped within real industrial systems.

5.4 Experimental evaluation

In this section, we show how we implemented the synthetic tasks generated by the HGT tool so that they replicate the timing behavior of the AMALTHEA tasks given as input. The main problem was to accurately model memory and execution phases running on top of a general purpose OS. We ran different experiments, measuring the error between the expected emulated time and the actual one (both M and C phases). Measures are taken on an i7-4770T CPU @ 2.50GHz, with 32GB of RAM, running on an standard Ubuntu 4.4.0-53 with real-time extensions.

5.4.1 Memory Phase

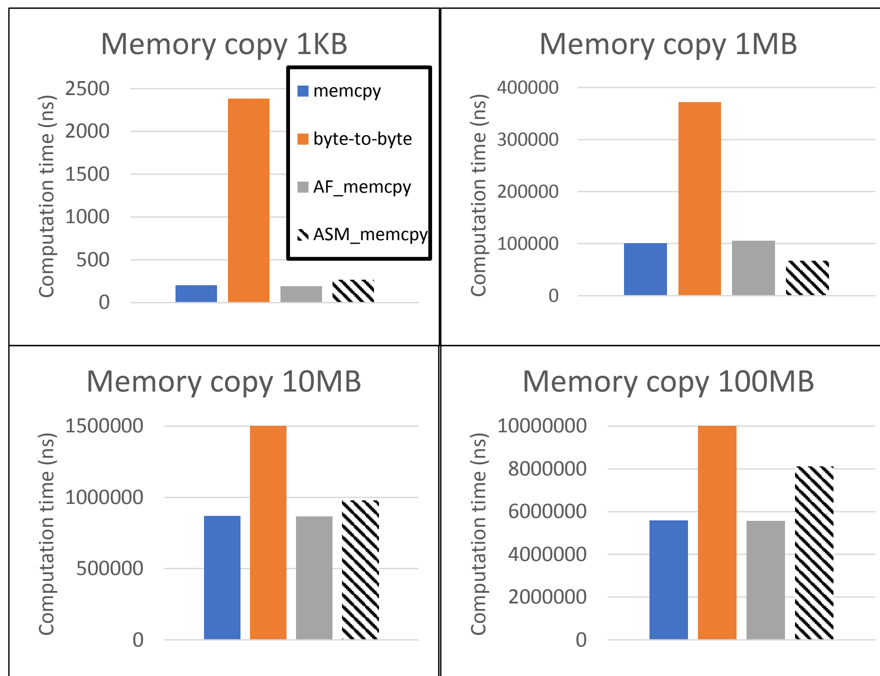


Figure 5.6: Memory performance test.

We first compared the memory transfer delay of the different *memcopy* variants we implemented in HGT, namely:

- the linux standard version of `memcpy`;
- a byte-to-byte copy;
- *AFmemcpy*[8], an optimization of the `memcpy` written in assembly; and
- ASM memcpy, our implementation of the `memcpy` written in x86 assembly.

As shown in Figure 5.6, the standard implementation of the AF memcpy performs slightly better than the "standard" `memcpy`. Still, the former implementation is preferable, because it relies on a standard `memcpy` that is available on all platforms that support a C development toolchain. For the very same reason, it might not be worth implementing *non-portable* ASM code for the M phase, even if in some cases it might be the best performing one.

5.4.2 Execution Phase

As explained in section 5.3, in its "generic" form, the front-end reads the AMALTHEA model. According to the model defined (see Section 4.3), each runnable (or node) is characterized by a worst-case execution time, specified in time-units, and a memory access size, specified in bytes. [9] Depending on the execution model adopted, PREM or Sparse, the HGT implementation of memory accesses at node level varies. Under the PREM model, memory phases are implemented using a single `memcpy` of corresponding size, followed by an execution phase lasting for the specified WCET. Under the Sparse model, we evenly divide the memory accesses into multiple sequential blocks, each accessing memory with a given *granularity* (specified in the front-end), and then performing a given amount of computation (see Figure 5.7). The number of blocks for a node $\tau_{i,j}$ can therefore be computed as

$$\phi_{i,j} = M_{i,j}/granularity. \tag{5.1}$$

Similarly, the worst-case execution time of the considered node is accordingly distributed among the blocks, so that each block has an execution time of $\beta_{i,j}$, where

$$\beta_{i,j} = C_{i,j}/\phi_{i,j}. \tag{5.2}$$

An example of both approaches is illustrated in Figure 5.7. We experimented two different ways to reproduce a given execution time: ASM and CLOCK. The ASM

[8] http://www.agner.org/optimize
[9] Both terms are calculated by a WCET analysis. This aspect is out of the scope of our work, at the moment.

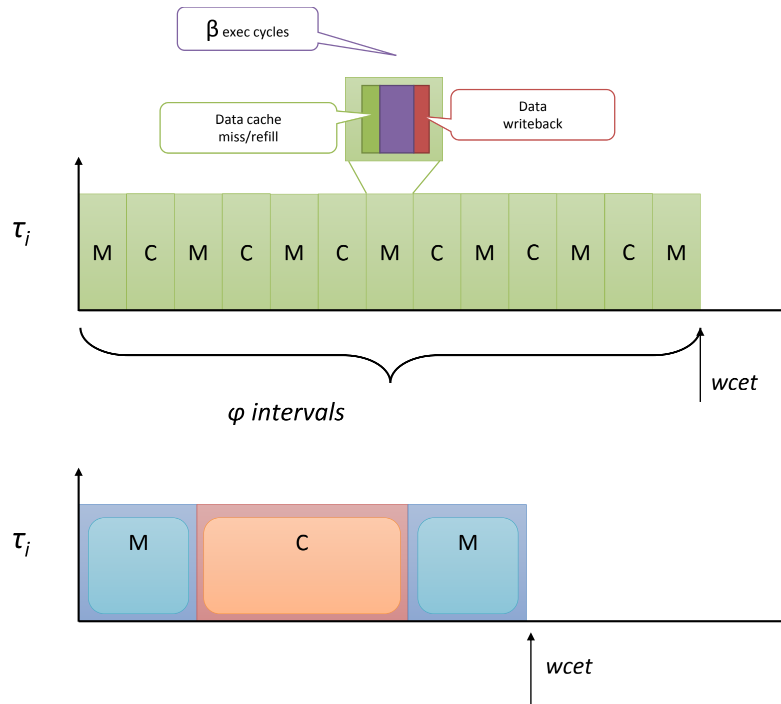


Figure 5.7: Task execution model.

implementation executes $\beta_{i,j} * F$ NOP ALU operations, coded in assembly, where F is the CPU frequency. The CLOCK implementation employs a spinning approach where the task continuously reads a timer to check when $\beta_{i,j}$ time units have elapsed. We are planning to improve this process in next versions. To compare the effectiveness of both approaches, we characterized the difference between the expected and the measured execution times under different configurations, i.e., for different $C_{i,j}$-$\phi_{i,j}$ combinations. Each configuration was ran 10K times, measuring the largest divergence w.r.t. the desired value. Results are summarized in Figure 5.8, showing the percentage of the accuracy error of our ASM implementation against the reference CLOCK. As expected, the most accurate approach is the ASM. Still, there are problems when reproducing tasks with a small execution time and a large data size (thus, a small C/ϕ ratio – columns in red, on the top-right of Figure 5.8). We are working on more enhanced ASM methods that may obtain a higher accuracy also for smaller block sizes. The drawback of ASM-based approaches is that they are not platform-independent, but they need to be re-written for every core Instruction Set Architecture that one wants to support as a backend. Still, these are very few lines of code, and they are wrapped within a HGR-specific subroutine.

With regard to the communication patterns we already give support to the code generation complying with the semantics defined previously, however, because we use

	PHI	512	1024	10240	102400	1024k	10240k	102400k
1 ms	CLOCK	6,10	10,27	89,37	889,25	9598,57	96614,45	966825,65
	ASM	2,70	3,97	11,39	32,60	78,19	1667,25	17589,36
10 ms	CLOCK	0,90	1,82	10,75	86,05	879,84	9572,47	96624,53
	ASM	3,71	2,15	2,47	9,47	13,73	76,47	1664,93
100 ms	CLOCK	0,11	0,23	1,22	8,61	83,31	866,89	9569,64
	ASM	0,62	0,48	0,79	1,07	8,19	11,60	73,29
500 ms	CLOCK	0,05	0,09	0,33	2,07	16,72	141,41	1833,61
	ASM	0,39	0,33	0,34	0,48	1,87	15,41	17,22
1 sec	CLOCK	0,03	0,05	0,19	1,06	8,41	82,54	866,99
	ASM	0,40	0,39	0,22	0,34	0,95	8,05	10,97
2 sec	CLOCK	0,00	0,01	0,07	0,58	4,31	46,79	383,48
	ASM	0,41	0,20	0,23	0,28	0,54	4,21	38,18

Figure 5.8: Computation performance test.

a general purpose operating system there are several limitations in the creation of a representative representation of the model. In order to fill this gap, we plan to give code generation support for the real-time operating system Erika using the Aurix TC275 microcontroller [10].

5.5 Summary

In this chapter, we presented an open-source tool for generating synthetic real-time tasks complying with different memory and execution models: parallel or sequential, DAG-based or AUTOSAR, PREM or sparse. We introduced a formal basis for the generation of synthetic AUTOSAR based tasks, complying with different memory and execution models. We proposed a software mapping from the RT-DOT and the AMALTHEA model, that extracts and translates into code the modeled software properties. The tool has been designed following a model-driven development approach, to allow for an easier extensibility and customization to different hardware and software architectures. The generated code can be adopted to test the effectiveness of scheduling algorithms, operating systems and runtimes under a variety of configurable workloads, allowing one to test the impact of different execution models over a considered architecture.

HGT is licensed under GPL, and a final version of the code is be released as open source. The source code and the tool may be downloaded from the website[11].

[10] https://www.infineon.com/cms/de/product/evaluation-boards/kit_aurix_tc275_ard_sb/
[11] https://github.com/HiPeRT/HGT

6 Conclusions

Finally, we briefly review the results achieved and draw a conclusion from this dissertation. We addressed several research problems related to the allocation, analysis and development of real-time applications in many different domains.

First, we summarized many of the the key concepts behind functional safety giving a higher level insight of how ISO-26262 works at software level and how this development is managed. Moreover, we reviewed the AUTOSAR standard. We described many of the mechanisms at operating system level and how those mechanisms are translated in the real-time scheduling theory such as, scheduling algorithms or communication patterns. Then, we explained the Model based development concept.

Chapter 3 presented a comprehensive and up-to-date survey of the literature on I/O management within virtualized environments. We exposed observations and considerations concerning predictability on shared hardware devices. Specifically, we considered I/O-disk scheduling as a particular case for resource sharing, we highlighted the main results concerned with improving the delays due to competing accesses to storage devices in virtualized environments. Furthermore, we demonstrated how I/O intensive tasks executing on top of non-critical virtual machines can easily cause more critical partitions to experience high blocking delays. In this context, we presented the concept of Freedom From Interference and the mechanisms proposed by ISO-26262 to guarantee a certain level of determinism and isolation. We believe that this contribution can help an interested reader in understanding the importance of Freedom From Interference and the metrics used to certificate those kinds of systems under automotive safety assessments for the execution of mixed-criticality applications.

In Chapter 4 we proposed and analyzed the implementation of different communication patterns that are used for shared-memory inter-task communication in the automotive domain. Specifically, we presented a tight schedulability analysis for the AUTOSAR task model in which cooperative and preemptive tasks are concurrently scheduled on the same partitioned platform. Moreover, we proposed a formal implementation for the Explicit, Implicit and LET communication patterns. In this context, we introduced a precise calculation of two propagation delay semantics: *Age* and *Reaction*. We presented an analysis of a real engine control application. As proved in the analysis, neither the Explicit nor the Implicit communication consider the concept of time deter-

minism that is very important for control applications in the automotive context. On the other hand, the LET communication implementation proposed, guarantees a deterministic behavior in the execution but also a deterministic inter-task communication while guaranteeing data consistency. Furthermore, we discussed the trade-off between the communication patterns in terms of memory footprint. This contribution can serve as a guidance for automotive engineers in the partitioning and implementation of the different communication models proposed. The metrics and trade-offs presented may be also used to consider how communication must be performed in function of the application requirements.

Chapter 5 presented the *Hipert Generator Tool (HGT)* a framework that generates code from different modeling languages following a Model-based approach. This tool was motivated by the lack of real-time frameworks for the test and execution of synthetic benchmarks. Moreover, in order to test the advantages and disadvantages of the communication mechanisms described in the previous chapter, we defined the constraints needed for code generation of the Explicit, Implicit and LET communication patterns. The tool is already compatible with AUTOSAR-specific mechanisms and synchronization constructs. Moreover, the tool was developed to support the DOT modeling language. Furthermore, we provided the performance model of our implementation in terms of execution and memory accuracy. This framework can help researchers in the creation of synthetic and realistic test cases.

6.1 Future Work

Many issues remain still open concerning the topics treated in this dissertation. In this sense, the work presented can be extended towards several directions for future research.

Future work, Virtualization. In the survey presented in Chapter 3 we were concerned with shared resource management within virtualization platforms at I/O level. Furthermore, in Chapter 4 we showed that LET provides mechanisms to improve the determinism while guaranteeing data consistency. In this sense, we intend to explore LET as a mechanism for provide determinism and data consistency within virtualized environments. To better understand this idea, consider two different virtual machines with different applications that communicates through the shared memory among each other. Both virtual machines read data from the shared memory at fixed temporal intervals, one of the virtual machines also writes into the shared memory region the information that is produced at the end of the execution. As can be easily observed, this can lead to data inconsistency issues. Data consistency among partitions can be achieved

using lock-based protocols, leading to high blocking delays on the reader partition. Many lock-based resource sharing protocols have been proposed in the literature, most of these protocols are complex in the analysis or difficult to implement, for example, *Multiprocessor Stack Resource Policy (MSRP)* or *Multiprocessor Priority Ceiling Protocol (MPCP)*. Thanks to its simplicity, we believe that the Logical Execution Time model and/or wait-free mechanisms are suitable for providing determinism, data consistency and consequently for managing the access to shared resources. We will explore this idea in depth in the near future using the Jailhouse hypervisor.

Future work, End-To-End. As a future work, we plan to extend the presented analysis to integrate different task models, like adaptive variable rate tasks (AVR) and aperiodic arrivals. Furthermore, we plan to improve the provided end-to-end bounds increasing the complexity of the analysis for ruling out pessimistic scenarios that may not appear for selected effect chains. We plan to propose task-to-core partitioning strategies to improve, i.e., reduce the latency metrics of selected effect chains. Moreover, since in this dissertation we dealt with the end-to-end latency of tasks that are allocated in the same ECU (intra-ECU communication). In the near future we will derive an analysis of the presented communication patterns (Explicit, Implicit and LET), when the communication is established between two or more different ECUs (Inter-ECU communication) that are potentially composed of multi-core processors. In this sense, the problem is more related to synchronization between buses. For instance, let us suppose that we have a periodic task that triggers a communication with a task allocated onto a different ECU. Local clocks placed on different ECUs are not synchronized. However, in [107] authors proposed a period optimization for distributed ECUs satisfying end-to-end latency constraints. We are interested in the behavior and determinism provided by the mentioned communication patterns in this communication setting. We also plan to explore static offset assignment with the LET model. In this sense, the real-time performance of non-harmonic tasks may improve, getting closer to the constant end-to-end latency experienced in the harmonic case. The introduction of offsets not only may reduce response times and end-to-end latencies, but it also allows decreasing the jitter of important control parameters.

Future work, HGT. Regarding the code generator, we plan to use the tool to generate synthetic benchmarks to test the effectiveness of PREM-based execution models with respect to standard approaches under different multi/many-core architectures and operating systems. The tool will be enhanced to include a configurable number of shared resources and critical sections for more realistic modeling of real industrial applications. Due to the impossibility to execute particular model mechanisms, like

scheduling algorithms or particular task activations, we excluded to test the effectiveness of the AMALTHEA execution model. In this context, the implementation for the communication patterns are ready to be ported to an AUTOSAR/OSEK OS. We plan to port the code generator to be executed on top of the Erika operating system, to prove the effectiveness of the execution model and mechanisms proposed. Moreover, the backend will be ported on a ECU based on a quad-core Aurix microcontroller [106], as well as on top of the OpenMP/CUDA runtimes supported by Kalray MPPA and NVIDIA GPUs.

Bibliography

[1] G. C. Buttazzo, *Hard Real-Time Computing Systems (The International Series in Engineering and Computer Science)*, 1st, ser. The International Series in Engineering and Computer Science. Springer, 1997, ISBN: 0792399943,9780792399940,9780585280059. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=8573C47BDBEE5CD46AFDD1D5B58C7BEB.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling", in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, San Jose, California, USA: ACM, 2011, pp. 365–376, ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108.

[3] S. K. Baruah, M. Bertogna, and G. C. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*, ser. Embedded Systems. Springer, 2015, ISBN: 978-3-319-08695-8. DOI: 10.1007/978-3-319-08696-5. [Online]. Available: https://doi.org/10.1007/978-3-319-08696-5.

[4] *ISO 26262-1:2011 - Road vehicles – Functional safety*, Standard, Geneva, CH, 2011.

[5] A. R. Inc., *Avionics application software standard interface Part 1 – required services*, Standard, 2005.

[6] E. W.-. RTCA SC-205, *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[7] 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. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01121700.

[8] 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 *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, 2011, pp. 269–279. DOI: 10.1109/RTAS.2011.33. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2011.33.

[9] C. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm", in *Advances in Real-Time Systems*, 2012, pp. 103–120. [Online]. Available: /pubpdf/ARTS-chapter.pdf.

[10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003, ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805825.

[11] T. Kloda, B. d'Ausbourg, and L. Santinelli, "EDF schedulability test for the E-TDL time-triggered framework", in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016, pp. 1–10. DOI: 10.1109/SIES.2016.7509414.

[12] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewycz, "2017 Formals Methods and Timing Verification (FMTV) challenge", 2017, pp. 1–1. [Online]. Available: https://waters2017.inria.fr/challenge/.

[13] N. Navet and F. Simonot-Lion, *Automotive Embedded Systems Handbook*, 1st. Boca Raton, FL, USA: CRC Press, Inc., 2008, ISBN: 084938026X, 9780849380266.

[14] "Functional safety of electrical/electronic/programmable electronic safety-related systems", International Organization for Standardization, Geneva, CH, Standard, 2000.

[15] N. H.T. S. Administration, *NHTSA Report on Toyota Unintended Acceleration Investigation*, 2011. [Online]. Available: https://one.nhtsa.gov/About-NHTSA/Press-Releases/ci.NHTSA%E2%80%93NASA-Study-of-Unintended-Acceleration-in-Toyota-Vehicles.print%7D%7D.

[16] M. I.S. R. Association and M. I.S.R. A. Staff, *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*, 2013. [Online]. Available: https://books.google.es/books?id=3yZKmwEACAAJ.

[17] The AUTOSAR consortium, *AUTOSAR: The Software Component Template*. [Online]. Available: http://www.autosar.org.

[18] N. Naumann, "Autosar runtime environment and virtual function bus", *Hasso-Plattner-Institut, Tech. Rep*, p. 38, 2009.

[19] E. GmbH, "RTA-RTE V6.2.0", Tech. Rep. [Online]. Available: https://www.etas.com/download-center-files/products_RTA_Software_Products/RTA-RTE_User_Guide.pdf.

[20] H. Zeng and M. D. Natale, "Efficient implementation of AUTOSAR components with minimal memory usage", in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 2012, pp. 130–137. DOI: 10.1109/SIES.2012.6356578.

[21] D. Wang, H. Li, Y. Yang, M. Zhao, and J. Wang, "Enforcing model consistency in the AUTOSAR modeling environment", in *2010 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 226–230. DOI: 10.1109/ICIME.2010.5477460.

[22] *AUTomotive Open System ARchitecture (AUTOSAR), Specification of Operating System*, version 5.0.0, 2011.

[23] K. Reif, *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics*, ser. Bosch Professional Automotive Information. Springer Fachmedien Wiesbaden, 2014, ISBN: 9783658039745. [Online]. Available: https://books.google.it/books?id=4ymWngEACAAJ.

[24] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems", in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, M. Bertogna, Ed., ser. Leibniz International Proceedings

in Informatics (LIPIcs), vol. 76, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:20, ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7162.

[25] A. Biondi, "Analysis and Design Optimization of Real-Time Engine Control Software", 2016.

[26] H. Zeng and M. D. Natale, "Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms", in *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, 2011, pp. 140–149. DOI: 10.1109/SIES.2011.5953656.

[27] R. Wyss, F. Boniol, C. Pagetti, and J. Forget, "End-to-end Latency Computation in a Multi-periodic Design", in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13, Coimbra, Portugal: ACM, 2013, pp. 1682–1687, ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480678. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480678.

[28] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree, "From control models to real-time code using Giotto", *IEEE Control Systems*, vol. 23, no. 1, pp. 50–64, 2003, ISSN: 1066-033X. DOI: 10.1109/MCS.2003.1172829.

[29] C. Atkinson and T. Kühne, "Model-Driven Development: A Metamodeling Foundation", *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, Sep. 2003, ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231149. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231149.

[30] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008, ISBN: 0321553454, 9780321553454.

[31] G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, vol. 4735, Lecture Notes in Computer Science, Springer, 2007, ISBN: 978-3-540-75208-0.

[32] ——, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008, ISBN: 0321553454, 9780321553454.

[33] "ARP4754A. Guidelines For Development Of Civil Aircraft and Systems", SAE International, Standard, 2010.

[34] "DO-178B. Software Considerations in Airborne Systems and Equipment Certification", RTCA SC-167, EUROCAE WG-12, Standard, 2010.

[35] R. Höttger, L. Krawczyk, and B. Igel, "Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems", vol. 1, no. 1, p. 6, 2014, ISSN: PISSN:2010-376X, EISSN:2010-3778. [Online]. Available: http://waset.org/abstracts/Software-and-Systems-Engineering.

[36] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd. Addison-Wesley Professional, 2009, ISBN: 0321331885.

[37] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real Time Application Interface", *Linux J.*, vol. 2000, no. 72es, Apr. 2000, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=348554.348564.

[38] Evidence srl, *ERIKA Enterprise RTOS*. [Online]. Available: http://erika.tuxfamily.org.

[39] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline Scheduling in the Linux Kernel", *Softw. Pract. Exper.*, vol. 46, no. 6, pp. 821–839, Jun. 2016, ISSN: 0038-0644. DOI: 10.1002/spe.2335. [Online]. Available: https://doi.org/10.1002/spe.2335.

[40] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux PREEMPT-RT V2.6.33 Versus V3.6.6: Better or Worse for Real-time Applications?", *SIGBED Rev.*, vol. 11, no. 1, pp. 26–31, Feb. 2014, ISSN: 1551-3688. DOI: 10.1145/2597457.2597460. [Online]. Available: http://doi.acm.org/10.1145/2597457.2597460.

[41] L. Albeni, C. Scordino, J. Lelli, and L. Palopoli, "Greedy CPU reclaiming for SCHED_DEADLINE", in *16th Real Time Linux Workshop*, 2014.

[42] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited Preemptive Scheduling for Real-Time Systems. A Survey", *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013, ISSN: 1551-3203. DOI: 10.1109/TII.2012.2188805.

[43] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing", *Journal of Systems Architecture - Embedded Systems Design*, vol. 60, no. 9, pp. 726–740, 2014. DOI: 10.1016/j.sysarc.2014.07.004. [Online]. Available: https://doi.org/10.1016/j.sysarc.2014.07.004.

[44] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-time Multi-core Virtual Machine Scheduling in Xen", in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14, New Delhi, India: ACM, 2014, 27:1–27:10, ISBN: 978-1-4503-3052-7. DOI: 10.1145/2656045.2656066. [Online]. Available: http://doi.acm.org/10.1145/2656045.2656066.

[45] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[46] V. Sinitsyn, "Jailhouse", *Linux Journal*, vol. 2015, no. 252, p. 2, 2015.

[47] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture", in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, Paris, France: ACM, 2010, pp. 209–222, ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755935. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755935.

[48] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization", *Journal of software Engineering and Applications*, vol. 5, no. 4, pp. 227–290, 2012. DOI: 10.4236/hsea.2012.540333.

[49] G. Taccari, L. Taccari, A. Fioravanti, L. Spalazzi, A. Claudi, and A. B. SA, "Embedded Real-Time Virtualization: State of the Art and Research Challenges", 2014.

[50] P. Valente and F. Checconi, "High Throughput Disk Scheduling with Fair Bandwidth Distribution", *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1172–1186, 2010, ISSN: 0018-9340. DOI: 10.1109/TC.2010.105.

[51] M. Caccamo, M. Cesati, R. Pellizzoni, E. Betti, R. Dudko, and R. Mancuso, "Real-time Cache Management Framework for Multi-core Architectures", in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, ser. RTAS '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 45–54, ISBN: 978-1-4799-0186-9. DOI: 10.1109/RTAS.2013.6531078. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2013.6531078.

[52] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems", in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11. DOI: 10.1109/RTAS.2016.7461321.

[53] J. Kiszka, "Linux-based partitioning hypervisor", 2011. [Online]. Available: https://github.com/siemens/jailhouse.

[54] *QNX Hypervisor*. [Online]. Available: http://blackberry.qnx.com/en/products/hypervisor/index.

[55] *PikeOS Hypervisor*. [Online]. Available: https://www.sysgo.com/products/pikeos-hypervisor/.

[56] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in Virtual Machine Monitors", in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08, Seattle, WA, USA: ACM, 2008, pp. 1–10, ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346258. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346258.

[57] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware Virtual Machine Scheduling for I/O Performance.", in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09, Washington, DC, USA: ACM, 2009, pp. 101–110, ISBN: 978-1-60558-375-4. DOI: 10.1145/1508293.1508308. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508308.

[58] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality I/O virtualization", in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ACM, 2009, p. 12.

[59] H. Chen, H. Jin, K. Hu, and M. Yuan, "Adaptive Audio-aware Scheduling in Xen Virtual Environment", in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*, ser. AICCSA '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8, ISBN: 978-1-4244-7716-6. DOI: 10.1109/AICCSA.2010.5586974. [Online]. Available: http://dx.doi.org/10.1109/AICCSA.2010.5586974.

[60] M. Kesavan, A. Gavrilovska, and K. Schwan, "On disk I/O scheduling in virtual machines", in *Proceedings of the 2nd conference on I/O virtualization*, USENIX Association, 2010, pp. 6–6.

[61] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia, “I/O Scheduling Model of Virtual Machine Based on Multi-core Dynamic Partitioning”, in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC ’10, Chicago, Illinois: ACM, 2010, pp. 142–154, ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851494. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851494.

[62] A. Landau, M. Ben-Yehuda, and A. Gordon, “SplitX: Split Guest/Hypervisor Execution on Multi-Core.”, in *WIOV*, 2011.

[63] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, “A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems”, in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2016, pp. 1–11.

[64] P. Zhao and G. Tan, “Evaluating I/O Scheduling in Virtual Machines Based on Application Load”, *Engineering Journal*, vol. 17, no. 3, pp. 105–112, 2013.

[65] L. Cherkasova and R. Gardner, “Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor”, in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05, Anaheim, CA: USENIX Association, 2005, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247384.

[66] M. Yi, D. H. Kang, M. Lee, I. Kim, and Y. I. Eom, “Performance Analyses of Duplicated I/O Stack in Virtualization Environment”, in *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*, ACM, 2016, p. 26.

[67] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, “Understanding the effects of hypervisor I/O scheduling for virtual machine performance interference”, in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, pp. 34–41. DOI: 10.1109/CloudCom.2012.6427495.

[68] A. Reddy and J. Wyllie, “Disk scheduling in a multimedia I/O system”, in *Proceedings of the first ACM international conference on Multimedia*, ACM, 1993, pp. 225–233.

[69] 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.

[70] H. Jin, X. Ling, S. Ibrahim, W. Cao, S. Wu, and G. Antoniu, “Flubber: Two-level disk scheduling in virtualized environment”, *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2222–2238, 2013.

[71] X. Ling, H. Jin, S. Ibrahim, W. Cao, and S. Wu, “Efficient Disk I/O Scheduling with QoS Guarantee for Xen-based Hosting Platforms”, in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID ’12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 81–89, ISBN: 978-0-7695-4691-9. DOI: 10.1109/CCGrid.2012.17. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2012.17.

[72] T.-Y. Chen, H.-W. Wei, Y.-J. Chen, W.-K. Shih, and T.-s. Hsu, “Integrating deadline-modification SCAN algorithm to Xen-based cloud platform”, in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, IEEE, 2013, pp. 1–4.

[73] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems", *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 4, pp. 837–863, Nov. 2004, ISSN: 1539-9087. DOI: 10.1145/1027794.1027801. [Online]. Available: http://doi.acm.org/10.1145/1027794.1027801.

[74] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-Associative Sector Translation for NAND Flash Memory in Real-time Systems", in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09, Nice, France: European Design and Automation Association, 2009, pp. 507–512, ISBN: 978-3-9810801-5-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874620.1874745.

[75] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free", in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[76] M. Lauer, F. Boniol, C. Pagetti, and J. Ermont, "End-to-end latency and temporal consistency analysis in networked real-time systems", *IJCCBS*, vol. 5, no. 3/4, pp. 172–196, 2014. DOI: 10.1504/IJCCBS.2014.064667. [Online]. Available: https://doi.org/10.1504/IJCCBS.2014.064667.

[77] T. A. Henzinger, B. Horowitz, and M. Kirsch, "Embedded Control Systems Development with Giotto", in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '01, Snow Bird, Utah, USA: ACM, 2001, pp. 64–72, ISBN: 1-58113-425-8. DOI: 10.1145/384197.384208. [Online]. Available: http://doi.acm.org/10.1145/384197.384208.

[78] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains", in *The 22th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2016. [Online]. Available: http://www.es.mdh.se/publications/4368-.

[79] ——, "End-to-end timing analysis of cause-effect chains in automotive embedded systems", *Journal of Systems Architecture*, vol. 80, no. Supplement C, pp. 104 –113, 2017, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.09.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762117300681.

[80] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines", in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, 1990, pp. 201–209. DOI: 10.1109/REAL.1990.128748.

[81] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited Preemptive Scheduling for Real-Time Systems. A Survey", *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013, ISSN: 1551-3203. DOI: 10.1109/TII.2012.2188805.

[82] I. Sanudo, P.Burgio, and M.Bertogna, "Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System", in *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'16), in conjuction with the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016), Toulouse, France, July 2016*, 2016.

[83] A. Biondi and G. Buttazzo, "Engine control: Task modeling and analysis", in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 525–530. DOI: 10.7873/DATE.2015.0147.

[84] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics", in *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*, IEEE Communications Society, 2009.

[85] A. Hamann, D. Ziegenbein, S. Kramer, and M. Lukasiewyz, "Demo Abstract: Demonstration of the FMTV 2016 Timing Verification Challenge", in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–1. DOI: 10.1109/RTAS.2016.7461330.

[86] P. Marti, R. Villa, J. M. Fuertes, and G. Fohle, "On real-time control tasks schedulability", in *2001 European Control Conference (ECC)*, 2001, pp. 2227–2232.

[87] S. Lampke, S. Schliecker, D. Ziegenbein, and A. Hamann, "Resource-Aware Control-Model-Based Co-Engineering of Control Algorithms and Real-Time Systems", 2015-01-0168, vol. 8, 2015, pp. 106–114.

[88] The Hercules Consortium, *Hercules – High-Performance Real-time Architectures for Low-Power Embedded Systems*. [Online]. Available: http://hercules2020.eu/.

[89] L. M. Pinho, V. Nélis, P. M. Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, and M. Mardiak, "P-SOCRATES: A parallel software framework for time-critical many-core systems", *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 39, no. 8, pp. 1190–1203, 2015. DOI: 10.1016/j.micpro.2015.06.004. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2015.06.004.

[90] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future", in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, B. Lisper, Ed., ser. OpenAccess Series in Informatics (OASIcs), The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7, vol. 15, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146, ISBN: 978-3-939897-21-7. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2010.136. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2833.

[91] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, and C. R. et al, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research", in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, M. Schoeberl, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 55, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–10, ISBN: 978-3-95977-025-5. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2016.2. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6895.

[92] E4Coder, *The toolset for simulation and code generation for embedded devices*. [Online]. Available: http://www.e4coder.com/.

[93] P. Parra, A. Viana, O. Rodriguez, M. Knoblauch, F. Alcojor, S. Sanchez, I. Garcia, O. Garcia, and D. Meziat, "EdROOM. Automatic C++ Code Generator for Real Time Systems Modelled with ROOM", in *Actas del Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones. Sitges, Spain, October 3, 2006*, 2006. [Online]. Available: http://ceur-ws.org/Vol-227/paper12.pdf.

[94] L. Carnevali, D. D'Amico, L. Ridi, and E. Vicario, "Automatic Code Generation from Real-Time Systems Specifications", in *Proceedings of the Twentieth IEEE/IFIP International Symposium on Rapid System Prototyping, Shortening the Path from Specification to Prototype, RSP 2009, Paris, France, 23-26 June 2009*, 2009, pp. 102–105. DOI: 10.1109/RSP.2009.24. [Online]. Available: http://dx.doi.org/10.1109/RSP.2009.24.

[95] B. Kim, L. T. X. Phan, O. Sokolsky, and I. Lee, "Platform-dependent code generation for embedded real-time software", in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013*, 2013, 8:1–8:10. DOI: 10.1109/CASES.2013.6662512. [Online]. Available: http://dx.doi.org/10.1109/CASES.2013.6662512.

[96] E. Salazar, A. Alonso, M. A. de Miguel, and J. A. de la Puente, "A Model-Based Framework for Developing Real-Time Safety Ada Systems", in *Reliable Software Technologies - Ada-Europe 2013, 18th Ada-Europe International Conference on Reliable Software Technologies, Berlin, Germany, June 10-14, 2013. Proceedings*, 2013, pp. 127–142. DOI: 10.1007/978-3-642-38601-5_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38601-5_9.

[97] G. C. Buttazzo and G. Lipari, "Ptask: An educational C library for programming real-time systems on Linux", in *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, ETFA 2013, Cagliari, Italy, September 10-13, 2013*, 2013, pp. 1–8. DOI: 10.1109/ETFA.2013.6648001. [Online]. Available: http://dx.doi.org/10.1109/ETFA.2013.6648001.

[98] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators", in *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, 2015, pp. 1–8. DOI: 10.1109/RTEST.2015.7369851.

[99] OMG, *MOF 2.0/XMI Mapping V. 2.1.1. Technical report, Object Management Group*, 2007.

[100] A. G.-D.R. P. Dimitris Kolovos Louis Rose, *The Epsilon Book. The Eclipse Foundation.* 2014.

[101] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "The Epsilon Generation Language", in *Model Driven Architecture – Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, I. Schieferdecker and A. Hartman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–16, ISBN: 978-3-540-69100-6. DOI: 10.1007/978-3-540-69100-6_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69100-6_1.

[102] Kalray Corporation, *Many-core Kalray MPPA*, [Online] http://www.kalray.eu/, 2012.

[103] NVIDIA, "NVIDIA Tegra X1 White Paper, NVIDIAś new mobile SuperChip", *NVIDIA Corporation*, 2015. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[104] ——, "NVIDIA Tegra K1 White Paper, A New Era in Mobile Computing", *NVIDIA Corporation*, 2014. [Online]. Available: http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf.

[105] *CUDA Toolkit Documentation v7.0*, http://docs.nvidia.com/cuda/index.html, Accessed: July, 30th 2015.

[106] Infineon Technologies AG, *Tricore AURIX Family*. [Online]. Available: {http://www.infineon.com}.

[107] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period Optimization for Hard Real-time Distributed Automotive Systems", in *2007 44th ACM/IEEE Design Automation Conference*, 2007, pp. 278–283.