Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Application Modelling and Performance Estimation of Mixed-Critical Embedded Systems

Von der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Philipp Ittershagen
geboren am 24. Oktober 1986 in Quakenbrück

Philipp Ittershagen: *Application Modelling and Performance Estimation of Mixed-Critical Embedded Systems*

GUTACHTER:
Prof. Dr.-Ing. Wolfgang Nebel

WEITERE GUTACHTER:
Prof. Dr. Martin Fränzle

TAG DER DISPUTATION:
23. November 2018

ABSTRACT

The focus of the challenges in developing embedded systems is significantly shifting towards the integration of multiple subsystems on a powerful single platform while considering the cost, size, and power consumption constraints as well as the heterogeneous, domain-specific requirements which are characteristic in the domain of embedded systems. The rise of mixed-critical embedded systems moreover imposes novel challenges on the specification, development, and functional validation in the design flow. In the emerging dynamic scheduling context of mixed-criticality platforms, the system behaviour needs to be estimated in an early step in the design flow to assess the integration impact, especially for quality of service-driven, low-critical subsystems.

This work provides a modelling and integration flow for specifying, estimating, and evaluating software functions, ranging from an initial executable specification to an implementation candidate on a Multi-Processor System-on-a-Chip. The flow starts with a mixed-criticality programming model proposed to express safety- as well as performance-critical functional behaviour along with their real-time requirements. A component-based refinement flow then provides an implementation candidate on a contemporary MPSoC. The integrated measurement infrastructure then allows for systematically analysing the functional behaviour as well as the overhead of mechanisms for isolating, managing, and observing the refined mixed-criticality subsystems. Based on a data-driven model to evaluate dynamic resource consumption effects of high-critical subsystems, the thesis proposes a method for constructing workload models of safety-critical software components on the target platform. The evaluation of this work demonstrates that these models can support mixed-criticality system integration scenarios where intellectual property issues may prevent integration or the need for fast time-to-market goals require a decoupled development and integration phase of mixed-critical applications.

ZUSAMMENFASSUNG

Der Fokus der Herausforderungen in der Entwicklung eingebetteter Systeme verschiebt sich immer mehr in Richtung der Integration mehrerer Teilsysteme auf einer gemeinsamen, leistungsfähigen Plattform. Dabei spielt die gleichzeitige Betrachtung der Einschränkungen bezüglich Kosten, Größe und Leistungsaufnahme sowie der für eingebettete Systeme charakteristischen heterogenen, domänenspezifischen Anforderungen eine entscheidende Rolle. Der Trend zu gemischt-kritischen eingebetteten Systemen stellt darüber hinaus neue Herausforderungen an die Spezifikation, Entwicklung und funktionale Validierung in deren Entwicklungsfluss. Durch den Einzug dynamischer Scheduling-Verfahren in gemischt-kritischen Plattformen ist eine Abschätzung des Systemverhaltens zur Bewertung der Auswirkung der Integration von Teilsystemen bereits in den ersten Schritten des Entwicklungsflusses erforderlich, insbesondere bei der Integration von niedrigkritischen, serviceorientierten Funktionen.

In dieser Arbeit wird ein Modellierungs- und Integrationsfluss zur Spezifikation, Abschätzung und Bewertung von Softwarefunktionen, ausgehend von einer initialen, ausführbaren Spezifikation bis hin zu einer Implementation auf einem Multi-Processor System-on-a-Chip, vorgestellt. Dazu wird zunächst ein Programmiermodell für gemischt-kritische Systeme eingeführt, in dem sicherheitskritisches sowie leistungsorientiertes funktionales Verhalten zusammen mit ihren Echtzeitanforderungen ausgedrückt werden kann. Eine komponentenbasierte Verfeinerungsstrategie erlaubt dann die prototypische Entwicklung eines Implementationskandidaten auf einem Multi-Processor System-on-a-Chip. Darüber hinaus ermöglicht eine integrierte Messinfrastruktur, den zeitlichen Mehraufwand der Mechanismen zur Isolation, Kontextverwaltung und zeitlichen Beobachtung der gemischtkritischen Teilsysteme systematisch zu analysieren. Basierend auf einem datengetriebenen Modell zur Bewertung der dynamischen Nutzung von Plattformressourcen des sicherheitskritischen Teilsystems wird in dieser Arbeit anschließend eine Methode zur Konstruktion von Auslastungsmodellen für die sicherheitskritischen Teilsysteme auf der Zielplattform vorgeschlagen. Die Bewertung in dieser Arbeit zeigt, dass durch diese Modelle die Integrationsszenarien gemischt-kritischer Systeme unterstützt werden können, bei denen der Schutz geistigen Eigentums eine Integration erschweren könnte oder bei denen kurze Vorlaufzeiten für die Marktreife eine Entkoppelung der Entwicklung von der Integration gemischt-kritischer Anwendungen erfordern.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[1] Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel. 'Mixed-Criticality System Modelling with Dynamic Execution Mode Switching'. In: *Proceedings of the Forum on Specification and Design Languages (FDL'15)*. Barcelona, Spain, 2015. DOI: 10.1109/fdl.2015.7306356.

[2] Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel. 'A Task-level Monitoring Framework for Multi-processor Platforms'. In: *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES'16)*. ACM Press, May 2016. DOI: 10.1145/2906363.2906373.

[3] Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel. 'An Integration Flow for Mixed-Critical Embedded Systems on a Flexible Time-triggered Platform'. In: *ACM Trans. Des. Autom. Electron. Syst.* 23.4 (Apr. 2018). DOI: 10.1145/3190837.

[4] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner and Wolfgang Nebel. 'A Workload Extraction Framework for Software Performance Model Generation'. In: *Proceedings of the 7th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'15)*. ACM Press, Jan. 2015. DOI: 10.1145/2693433.2693436.

[5] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner and Achim Rettberg. 'Hierarchical Real-time Scheduling in the Multi-core Era – An Overview'. In: *Proceedings of the 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'13)*. IEEE, June 2013. DOI: 10.1109/isorc.2013.6913241.

CONTENTS

Appendix

1 INTRODUCTION

The design of embedded systems is currently facing a paradigm shift, driven by the rising complexity of hardware platforms and the economic need for increasing the number of functionalities on each device. Until recently, embedded systems consisted of a dedicated set of functions executed on a single hardware device. However, the amount of embedded devices in large-scale systems such as aircraft and automobiles currently rises significantly to the point where an isolated approach of developing and integrating these devices on separate hardware architectures becomes infeasible due to the size, weight, and power consumption requirements of each individually packaged hardware component.

Due to the continuous improvements in chip design manufacturing techniques over the last decades, the physical size of transistors has been reduced significantly, resulting in an ever-increasing transistor count per chip area. Moore's Law originally states that the transistor count on a fixed area doubles every 18 months. In recent years however, transistors are slowly reaching the physical limitations in terms of their structural size. The chip manufacturing industry is thus racing towards the end of Moore's Law, an era which has lasted for more than fifty years [58].

Moore's Law however has not only predicted the structural improvements in terms of transistor density. The symptom of these improvements, the doubling of computing power every 18 months, still prevails, even after reaching the physical limits in current manufacturing processes. The trend in Figure 1.1 shows that the overall transistor count is still increasing exponentially. As a consequence, the chip area of contemporary hardware designs continues to grow. Advances in system-level design methodologies and design automation additionally make it feasible to utilise the increased chip area by specifying and implementing chip layouts consisting of multiple complex hardware/software components on a so-called Multi-Processor System-on-a-Chip (MPSoC). Such platforms feature multiple processing elements and complex interconnect hierarchies, and therefore contribute in further increasing the possible feature density of functionality in embedded devices.

Another effect of the advances in chip design is the increase in the number of devices due to lower manufacturing costs. Its consequences can be observed in the automotive domain, as the number of devices per automobile constantly grows due to consumer needs and the requirements on driver assistance systems. However, such embedded systems usually have tight constraints on the space, weight, and power consumption due to their integration environment. As a result, the automotive industry is looking into methods for mapping multiple (independent) functionalities to powerful MPSoCs to save on the wiring and packaging overhead compared to isolated device solutions.

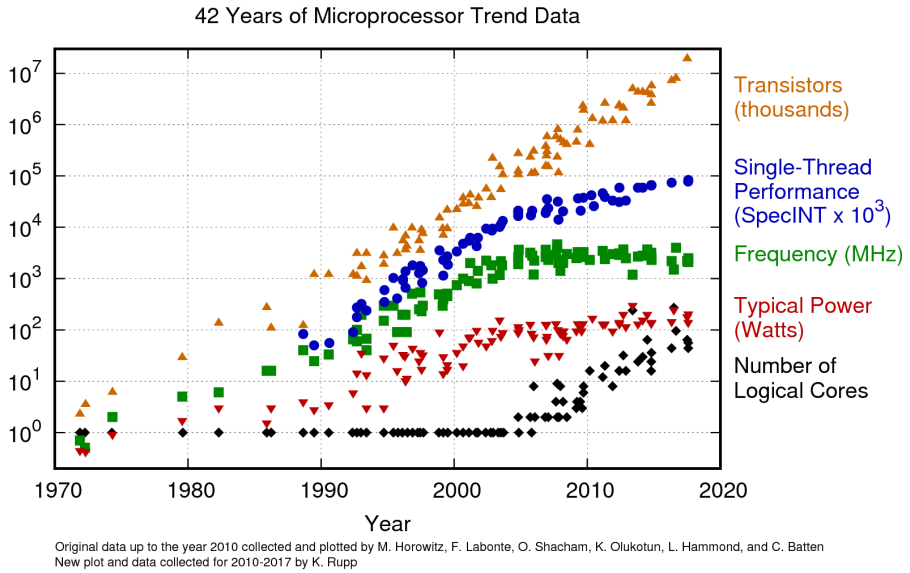


Figure 1.1: 42 years of microprocessor trend data, covering transistor count, single-thread performance, frequency, power consumption, and the number of logical cores (from [70]).

Advances in chip design cause the embedded systems design process to be heavily shaped by the rise of complex MPSoCs, because the resulting hardware platforms allow an increased feature density through the integration of a wide range of functionality. Such powerful hardware platforms are furthermore a key enabling technology for the emerging Cyber-Physical Systems (CPS) domain. This is because, compared to single-core devices, these platforms provide superior processing power and heterogeneous computation resources, such as general-purpose processing elements or dedicated real-time processors, and even user-logic for application-specific logic cores. CPS operate in the *physical* domain through the use of sensors and actors for gathering information from their physical environment and the *cyber* domain through their integration in large-scale cross-application communication networks. They impose novel challenges to the design process due to their demand in computing power for communication as well as their requirements towards extra-functional quantities such as time, power or temperature for embedding them into the physical environment.

Embedded systems design flows play a key role in establishing an economically feasible product: design methodologies, which cover the process of stating system requirements and provide a guided, stepwise refinement towards an implementation candidate, form the cornerstone of modern embedded systems design automation. The heterogeneous requirements of CPS cause academics and industry to face novel challenges in the context of designing, validating, and integrating CPS, as stated by Tripakis [74] due to the combination of computational demand and extra-functional requirements.

The economic advantage of such MPSoCs and the need for increasing functionality to meet the demand of emerging CPSs has lead to efforts for consolidating multiple functionalities on the same hardware platform, often times

categorised into different criticalities. In fact, CPS by definition consist of multiple functionalities that require different levels of assurance regarding their criticality. Depending on the use-case, sensor data processing and actor operations may require a high degree of reliability and therefore possess real-time requirements, for example to avoid over- and under-sampling or to react on environmental changes in a specified interval. On the other hand, a CPS may also provide functionality for user interaction and other best-effort, non-critical operations such as infotainment. As such, most CPS can thus be labelled as *mixed-critical systems*, since they are composed of functions with different criticalities.

1.1 CONTEXT

Until recently, the design methodology for embedded systems focussed on implementing functionality on an isolated HW/SW platform. In such designs, the functionality and the extra-functional properties such as timing behaviour and power consumption are estimated in the context of the dedicated hardware platform. Certification authorities from safety-critical embedded domains such as automotive and avionics require that a Failure Mode and Effect Analysis (FMEA) assess the system's criticality in the context of the environment. The criticality of a function is usually determined by its failure impact. Mitigation techniques and methods need to be implemented in these systems to achieve a criticality-dependant maximum failure probability. Possible mitigations for such failures include isolating the device from other parts of the system, providing backup power wiring, or implementing the function in a redundant setup. Besides their functional correctness, real-time systems adhere to the timing requirements of the function they perform. As such, they are considered malfunctioning if the function is unable to meet the specified timing deadline. Therefore, fault mitigation in real-time systems considers the execution time of functions implemented on a platform. In the context of this thesis, we consider the term mixed-criticality as the combination of functions with different real-time criticalities on the same hardware platform.

Combining systems (with different criticalities) on a single platform invalidates the timing guarantees stated in an isolated environment and need to be reconsidered for the integrated platform. A manual integration and revalidation process can quickly become economically infeasible. Embedded systems design flows thus provide tools and methods for estimating platform timing behaviour already in the specification and refinement phases. When functions with different criticalities are integrated on the same platform, any inter-dependencies which may impact their timing behaviour need to be considered rigorously.

Cullmann et al. [18] have shown that pessimistic timing estimates on complex MPSoC can deviate from their observed worst-case behaviour considerably. Additionally, integrating more functionality on the same platform also raises the potential for timing interferences, thereby increasing such pessimistic timing estimates even more. These factors reduce the design space considerably when integrating functionality on a MPSoC, causing an *utilisation*

gap between the estimated timing behaviour of safety-critical functionality at design-time and its average timing behaviour observed at run-time.

Research regarding mixed-criticality real-time task models and run-time strategies for observing the dynamic execution time behaviour of mixed-critical functionality consider reducing the utilisation gap. At the same time, they can guarantee the temporal requirements specified at design-time [14]. Such mixed-criticality models provide means for describing *criticality scenarios* such that task models contain multiple execution time estimates differing in their level of assurance [33]. When a high-critical function exhibits average-case execution time behaviour, computational resources are optimistically allocated to lower-critical functions and dynamically re-allocated in case the task overruns the optimistic timing guarantees at run-time. Such dynamic resource management methods are especially interesting in the context of mapping safety-critical and performance-critical functionality to the same platform. While timing guarantees are given for the former, the latter can exploit the high-performance resources available on MPSoC platforms.

1.2 MOTIVATION

When designing performance-critical functions which adhere to Quality of Service (QoS) metrics, the refinement decisions typically depend on post-integration performance estimates, obtained either using suitable performance models or measurements on the platform. A *mixed-criticality integration flow* therefore needs provide accurate models for evaluating QoS metrics while considering the mapped safety-critical functions, the platform segregation overhead, and the impact of dynamic resource utilisation artefacts caused by dynamic mixed-criticality scheduling policies. The challenge in the integration process for mixed-criticality systems manifests in modelling the dynamic application behaviour which criticality-aware scheduling techniques exploit for increasing resource utilisation on the platform.

We claim that reducing the manual refinement step is a key factor for managing the design complexity in mixed-criticality system design. In order to rapidly provide an implementation candidate, a design and integration flow needs to consider mixed-criticality properties along their functional behaviour already at the specification level and provide tool-assisted deployment by enabling a systematic configuration of platform segregation techniques. We argue that omitting mixed-criticality properties at the system-level prevents the designer from performing scheduling analyses and delays the evaluation of partitioning and mapping decisions. Furthermore, integrating the functional behaviour only in a late step in the design flow limits the possibilities for re-evaluating design decisions regarding partitioning and mapping.

Mixed-criticality scheduling policies combine worst-case upper-bounded timing properties with average-case estimates that only give a lower level of assurance on the temporal behaviour of a software component. The timing behaviour variations of these tasks affect the dynamic scheduling policy which influences the execution performance of lower-critical components mapped to the same platform. Suitable models which take into account this dynamic timing and scheduling behaviour are therefore essential to

provide feedback during the design flow for a performance estimation in lower-critical subsystems. Analysing the performance behaviour of lower-critical function implementations in mixed-criticality platforms using worst-case timing models of safety-critical subsystems fails to take into account the dynamic scheduling behaviour, since they do not consider the average-case timing behaviour which can significantly deviate, especially when considering MPSoC platforms. As a result, a mixed-criticality design flow needs to consist of timing models which consider the dynamic scheduling behaviour to provide accurate modelling results for estimating performance metrics.

The complexity of mixed-criticality software mapped to MPSoC systems, caused by the implemented segregation techniques and the requirements on the component's functionality is constantly rising. However, due to the coupling effects of the dynamic scheduling behaviour in mixed-criticality systems, the performance impact when integrating functionalities is only visible when all components are integrated on the platform. Additional re-iterations regarding functional refinement or other design decision require the full platform to be integrated again to re-evaluate the performance goals. This traditional embedded systems workflow may ultimately render many products economically infeasible due to the added cost of development and time consumed by the integration process. Therefore, providing timing models for performance estimation is necessary but not sufficient. In mixed-criticality systems, the high degree of complexity regarding the software stack, especially in performance-critical systems, requires extensive implementation support for the integration on a platform. Such systems implementing heterogeneous use-cases and design assurance levels require an integration environment that allows to decouple the functional refinement from the performance estimation.

1.3 SCOPE & RESEARCH QUESTIONS

In this thesis, we address the challenges of designing and integrating embedded mixed-critical software components on a common platform. In particular, we focus on the effect of computation resource sharing effects and segregation techniques for *dual-criticality* systems. They consist of functionalities which can be categorised along their criticality into *high-* and *low-critical* which we denote as `HI`- and `LO`-critical, respectively. The categorisation of these functionalities can be derived from a domain-specific FMEA. In such mixed-criticality systems, the `HI`-critical functionality needs to be designed and implemented according to tight temporal constraints. On the other hand, the `LO`-critical function generally represents a best-effort functionality where the designer is interested in the overall performance behaviour in terms of domain-specific metrics such as *frames per second*, *average bandwidth*, or *requests per minute*. In contrast to the `HI`-critical subsystem, these functions do not have any real-time requirements except their performance metrics. Summarising, a `HI`-critical subsystem represents a function with guaranteed real time requirements, while a `LO`-critical component denotes a purely performance-driven software subsystem with requirements on QoS metrics that need to be estimated early on in the design process.

Currently, on the one hand, research is addressing the question of how to sufficiently separate mixed-criticality systems both in the functional and extra-functional sense to eliminate coupling regarding behaviour, timing, power consumption, and temperature. On the other hand, models have been proposed which aim at taking into account the dynamic aspects of run-time behaviour of mixed-criticality systems to allow a more sophisticated scheduling by considering the utilisation gap caused by design-time worst-case assumptions and observed average run-time behaviour. These activities consider platform-level segregation and isolation techniques, as well as activities regarding the improvement of resource utilisation.

The first scientific context of this thesis focusses on how the application-level mixed-criticality requirements can be specified, systematically considered, and refined towards an implementation candidate, while being able to assess the integration impact regarding timing behaviour of mixed-criticality applications and segregation techniques at multiple stages in the design flow. The focus of this work is a technical realisation of a mixed-criticality system-level specification model and the proposal of evaluation models suitable for performance estimation of `LO`-critical functionality integrated with a `HI`-critical software component. In particular, this work considers the dynamic aspects of contemporary mixed-criticality scheduling policies by providing suitable models that capture this dynamic temporal behaviour and the resulting impacts on platform utilisation. The second scientific context of this thesis is the assessment on how to capture dynamic mixed-criticality scheduling effects in timing models of the platform and application. We formulate the following scientific questions regarding a mixed-criticality system-level design flow:

1. How can mixed-criticality properties of safety- and performance-critical software applications be considered along their functional behaviour in an embedded systems design flow?

2. How can the functional description along with the mixed-criticality properties be used to derive an implementation on an MPSoC?

3. What are the platform requirements that fulfil the needs of mixed-criticality systems in terms of temporal and spatial segregation?

4. What dynamic timing properties arise from the application behaviour as well as the platform-level scheduling behaviour in an integrated mixed-criticality system?

5. How can we generate models which represent the integration impact of dynamic mixed-criticality timing properties in such systems?

6. What is the timing accuracy of the models derived from the measured safety-critical temporal behaviour in terms of their estimates for performance-critical applications?

7. How can these timing models be used to separate the individual integration process of different subsystems in a mixed-criticality system?

To summarise, this work addresses the main scientific question on how to provide a design flow and integration environment for mixed-critical systems consisting of safety-critical and performance-critical software components. In particular, the work focusses on the criticality of these systems on the specification level, and the question of how to provide a refinement flow towards an implementation candidate on an MPSoC and suitable models along the design flow to assess the dynamic mixed-criticality timing behaviour. Chapter 4 will revisit these questions and formulate the contributions of this thesis.

1.4 OUTLINE

The thesis is organised into three parts. The first part provides an overview of the foundations for this thesis in Chapters 2 and 3. Based on the scientific context and the research questions stated above, Chapter 4 defines the contributions, while the related scientific work is discussed Chapter 5.

The second part starts with Chapter 6 containing the presentation of OSSS/MC, the proposed mixed-critical system design flow. After specifying the modelling components and their implementation on a target, Chapter 7 describes the performance modelling approach for the proposed models. Section 7.1 then presents the measurement infrastructure that has been integrated into the implementation of the programming model components on the target platform. Finally, Section 7.2 discusses the data-driven modelling approach, while Section 7.3 presents the application proxy implementation.

The third part of this thesis contains the evaluation of the contributions and discusses the results. The evaluation of the integration flow is presented in Chapter 8, with a focus on the implications of a target implementation in terms of its segregation properties. Chapter 9 focusses on evaluating the data-driven performance modelling approach as well as the proxy generation approach embedded in our design flow. Chapter 10 concludes the thesis and discusses possible future activities.

Part I

FOUNDATIONS

2 SYSTEM-LEVEL DESIGN

This chapter provides the foundations concerning the complexity of designing embedded systems. We first describe a framework for categorising different design methodologies. Next, we take a closer look at each step in the design flow and present the possible choices and highlight the challenges the designer has to overcome. We will then present Oldenburg System Synthesis Subset, a methodology for HW/SW co-design featuring high-level synthesis and automatic target code generation.

In the next chapters, we discuss the term *mixed-criticality* and provide an overview of established and current efforts as well as research areas regarding the mapping of different criticality functions on a common platform. Based on these topics, we then re-visit the scientific questions stated in the previous chapter and derive the main contributions of this thesis: the extension of Oldenburg System Synthesis Subset (OSSS) in terms of mixed-criticality and the modelling and analysis of timing artefacts in these systems.

Designing and implementing an embedded system is challenging due to several factors. First of all, the term *embedded* denotes the connection between the system and its physical environment. As such, an embedded system acts with real-world components and therefore possesses constraints such as timing behaviour, power consumption, or even physical size. These properties are the result of the implementation strategies of its *functional behaviour* and depend on the physical and temporal properties of the implemented system, such as the overall structural layout and the physical components used. The *refinement* of a functional behaviour denotes the transformation from an abstract description to a more detailed representation. Therefore, *extra-functional properties* depend on the choices made during the refinement of *functional properties* to a physical implementation.

Depending on the operating context of an embedded system, these requirements have varying degrees of importance to its overall functionality. Therefore, in the embedded systems design context, one of the key challenges is the accurate estimation, validation, and even verification of extra-functional properties during the design flow. On the one hand, a design process should provide models for estimating the expected extra-functional behaviour, and on the other hand, the process needs to incorporate proper validation and verification methods to guarantee that the system operates within the expected boundaries.

As a consequence, the embedded systems design flow particularly considers modelling techniques to capture the desired functional behaviour and their extra-functional constraints. The refinement process of an embedded system consists of a series of decisions that affect the final system behaviour and its extra-functional properties. Some of these questions are:

- How should the functional behaviour be partitioned on the available system components?

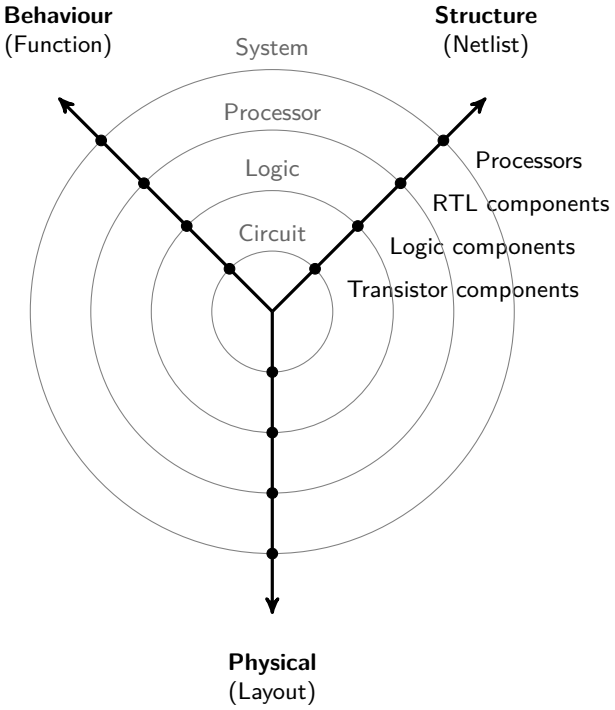


Figure 2.1: Gajski-Kuhn Y-Chart [26].

- What are the performance impacts on a particular partitioning configuration?
- Does the mapped functional behaviour meet its real-time requirements when implemented on this software processing unit?
- What is the interference impact of functionality implemented on the platform in terms of their temporal behaviour?
- Do we need to separate the functionalities spatially and temporally on the platform to avoid unbounded timing interferences?

These questions need to be addressed by providing suitable models that provide feedback in the design process. Furthermore, to compare the impact of these design decisions, a systematic transformation from an initial functional specification towards an implementation candidate is an essential requirement for automating embedded systems design.

A *design methodology* refers to a collection of modelling techniques and transformations, as stated by Gajski et al. [26]. These modelling techniques need to be suitable to express the functional and extra-functional behaviour in all stages of the design flow. Gajski et al. [26] further provides a classification of design methodologies as a means to categorise and reason about the different challenges for each approach. The following section provides an overview of the key design methodologies prevalent in the Electronic Design Automation (EDA) industries and their integration challenges with the help of the *Y-Chart* [26] depicted in Figure 2.1.

The Y-Chart provides a common view on different design methodologies that have been proposed in academics and used in the EDA industry.

It provides a common structure to represent design techniques used in the early years of circuit design, but is also capable of providing means to discuss methodologies that are used today. The chart contains three axes which represent different modelling views on a system. Each view consists of multiple levels of abstraction for categorising their corresponding models. The three axes describe the system design in terms of behaviour, structure, and physical layout. At the behavioural axis, models describe *what the system should do* in terms of its functional behaviour. In the structural level, the modelling focus lies on *how the functionality is implemented* in terms of electrical components and their structural connections. The third axis, the physical level, describes how these components are allocated and arranged on a physical chip layout.

Starting with a pure-functional description of the system, the Y-Chart provides an illustration on how to refine the behavioural model towards an implementation. Design methodologies systematically transform models from different levels of abstraction to different axes. Transforming a behavioural description to a structural representation is denoted as a *synthesis*. Depending on the level of abstraction, the synthesis process targets different *component libraries* on the structural axis. For example, on the behavioural logic level, a suitable model for representing functionality is the boolean algebra. A synthesis of boolean equations (representing the behaviour) results in a netlist consisting of logic gates (representing the structure). On the System Level – the highest abstraction level in the chart – a suitable model for specifying functional behaviour is the Hierarchical Concurrent Finite State Machine. In the synthesis step, these state machines are then implemented by mapping them on a netlist consisting of processing elements. Gajski et al. [26], categorise multiple system-level design methodologies using the Y-chart. This section focusses on two main categories of design methodologies, the vertical top-down or bottom-up design methodology and the meet-in-the-middle approach, in particular *platform-based design*.

2.1 TOP-DOWN & BOTTOM-UP APPROACH

The *top-down approach* is illustrated in Figure 2.2 and starts with a system-level description of its behavioural components, for example in the form of Hierarchical Concurrent Finite State Machine (HCFSM). Next, a synthesis step transforms these behavioural models into their structural representation consisting of components from the processor library. After synthesising the structural layout in terms of these processor components, the next step involves refining the behaviour of each processor component. Here, the behaviour mapped to a single processor instance is synthesised towards a structural representation consisting of RT-level components. Similarly, further refinement steps are now performed on each lower level of abstraction until the structural representation consisting fo transistor components has been achieved. The final step involves transforming the structural transistor layout to a physical layout and completes the top-down approach.

In contrast, the *bottom-up approach* focusses on providing building blocks of low-level component libraries on the structural axis and is based on the

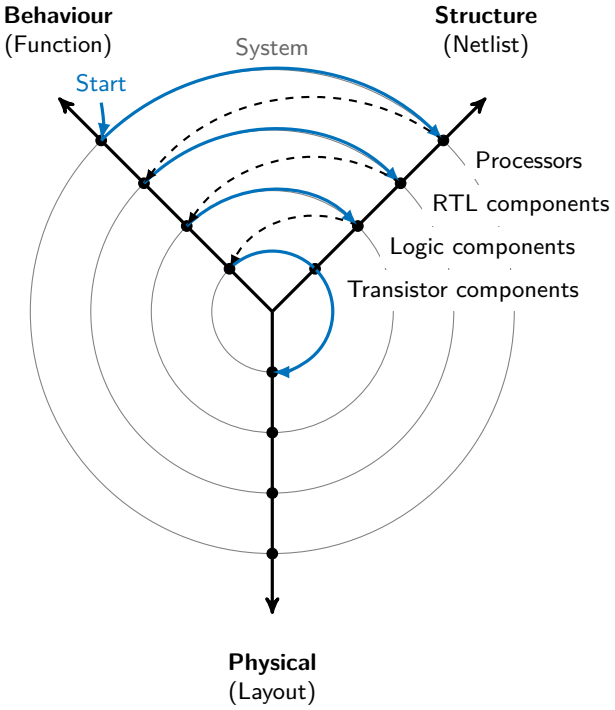


Figure 2.2: Top-Down Approach illustrated in the Y-Chart [26].

idea of re-using these components on each lower level of abstraction to construct more complex components at higher levels of abstraction. Components are designed on a given abstraction layer and then provided for re-use in component libraries. To increase flexibility, the low-level components are often able to be parametrized to adjust their use in the higher level of abstraction. On each abstraction layer, such a component library contains the behavioural, structural, and physical representation of the modelling components. Designing a system with the bottom-up approach therefore involves selecting the existing modelling components from the library and configuring them according to the requirements. The clear levels of abstraction using these component interfaces allow teams to communicate effectively when working on the same design.

Looking at both approaches, we can see that, in order for the design flow implementation to be economically viable, the top-down approach must be guided with the help of automated modelling transformation, a key component in the design automation of embedded systems. Otherwise, a manual model transformation and validation needs to be performed for every design decision at any level of abstraction. In the case of the bottom-up approach, the advantage of clear levels of abstraction are quickly diminished by the reduced flexibility and the effort of parametrizing and characterizing each component due to the unknown context they will be used in. This ultimately results in a reduced flexibility in the design of embedded systems and increases the time-to-market duration.

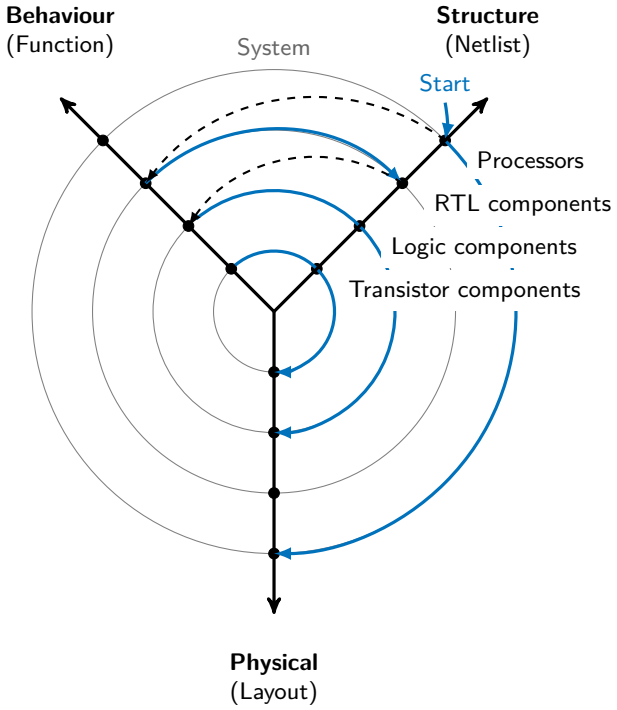


Figure 2.3: Platform-Based Design illustrated in the Y-Chart [26].

2.2 PLATFORM-BASED DESIGN

A meet-in-the-middle approach for designing embedded systems is *platform-based design*, as described by Sangiovanni-Vincentelli and Martin [71]. This approach tries to provide a trade-off between the advantages of flexibility regarding the top-down approach and the component reuseability of the bottom-up approach. The key principle of the platform-based-design is the definition of a platform layer which abstracts the underlying complexity of architectural components. Typically, the platform layer exists at the system level, where structural components are available in terms of processing elements, bus-based communication hierarchies, and memory components. These components are already available as a physical implementation on the platform, as indicated in Figure 2.3. The system design starts at the structural axis of the Y-chart, reducing the designer's flexibility in terms of system-level processor synthesis, since the behavioural models need to be mapped to existing structural components.

To mitigate the loss of flexibility in this approach, most MPSoC platforms also contain custom logic cells which allow the designer to choose between software and hardware implementations of mapped behaviours. As we can see in Figure 2.3, the available custom logic cells still allow the designer to perform RT-level synthesis and map them on to logic components which can then be implemented on the platform.

Platform-based design, as described by Sangiovanni-Vincentelli and Martin [71], defines platform layers at different levels of abstraction and describes the degrees of design exploration freedom resulting in these abstraction layers. As they state, an *architecture platform* specifies processor com-

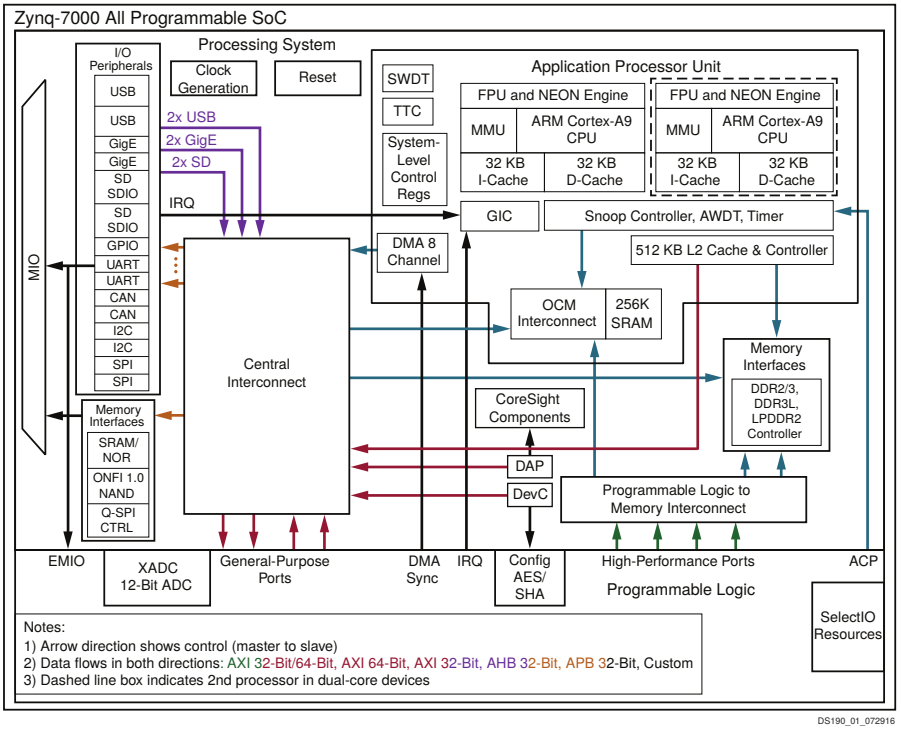


Figure 2.4: Overview of the Xilinx Zynq 7000 series MPSoC architecture ([82]).

ponents and interconnect schemes (e. g. ARM Cortex processing elements and the AMBA[1] bus technology). By choosing an architecture platform, the designer constraints the design space to this family of architectural components and further design space exploration considers the parameters of each component. The *Application Program Interface (API) platform* represents another level of abstraction consisting of a software interface which abstracts the underlying hardware capabilities, the Real-Time Operating System (RTOS) services, and the communication mechanisms. As such, the API platform enables re-use of software components across different operating systems and hardware implementations.

2.3 MPSOC PLATFORM CHALLENGES

The platforms in today's embedded system design usually consist of multiple processing elements attached to a common, shared bus-based communication infrastructure. They furthermore provide additional dedicated Field-Programmable Gate Array (FPGA) cells for implementing custom hardware logic. These MPSoC platforms are the result of a shift towards a platform-based design approach and provide higher flexibility when targeting embedded systems due to their high-performance application processing units and the presence of dedicated hardware logic cells. At the same time, they support a variety of different application domains due to their flexibility. Figure 2.4 shows an example of such an architecture. As we have discussed in the previous section, the platform-based design methodology

1 Advanced Microcontroller Bus Architecture

is a trade-off between the designer's flexibility and the ability of platform component reuse.

Platform-based design has particularly emerged due to the ongoing shift towards a software-centric embedded systems design [25]. As a result of this shift, processing elements on MPSoC platforms are becoming more powerful to help implement the variety of use-cases on these platforms. On the one hand, this rising processor complexity manifests in their support for parallel execution, on the other hand, implicit resource sharing can further increase the single-thread execution performance. Here, we briefly discuss the challenges which emerge when targeting real-time sensitive software components on such MPSoC platforms.

In the context of MPSoCs, we can distinguish between *multi-processor* and *multi-core* systems. The former denotes a system consisting of multiple processing elements which share a common interconnect infrastructure. From the programmer's perspective, a multi-processor system performs Asymmetric Multi-Processing (AMP). The available processors are not explicitly synchronised and – with possibly dedicated cache hierarchies – can have different views on the state of the memory subsystem. As an example, such a multi-processor system may consist of a powerful, hard-wired application processing unit and a set of soft-core processors implemented in the FPGA part of the MPSoC.

The term *multi-core* on the other hand refers to processing elements which contain more than one execution unit. All cores connect to the same cache hierarchy, enabling configurations such as Symmetric Multi-Processing (SMP), where special hardware support provides mechanisms to ensure a coherent view on the memory subsystem by propagating cache state changes to all attached cores.

Both AMP and SMP configurations are challenging in the context of estimating the temporal behaviour of mapped software components. In general, the parallel execution of software on multiple processing units, regardless of their AMP or SMP configuration, leads to contention on shared resources (e. g. memory or peripheral access) which can delay task execution and thus invalidate any Worst-Case Execution Time (WCET) assumptions about the software task running on one of the processing units. In the case of SMP systems, the contention can also occur on the shared cache hierarchy, which increases the possible interference even more. Any contention may not only delay execution, but can also cause a speed up, since a parallel executing task can prepare cache lines for the other task running on the synchronised cache unit.

These contention scenarios need to be taken into account when considering execution times of mapped software tasks. As a consequence, the main consideration when targeting such MPSoCs for embedded real-time systems is how to handle the introduced uncertainty regarding the temporal behaviour of software mapped on the processing elements. Especially in the case of SMP configurations, the parallel execution of software can drastically alter the temporal behaviour due to its interference on the shared cache.

These facts result in a limited use of SMP and AMP platforms in the context of safety-critical systems which typically require the definition of a strict upper bound on the execution time of their functions. Depending on the

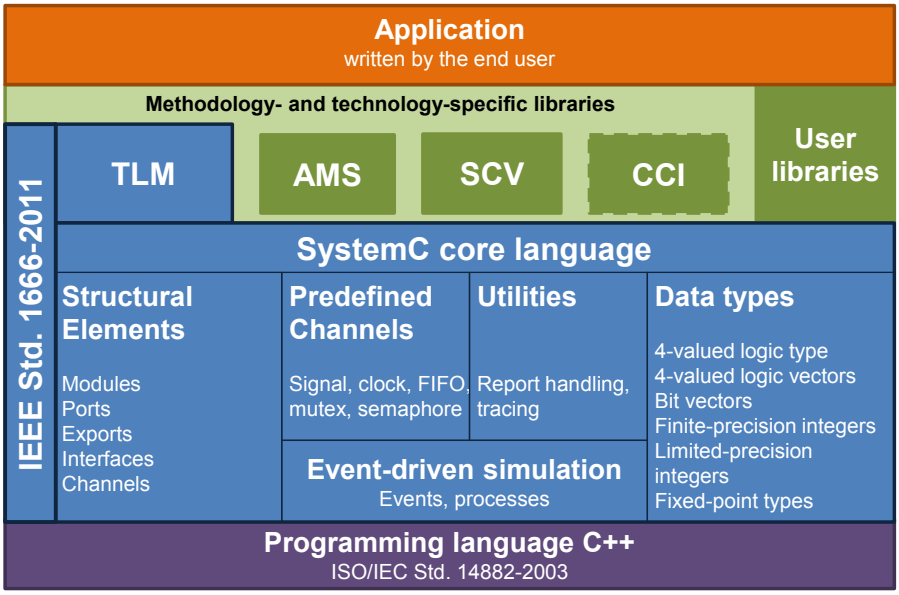


Figure 2.5: Overview of the SystemC library structure and its components ([41]).

performance-increasing features of a platform regarding implicitly shared resources (pipelines, branch prediction, caches, ...) and their impact on timing contention effects, they have to be controlled or configured such that their impact on the timing behaviour is still manageable in the context of execution time estimation [18].

2.4 SYSTEM-LEVEL DESIGN WITH SYSTEMC

The previous sections have shown that a crucial part in the design of embedded systems is the ability to specify and model the system components. The following section provides an overview of SystemC, a System-Level Design Language (SLDL) for the modelling, specification, and refinement of HW/SW components. This section further demonstrates how SystemC provides different modelling granularities and details of computational and communication refinement.

2.4.1 *Structural & Behavioural Modelling*

SystemC provides system-level modelling components and an event-driven simulation kernel implemented as a C++ library. Figure 2.5 provides an overview of the library components. The library contains structural and behavioural extensions for C++ to support the modelling and simulation of complex HW/SW systems. It has been standardised as IEEE 1666-2011 [41] and is widely used in today's industrial and academic areas for prototyping, modelling, validating, refining, and synthesising complex HW/SW system descriptions. Although technically provided as a C++ library, SystemC is equipped with quasi-language constructs that build upon the powerful abstractions and generic programming techniques provided by the language.

To model the structure of systems, SystemC provides *modules* that can be nested and combined to provide a hierarchical view of the system. They can further be connected via *ports* which model communication interfaces. Such ports are used to access channel behaviour outside of a module via a well-defined interface. Ports can be structurally bound to matching *channels*, i. e. components which implement the interface defined by the port. Ports, interfaces, and channels therefore are the core concepts of SystemC for allowing structural system modelling and separating communication and computation. Additionally, channels can range from simple signals (*primitive channels*) that model hardware wires to complex stateful communication channels such as FIFO buffers or transactors (*hierarchical channels*).

Each SystemC module may contain multiple processes which model (concurrent) behaviour. Processes can be sensitive on certain events generated from other processes. Events are either directly sent or indirectly triggered, e. g.through a value change on a signal to which a process is sensitive. When such an event occurs, the process is woken up and executes according to its type. SystemC supports three different process types:

METHODS provide processes that model asynchronous behaviour and are executed in a run-to-completion manner.

THREADS are suitable for modelling behaviour across multiple simulation cycles. In fact, they are usually implemented with an endless loop which, after waking up and processing the events, will sleep again until the next event arrives.

CLOCKED THREADS are processes based on the thread behaviour which additionally provide a clock and reset interface to easily describe synchronous hardware processes.

The SystemC simulation kernel provides an *event-driven simulation* with the concept of an evaluate-update phase and delta cycles to simulate concurrent behaviour. Due to the semantics of an event driven simulation, the simulation strictly operates on a global event queue and simulation time is always advanced to the timestamp of the next event. Once all events have been processed for the current time stamp, the simulation time is advanced to the next event in the queue. The simulation finishes when all events have been processed and the queue is empty. Each behavioural process has a *sensitivity list* which defines if the process is invoked when a certain event occurs. Unless explicitly stated, at the start of the simulation, processes are executed once to generate initial simulation events.

2.4.2 *System-Level Modelling*

As can be seen, each event forces to SystemC kernel to schedule and execute the underlying process that is sensitive to the event. Therefore, a detailed simulation which relies on fine-grained events, which is typically the case for pin- or cycle-accurate hardware models, can significantly reduce the simulation speed. Compared to other Hardware Description Language (HDL) simulators, the slowdown of simulation performance is amplified since the

input models cannot be efficiently compiled and optimised with the structural information about the models, a technical consequence of the simulation kernel being integrated into the SystemC library.

SystemC is therefore not suitable for modelling cycle-accurate hardware designs. Instead, it operates on a higher level of abstraction. The SystemC Transaction-Level Modelling (TLM) library (TLM 2.0, which is included in the SystemC releases since 2.3.0) provides high-level components for modelling Memory-Mapped I/O (MMIO) based HW/SW systems by extending the SystemC data type library with components for MMIO *initiators* such as processing elements, or direct memory access (DMA)-capable peripherals, and *targets*, which represent bus peripherals or memory models. TLM 2.0 extends the existing SystemC Register-Transfer Level (RTL) modelling techniques with a new level which abstracts pin-accurate communication interfaces with *interface method calls* between initiators and targets. Instead of pin-level interface descriptions through signals and ports, a TLM peripheral contains callbacks for bus-based MMIO access to registers.

As a consequence, the communication behaviour with TLM 2.0 is not described in terms of hardware wires, but can be seen from a software perspective, thus raising the abstraction from hardware-based pin interfaces to a software modelling view. Although obviously reducing simulation accuracy by combining a wire-based communication protocol to a method call, this greatly reduces the number of events the simulation kernel has to process.

TLM 2.0 further provides means to refine these transaction-based communication models. The designer can start at a high-level bus-functional model containing synchronous communication operations and temporal decoupling techniques with reduced timing accuracy towards a bus-cycle accurate model, where asynchronous communication operations and cycle-accurate cost models can be integrated. This technique enables the simulation of complex HW/SW MPSoCs platforms with a virtual prototype, an essential component in today's system modelling and development processes. Summarising, SystemC enables to provide *virtual prototypes* of complete MPSoCs platforms and is nowadays mainly used for system-level modelling and refinement, whereas other hardware description languages, such as VHDL or Verilog are used for RTL modelling.

2.5 REFINEMENT & IMPLEMENTATION

The Y-chart has shown that the implementation of a system is realised by starting with a specification model and then refining it towards either the implementation (in case of a top-down approach) or the platform (in case of a platform-based approach) on which it is implemented. Performing these design decisions on a flexible abstract model leads to a refined model containing more information and less flexibility about the final implementation. Each abstraction layer in the Y-chart contains models with various degrees of complexity and expressiveness regarding the functional and extra-functional properties of the system. This section describes how the SLDL SystemC provides tools for implementing these models at each step of the refinement flow.

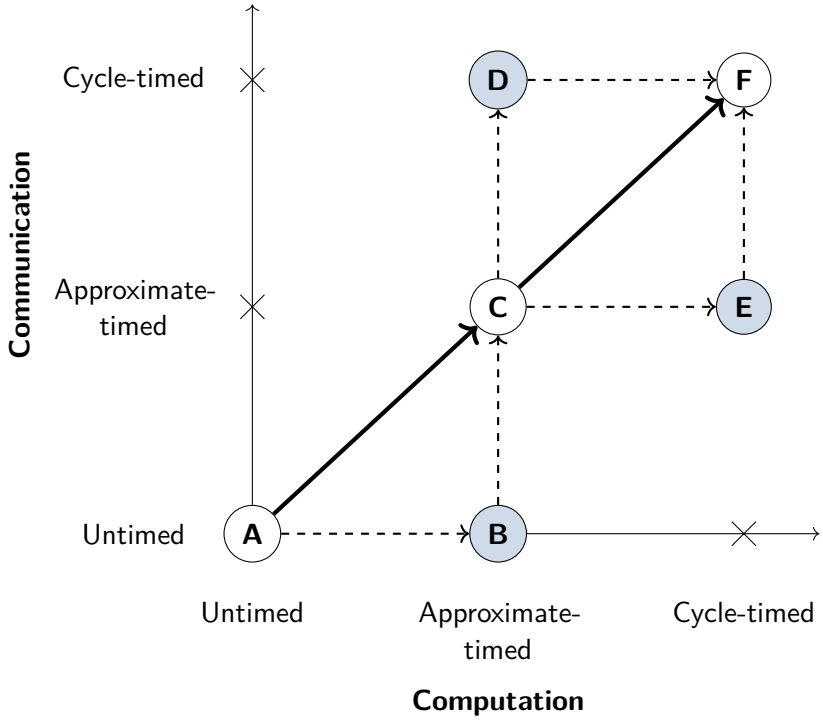


Figure 2.6: Model Granularities (from [26]).

2.5.1 *Modelling Layers*

Gajski et al. [26] categorise modelling granularities according to the complexity of their timing annotations, as depicted in Figure 2.6. Additionally, due to the separation of communication and computation in the system-level specification models, it is possible to consider communication and computation refinement independently.

Starting with the specification of the system, the model usually consists of an untimed, pure-functional representation of the desired system behaviour (A) which is called the *specification model*. This model provides a system-level behavioural view and it can be used to gather insights regarding the functional correctness without considering any timing or other extra-functional constraints.

The first refinement step then transforms the specification model to a timed-functional model (B). The model is refined according to its computation and initial timing annotations are attached, either through the use of execution models or measured results. Refining the communication of the model then results in a transaction-level model (C), where communication processes are modelled using time-accurate bus transactions.

Depending on the design flow, either the communication or computation is now further refined towards a cycle-accurate model. The result is either a bus-cycle accurate model (D) with pin-accurate communication behaviour or a computation-cycle accurate model (E) of the system where computation is described using cycle-accurate timing behaviour. Finally, combining computation and communication models results in a full cycle-accurate model (F).

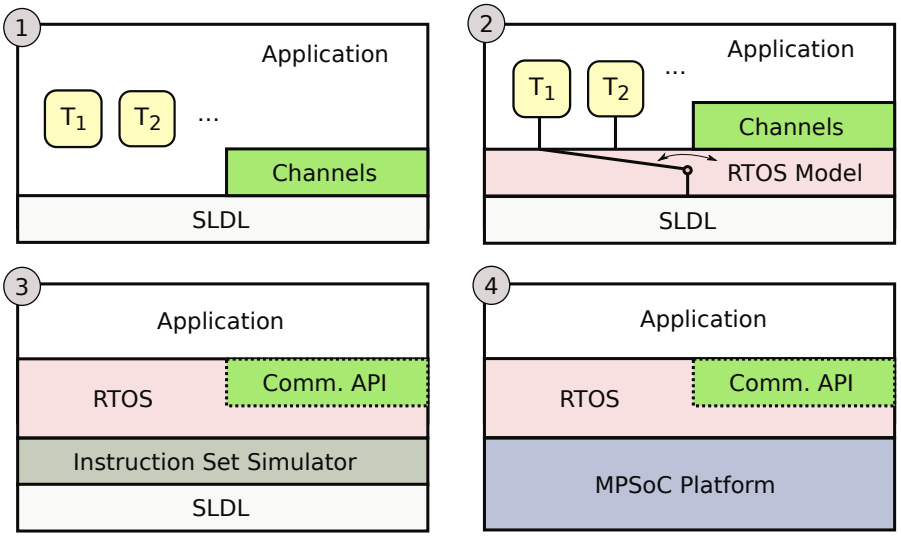


Figure 2.7: Illustration of the abstraction layers in computational refinement. Starting at the specification-level (left), the compuation model is incrementally refined and new layers are added until the refined software layer architecture can be implemented on the platform (from [26]).

Generally, a fully refined cycle-accurate model of a complete system based on today's MPSoC platforms is infeasible to analyse due to the rising complexity of these systems and the resulting number of artefacts in the simulation models. Mixing different abstraction levels for describing communication and computation components can therefore be useful. This allows the functional behaviour of software components to be modelled early on in the design process, prior to implementation and cross-compilation for a certain target processor.

2.5.2 *Computational Refinement*

Computational refinement, the focus of the contributions in this thesis, considers processor modelling as a means for specifying and modelling computational entities. Modelling and refinement of computation covers multiple stages, as indicated by the Y-chart in the previous section. At the starting point for system-level computational modelling and refinement, the functional behaviour is represented as (untimed) processes and typically modelled using a high-level concurrent computation model. These behavioural models can either be natively supported by a SLDL (e. g. Program State Machines in SpecC [28]) or implemented on top of the available simulation semantics (e. g. SystemC [35]).

The mapping of high-level execution processes in the application layer to execution units (e. g.tasks) on the operating system level can vary. Generally, such processes are partitioned into behavioural units which are then executed sequentially within an operating system task. If multiple tasks are mapped to the same resource, a *multi-tasking* environment needs to be provided on the processor and a runtime has to manage the sequential exe-

cution of the mapped tasks. These scheduling policies can be categorised by their flexibility at run-time.

Choosing the suitable policy depends on the temporal requirements of the application-level processes as well as the platform capabilities and the constraints on the analysis complexity of the system. Generally, tasks can either be executed *statically* according to a schedule defined at design-time, or *dynamically*, where decisions are performed at run-time depending on the scheduling policy implementation and dynamic properties of the system. In general, tasks are scheduled according to their *priority*, which is chosen by the scheduler and based on application-level requirements such as response time or criticality. If the priority of a task does not change at run-time, the scheduling policy is called *fixed priority*. An example is the *deadline-monotonic* scheduling policy introduced by Liu and Layland [56], which monotonically assigns task priorities according to their period length at design-time and executes them on the processor in the order of their priority.

A scheduling policy which assigns priorities at run-time can be further divided into the categories *job-level fixed priority* and *dynamic priority*. While task-level fixed-priority policies statically assign a priority value to each process at design time, a *job-level fixed priority* policy is able to assign priorities individually for each task instance (job) at run-time. This means that different job instances of the same task can have different priorities, depending on the dynamic state of the system. A prominent example of a *job-level fixed-priority* scheduling policy is Earliest Deadline First (EDF) [56]. Upon a job arrival, the scheduler determines the next deadline of all active jobs and assigns the highest priority to it.

The job-level fixed priority scheduling policy re-evaluates the run-time properties each time a new task instance is ready for execution. The third category, *dynamic priority*, removes this restriction entirely, and thus allows the scheduling policy to re-assign priorities even for active jobs. These scheduling policies have been extensively studied according to their analysis capabilities [44].

Figure 2.7 illustrates four steps of software refinement, starting at the system-level down to an implementation based on a RTOS. After specifying the behavioural processes, the next step of model-based software refinement requires a RTOS model to ensure that tasks are executed in sequential order according to the chosen scheduling policy and that only one task is active at any time. The model also considers scheduling events and task preemption, such that it is possible to gain accurate results from scheduling artefacts while performing a simulation at near-native execution speed.

Since the application-level task execution is still being performed on the simulation host, any timing or otherwise platform-dependent behaviour needs to be annotated to the models. In order to achieve this, the RTOS model provides an abstraction layer around the SLDL primitives and interfaces for tasks to consume time. Furthermore, tasks use the underlying RTOS API for inter-process communication. The RTOS model is therefore able to simulate task preemption with an accuracy given by the annotated execution time. Therefore, the modelling detail directly depends on the granularity of the annotated execution time, both of the tasks and the RTOS. The model there-

fore can serve as a first indicator of congestion or locality issues and provide results for scheduling policy design decisions.

The RTOS model can be refined by adding a Hardware Abstraction Layer (HAL) which models the HW/SW interface and connections used for communicating with hardware peripherals, such as interrupt generators or memory. It is also used to implement the virtual channel-based communication on the processor interconnect. While the RTOS already provides drivers for accessing bus-based communication, the HAL implements the session management of the driver requests to the single bus connection. The HAL is also responsible for mapping the external interrupts to appropriate user-defined interrupt handlers. The next level, the hardware layer describes the connection of a pin-accurate protocol of the underlying hardware.

Finally, the application is cross-compiled on a Instruction-Set Simulator (ISS). Such a processor model features the final target memory layout as well as the desired instruction set architecture. An operating system executed on such an ISS then implements the modelled timing and interrupt handling behaviour and the communication via the external ports. The result is a refined computational model which can be deployed on the target platform.

2.5.3 *Communication Refinement*

Communication at the application layer is typically modelled using virtual channels and shared variables. Its refinement is tightly coupled to the ISO/OSI 7-layer abstraction [26]. This section gives a brief overview of the steps. In the first refinement phase, channels and variables are enriched with data types and a data layout to prepare their mapping to the underlying (untyped) data stream and communicating processing elements. The refinement step includes determining (and possibly converting) bit width, endianess, size, alignment, and other data-dependant properties. In the next step, the virtual channels are mapped to communication sessions on physical channel implementations, possibly merging multiple channels originating from the same processing element.

The next layers, the network layer, transport, and link layer are responsible for implementing the network infrastructure derived from the virtual channel hierarchy. Here, channels are combined into physical representations, and bridges or transducers are inserted to ensure correct message routing across different physical protocols. Moreover, the link layer ensures a synchronised message transfer for the virtual peer-to-peer channels by implementing polling techniques or dedicated interrupt lines. The final layers, stream, media access, and physical layer model the data streaming process, the abstraction of the underlying medium, and the bus protocol and arbitration models.

2.6 VALIDATION & VERIFICATION

An essential part in the design of embedded systems is the process of validation and verification during the refinement of the system components. Its purpose is to ensure that models still behave under the assumptions and

deliver the specified functionality when they are transformed into more detailed representations along the refinement process.

The processes of *validation* and *verification* differ in their expressiveness, level of assurance, and their capabilities regarding the input model complexity. The term *validation* refers to the experimental assurance that under some given input parameters, the refined system behaves as expected, i. e.provides equivalent output parameters. Validating a model therefore starts with defining the expected output trace of actions over time by the system, storing it along with the input parameters that trigger the system behaviour and finally compare them against the observable behaviour of the refined model.

The challenge in validating a system therefore lies in choosing the set of input parameters that accurately represents the overall input parameter space. Since the chosen input values may not fully cover the input parameter set due to their complexity, the validation step may not provide sufficient certainty that the refined system behaves as specified. As a consequence, the validation process cannot cover all possible states of the system and the designer often needs to provide expert knowledge for achieving the necessary *coverage* of the validation tests. Depending on the design assurance level of the component (i. e. derived by the Safety Integrity Levels (SILs) which will be detailed Chapter 3), the process of *validation* might not provide sufficient results.

The *verification* process on the other hand describes methods of formally proving assumptions about the models. They provide an exhaustive way of searching through the model state space and can therefore prove that certain assumptions will never be violated (*safety* properties), or that certain states will eventually be reached (*liveness* properties). Informally, safety properties therefore ensure that "something bad never happens", while liveness properties ensure that "something good will eventually happen" [48]. While models with a higher level of abstraction can represent reasonable complex systems, the challenge lies in formally proving that the refinement steps of the model do not invalidate the assumptions of the model properties on higher levels of abstraction. This section provides an overview of formal *verification* methods and simulative *validation* approaches.

2.6.1 *Formal Verification*

Formal verification is extensively used in safety-critical domains such as aerospace, where the certification of components adhere to the highest standards in the industry to provide the necessary level of assurance regarding their specified behaviour. To achieve this level of certainty, the design of components in the aerospace domain requires formally verifiable models for specifying the system and a correct-by-construction mechanism to prove that the satisfaction of properties regarding timing or functionality as specified still hold in their implementation. Since formal approaches essentially check the state space of a behavioural model against safety invariants and liveness properties, the input models for specifying such systems are care-

fully designed such that the overall expressiveness is constrained to avoid state space explosion.

In the case of integrating multiple different functionalities on the same platform, care must be taken to restrict their interferences. Otherwise, formal approaches may quickly face the state space explosion problem when different subsystem behaviours are integrated. Consider two application models A_0 and A_1 which are individually verified against some input specification consisting of safety and liveness properties. Let $|A_i|$ denote the state space represented by the application model A_i. Then, the overall state space which needs to be considered for the individual verification is $|A_0 + A_1|$, since both verification checks are performed separately. After integrating the applications on the same platform however, the final integrated model needs to be verified as well. The resulting integrated system $A_0 \bigoplus A_1$ is *composed* of the individual components by some mechanism $\bigoplus$. If there is insufficient isolation between the applications, they may interfere with each other regarding their timing properties and their functional behaviour. Instead of checking $|A_0 + A_1|$, the overall state space may increase up to $|A_0 \times A_1|$ which can render a formal verification infeasible.

Several techniques for segregating these systems and achieving composability have been proposed throughout the literature. As an example, the Integrated Modular Avionics (IMA) approach provides an industry-proven technique which guarantees that the individually verified applications do not interfere spatially and temporally, and therefore cause a state space explosion. A prominent specification language for formal models regarding timing requirements in the domain of embedded real-time systems are timed automata, introduced by Alur and Dill [2], which can be verified against specification invariants using the Uppaal [9] model checker. The model checker can prove safety properties through the notion of invariants and provides a temporal logic for specifying liveness properties.

2.6.2 *Simulation-based Methods*

Simulation-based approaches generally consist of two components: a testbench and a Design under Test (DUT), as illustrated in Figure 2.8. The input and output ports of the DUT are attached to the testbench, which consists of input stimuli and output monitors. The testbench then provides input stimuli over time (*traces*) to the DUT and reads the output signals. Validating the DUT is performed by checking the output against an expected trace output.

Simulation-based methods are categorised depending on the *invasiveness* of the input and output ports of the testbench. Stimulating and observing the observable behaviour on the ports of a DUT is referred to as *black-box testing* due to the fact that no internal design knowledge other than the externally visible behaviour is used for testing. In contrast, *white-box testing* refers to a more sophisticated approach where observers and stimuli generators are attached to internal ports of the design which are hidden from external components.

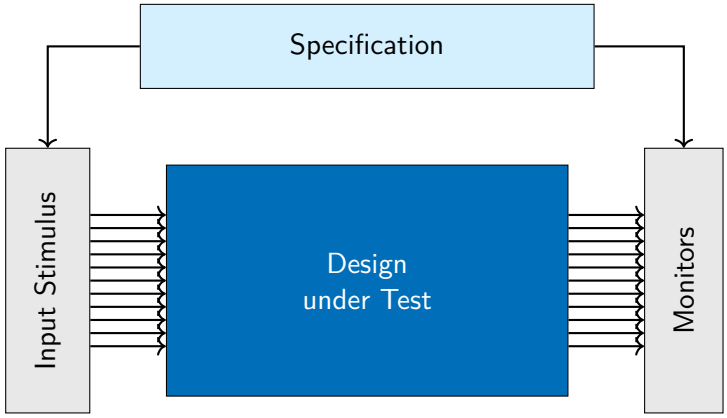


Figure 2.8: Simulation-based validation of a Design under Test based on a specification which states the input stimuli and the expected output trace captured by monitors [26].

The overall validation result highly depends on the chosen input parameter set and the *coverage* of the input traces regarding the expected workloads. To solve this issue, the designer first has to identify and define the workload characteristics of the system before specifying a subset of input traces which sufficiently represent their state space. Since an exhaustive test of all input parameters can quickly become infeasible, several techniques have been proposed to increase the overall validation performance [26].

White-box testing exploits expert knowledge of the internal behaviour of the DUT and can perform checks not only on the visible output parameters but also on its internal state. Depending on the complexity of the initialization phase of a system, this can greatly improve the validation performance. With an additional pre-selection of input test data, the validation methods can be used to force corner-cases and validate the desired behaviour.

2.6.3 *Performance Estimation*

The term *performance estimation* refers to the method of determining the execution effects regarding application-specific performance metrics when the behaviour is executed on a certain platform configuration. The notion of *performance* refers to the overall evaluation result of an application regarding metrics such as bandwidth, frames per second, throughput, etc.

Since these metrics depend on the platform characteristics on which the application is executed, a *performance model* needs to take into account not only the behaviour of the application, but also the platform configuration on which the application is executed. Therefore, the more detailed a platform model is, the higher the accuracy for predicting performance metrics can be. Performance estimation is an essential part in evaluating platform and application design decisions, because it provides early feedback on the resulting behaviour regarding essential application-specific requirements.

Simulation methods can be an effective way to gain initial performance metrics for evaluating the system behaviour. These metrics affect the design

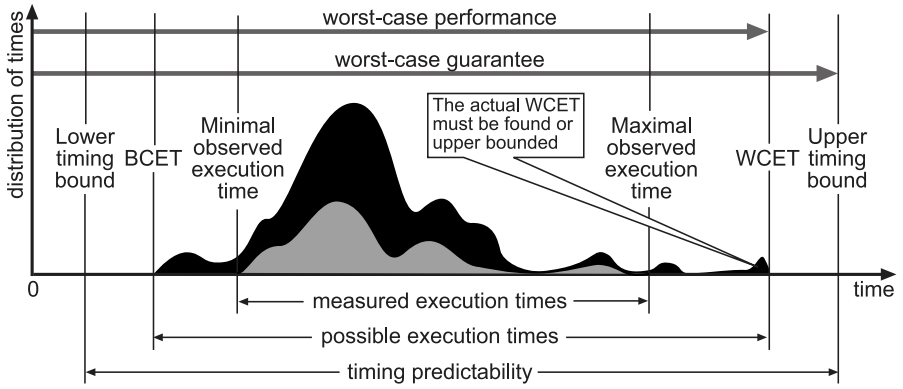


Figure 2.9: Comparison of execution time estimation using simulation- and analysis-based approaches [80].

decision in the design flow and as such, they are integrated in the flow in the form of feedback loops. Usually, these evaluation methods require models from existing low-level behavioural, structural, and geometrical descriptions. They are constructed using structural models, expert knowledge, or they can be generated by a pure measurement-based, data-driven approach.

Evaluating software performance behaviour requires constructing a timing model of the underlying processor. Depending on the desired granularity of the evaluation results, there are different methods for combining the software behaviour with the timing model. The abstract method for evaluating timing behaviour usually consists of timing back-annotations to the software control flow. Here, control flow blocks are annotated with their estimated temporal consumption on the target. Executing the behaviour then triggers the underlying timing model and results in an overall timing estimation of the software component. A more detailed timing model consists of an instruction-accurate simulation, where the software behaviour is cross-compiled for the chosen target and the instructions are simulated on an ISS which contains a timing model of the underlying target instructions.

Analytical performance estimation involves constructing a model of the software through static code analysis by evaluating the control-flow behaviour of the software at design-time without simulating or executing it. Static code analysis obviously has limitations regarding the input behaviour complexity and therefore often imposes restrictions on its expressiveness. But even if the behaviour can be analysed completely, the overall temporal behaviour still depends on the platform timing properties.

In today's MPSoC platforms, the actual temporal behaviour of software components depends on the current state of the processor and other components on the platform which might interfere its execution. With branch prediction, pipelining, and cache hierarchies, these processors are too complex for being considered in static analysis methods [80]. As we can see in Figure 2.9, due to the platform complexity and the complexity of the control flow behaviour, all observable execution times in a measurement-based approach only form a subset of the possible execution time behaviour. The obvious limitation of the measurement-based approach is due to the fact

that a set of input parameters has to be defined which limits the observed behaviour, indicating reduced *coverage*.

A complex processor micro-architecture contains many components which are used to optimise the instruction execution speed. In a von Neumann computer architecture, the main bottleneck is the slow access to the memory subsystem. Therefore, processors aggressively optimise access to the memory by caching, pre-fetching, speculative execution, and branch prediction. In all of these cases, the processor will read memory which is likely to be required in future execution paths, such that it does not have to wait for the memory subsystem if the execution finally happens.

However, these mechanisms have great implications on the timing predictability of the processor. The optimisation components of the processor are based on maintaining a global context to optimise future execution behaviour. As an example, in multi-core systems, the cache infrastructure is shared among individual cores. Since access time to the memory depends on the state of the cache (which determines whether the main memory needs to be accessed or the requested memory location already resides in the cache), analysis methods need to consider the cache state at each possible step in the execution. Furthermore, software executed on other cores can modify the shared cache state at any time. The result is a state space explosion which cannot be handled by contemporary analysis methods. The implications of these performance optimisations regarding analysis methods can be seen in Figure 2.9. Measurements can only identify a subset of the temporal behaviour due to their coverage issues, and since analysis methods are unable to iterate through the whole state space, static analysis results in pessimistic over-estimations, since the actual WCET cannot be determined.

The actual temporal behaviour of the software component forms a distribution over time, since it depends on the application control flow and data dependency as well as the execution context of the components on the platform. Thus, assuming one can trigger all possible control paths with the input data and test them under all possible hardware states, the actual distribution of the task execution time can be revealed. If the task is executed on top of a multi-tasking RTOS and mapped along other tasks to the same processing element, the timing analysis further needs to consider all side effects from other tasks which might have introduced temporal interference while running in parallel, including their functional behaviour impacting access to caches, bus, peripherals, and so on.

The only option to handle this interference is to assume the worst case in terms of timing properties when a task is accessing the resource. However, especially in the context of multi-processor platforms, this assumption is a heavy penalty regarding the estimated resource consumption. A complex platform therefore might be unable to guarantee timing requirements, while it is possible to implement them in simpler platforms. We can therefore conclude that for a sufficiently complex MPSoC, it is impossible to perform a static analysis of the executed software components and provide tight bounds on their temporal behaviour [80]. This shows that determining the WCET on complex MPSoC platforms is a challenging task and generally results in a huge over-estimation to provide a certain level of assurance.

The focus of performance estimation however is to provide accurate *average-case* temporal behaviour instead of determining the *worst-case* behaviour. Platform models have therefore been proposed which abstract their complex behaviour. These approaches can be categorised into static and dynamic profiling analysis techniques. In the static profiling analysis, the idea is to analyse the program control flow, determine the execution frequencies for each path that can be taken, and combine this information with knowledge of the platform execution time behaviour.

The dynamic profiling approach collects execution frequencies using targeted experiments with a chosen set of input parameters. Profiling the execution behaviour then yields the path execution pattern which serves as an input for the run-time characteristics of the task model. These models provide an abstract view of the task's control flow graph and are combined in a second step with platform-dependant information to yield timing estimates.

However, although these approaches provide a straight-forward approach of constructing task models due to their profiling phase, they are limited regarding the representation of the target task behaviour. The underlying assumption is that it is possible to reconstruct the target control flow graph from the native graph available in a *host-based* execution. This assumption does not hold in sophisticated target platforms such as Complex Instruction Set Computer (CISC) architectures. Since the profiling phase operates on the native control flow, the relation between native and cross-compiled target control flow needs to be preserved, which is usually only achieved by disabling compiler optimisations. However, especially in an embedded context, where the program size is constrained, this might limit the applicability of the approaches. Generally, resulting task behaviour models can be combined with relatively simple processor models (e. g.instruction cost tables) to gain initial insights into the target behaviour.

Apart from the control flow profiling, another option is to directly measure the application timing behaviour on the chosen hardware component. The main drawback of this approach is the limited availability of target hardware components for timing observability, the cost for integrating the software components on the target, and the need for a suitable measurement infrastructure.

2.6.4 *Workload Modelling*

The term *workload modelling* refers to models of applications and their typical workload behaviour on a target. Workload models mimic the target behaviour in terms of their platform component usage (e. g. processor, memory, interconnect, peripherals, ...). The benchmark suites such as Dhrystone and Whetstone [19, 79] were among the first to offer generic workload models for extracting performance characteristics regarding integer arithmetic or floating-point operations. These benchmark suites however only partially fit the workload characteristics of complex embedded software and therefore are only of limited use when estimating design tradeoffs.

To gain more accuracy and simulation speed, methods were proposed to derive synthetic workload models from the concrete to-be deployed embedded software. One way is to synthesise benchmarks from original workload use cases, as proposed by Noonburg and Shen [61]. One of the main motivations of this work is to provide a similar workload profile while at the same time reduce the evaluation effort of analysing the processor models by shortening the simulation time of the workload profiles. When parametrised according to given performance metrics, such workload models can provide an invaluable tool for platform and integration engineers to evaluate their design decisions.

Besides statistical approaches [22, 62], workload models can also be constructed using generative approaches where the target workload is measured and compared to an input workload. Each iteration then refines the workload to more accurately match the input workload. Another method is to use a data-driven workload modelling approach where the model is constructed based on the measured data. Such a synthetic workload construction appeared as a mechanism for mimicking software performance characteristics and consists of closely evaluating the input program execution in a profiling step and then selecting suitable control flow patterns and inserting the gathered performance parameters in the generation step [39]. Some approaches create benchmarks which depend on the micro-architectural behaviour of the platform [10, 76] or use randomized parameters to approximate the input program behaviour [4]. These methods mainly differ in their choice of primitives representing certain workload aspects and their handling of micro-architectural artefacts.

A workload model which is executed on the platform can have several advantages in the design of complex MPSoC. First, the model only needs to be integrated once and can then be parametrised according to the desired application parameters. The results in terms of resource utilisation can further be directly measured on the platform. Next, the model provides feedback due to the possibility of a preliminary integration of different subsystems (with their workload models). Finally, workload models allow distribution to external entities without disclosing any internal intellectual property. This can improve communication between different working groups and provides a technique for separating the subsystem implementation from the integration process.

Advanced workload generation methods include micro-architectural artefacts in their characterization process [4, 27]. These approaches thus allow the generation of benchmarks mimicking the spatial and temporal memory access behaviour on the target architecture beyond the CPU boundary and consequently include the memory hierarchy properties of the platform. However, in the case of mixed-criticality integration, this modelling complexity is not required, since the interference of memory access patterns is already mitigated on the platform-level. Therefore, in the context of mixed-criticality itegration, workload models are especially interesting in terms of accurately representing the application's timing behaviour.

2.7 OLDENBURG SYSTEM SYNTHESIS SUBSET

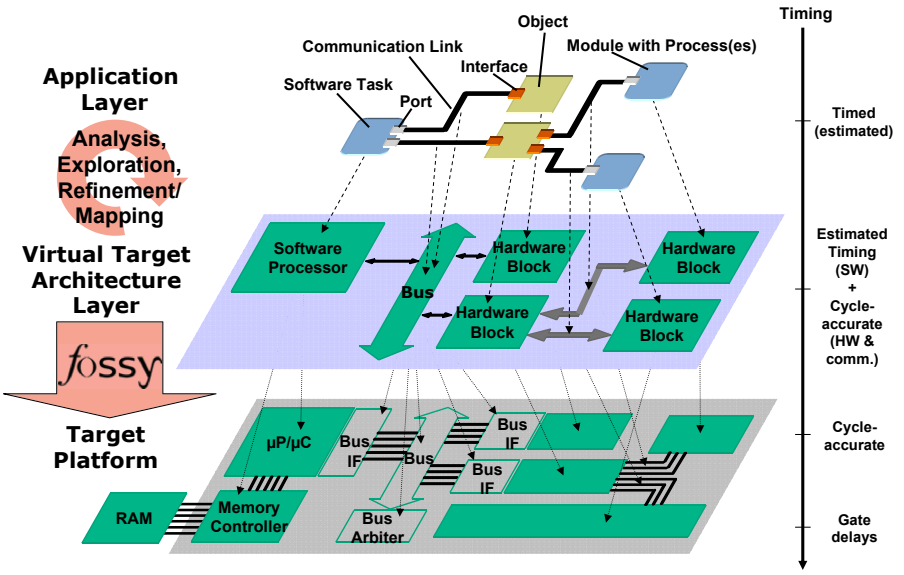


Figure 2.10: Overview of the Oldenburg System Synthesis Subset Design Methodology ([34]).

The Oldenburg System Synthesis Subset (OSSS) ([34]) consists of a design methodology and a synthesis subset based on SystemC. It provides modelling components for specifying, simulating, and synthesising complex HW/SW systems. OSSS thereby strictly follows the described platform-based design methodology. The proposed flow is depicted in Figure 2.10. First, the entry model and specification is described using behaviours, which contain functionality extracted from a C++ golden model. Based on these behaviours, the system is partitioned into SystemC modules and OSSS Software Tasks. Communication and synchronisation is performed explicitly via Shared Objects, which are special, C++-based objects that contain necessary synchronisation operations to guarantee mutually exclusive access and provide guarded, method-based access to their internal state.

This description of Software Tasks, SystemC modules, and Shared Objects is specified in the OSSS Application Layer (AL). The AL supports untimed model execution of the specified functional behaviour. Due to the flexibility, it is also possible to quickly evaluate performance-related measures by annotating profiling results from the golden model to the AL model. This approximately-timed model can serve for validating the functional correctness and provide profiling results for different design decisions regarding function locality and mapping of functional behaviour to OSSS entities.

The next step in the implementation of a system using OSSS is the mapping of application-layer objects to the Virtual Target Architecture (VTA). In this abstraction layer, models for different HW/SW components exist to model the physical communication infrastructure (along with their target properties such as bit width, alignment) as well as computation resource in a cycle-accurate manner. Different models are available for evaluating final design and architectural decisions.

As a final step, the VTA model serves as an input to an automated high-level synthesis process implemented by the tool FOSSY, which generates the overall system architecture and the necessary files for third-party vendor tools, such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) source code and other architecture description files. Behaviours mapped to Software Tasks are cross-compiled and linked against the platform support library in order to be deployed on the chosen target processor.

OSSS aims to provide a holistic design experience for embedded HW/SW systems. Starting with the C++ golden model, the tool-assisted workflow allows for designing hardware components as well as embedded software running on a processor on the chosen target platform. It integrates with existing industrial solutions (e. g.Xilinx ISE[2] Design Suite) to refine AL components to hardware blocks and provides target runtime support for cross compilation of AL components for targeting software processors.

The design methodology provides a C++ golden model consisting of Software Tasks, Shared Objects, and ordinary SystemC modules which acts as the common design entry point for the complete system. The advantage of this approach is the model consistency of the design entry point. As a result, given the annotation of early timing measurements, the analysis and exploration of refinement and mapping decisions can already be performed on the AL. Due to the model consistency, it is then easy for the designer to update the mapping of AL components to the VTA and therefore explore different HW/SW mappings. After choosing the mapping, the high-level synthesis tool FOSSY and the integration with external tool providers allows for quickly deriving a prototype of the described system.

As has already been discussed, the recent development in the domain of embedded systems follows a clear shift towards integrating complex software-based functionality on an embedded system, especially in the realm of complex CPS due to their nature of combining real-time sensor data processing in the physical domain with high-performance communication and computation in the cyber domain. While OSSS provides a common entry point for HW/SW synthesis, it does not offer the possibility of specifying different criticality domains for the model components. As a result, there is no method of prioritising across criticality domains in the implementation of scheduling and arbitration policies.

Moreover, the analysis relies on the back-annotation of timing measurements which need to be performed without the interference of other components. In the case of OSSS, where a single processor serves as a target of a Software Task, this approach might be sufficient for a software timing estimation early in the design flow. However, as we have discussed earlier, today's complex embedded platforms consist of multiple processing elements where executed software is creating timing interferences on multiple implicitly shared computation and platform communication resources. A more detailed estimation of the timing behaviour is therefore required to incorporate the uncertainties resulting from shared resource usage. However, such timing uncertainties cannot be represented accurately with a single value, as

2 Integrated Synthesis Environment

currently supported by OSSS. While we have already discussed the issue of timing uncertainty when modelling software timing behaviour, the next section discusses the challenges arising with the developement of mixed-critical embedded systems and how their solutions can be integrated in an embedded systems design flow such as OSSS.

3 MIXED-CRITICAL EMBEDDED SYSTEMS

So far we have taken a look at methodologies and flows for designing embedded systems in general. In this section, we will take a closer look at *mixed-critical* systems, a research topic which has lately been in the focus of academic and industrial efforts.

3.1 INTRODUCTION

There currently exist different views on what constitutes a mixed-criticality system. The term *mixed-criticality* used in this thesis refers to combining functions which fulfils objectives that can be categorised into different criticalities, specified by a safety assessment. The purpose is to provide an estimate on the required reliability of system components in hazardous environments. These reliability requirements are typically estimated by assessing the overall environmental outcome of a failure in a component and are in general part of a domain-specific Failure Mode and Effect Analysis (FMEA).

The categories of such analyses can typically range from *low-critical* operations which perform functionality irresponsible for the overall safety of the system and its environment it operates in, to *high-critical* functions, which are required to operate correctly and whose faults can result in severe or even catastrophic failure. Therefore, a *mixed-criticality* system is a system consisting of components with different classifications according to their criticality. In the context of real-time requirements, the desired reliability stated by an FMEA is achieved when high-critical components are guaranteed to operate under their pessimistic resource estimations and therefore correctly execute under their worst-case timing assumptions.

Mixed-criticality systems are formed whenever more than one functionality is mapped to a system and the functionalities can be categorised by some criticality property. Depending on the definition of the system boundaries, it is thus not even necessary to map the functionality on the same System-on-a-Chip (SoC) for it being categorised as a mixed-criticality system. Methods to perform a *separation of concerns* for such systems already exist, such that components with different criticalities are separated in order not interfere with each other. An example of this is the individual packaging of multiple sensor process chains in the automotive sector.

Designing *mixed-criticality* systems is thus not a new issue when dealing with multiple devices in a system. However, as the previous sections have detailed, the emerging challenge is to guarantee *sufficient independence* for functions mapped to the same MPSoC due to implicitly shared resources on the platform, a result of the ever-increasing complexity and the desired feature density on MPSoCs. The novel aspect which gained interest in the academic community was the idea of considering these criticality properties

Table 3.1: Safety Integrity Level categorization and failure probabilities per hour [11] for a continuous mode of operation.

SIL	Category	Failure probability (per hour)
1	*minor*	$10^{-5} > P \geqslant 10^{-6}$
2	*major*	$10^{-6} > P \geqslant 10^{-7}$
3	*hazardous, severe error*	$10^{-7} > P \geqslant 10^{-8}$
4	*catastrophic*	$10^{-8} > P \geqslant 10^{-9}$

when specifying the system, in particular in the context of resource sharing scenarios.

These challenge of designing mixed-criticality systems within the context of complex MPSoC platforms has lead to two major research areas. The first research objective considers *analysing* the coupling effects of behaviour which arise when combining functionalities with different criticalities on a shared hardware platform. Mitigating the resulting interferences requires careful analysis of their platform usage behaviour and thorough evaluation of design techniques to isolate functions of different criticalities, both in the temporal sense through resource scheduling techniques and the spatial dimension through the use of proper access enforcement techniques. This research area is thus based on the fundamental challenge of designing safety-critical, real-time sensitive embedded systems.

With the advent of mixed-criticality in the realm of real-time embedded systems, system designers are now facing a multi-objective optimisation problem when developing safe and resource-sensitive embedded systems. A major aspect of mixed-criticality research is the consideration of resource usage optimisation. Complex MPSoC platforms provide many powerful computational resources which are implicitly shared between different processing elements or even software components running on the same processor. The fact that safety-critical timing analyses are by definition performed pessimistically, specifying the system schedules based on these analyses naturally results in an under-utilisation of its resources in the average-case usage scenarios. In the context of mixed-criticality, this constitutes a multi-objective optimisation problem since after guaranteeing resources for the safety-critical operation, the designer is interested in optimally utilising its resources which may be used by lower-critical application components.

We consider the analysis of methods and scheduling policies to reduce this *utilisation gap* as the challenge of *optimising* mixed-criticality resource consumption which requires successful interference mitigation techniques to be implemented on the platforms. The second objective in mixed-criticality research therefore considers the question of how to exploit criticality properties to increase the platform utilisation in mixed-criticality systems while still adhering to the safety requirements stated by the criticality properties of an FMEA.

The result of an FMEA provides failure probabilities which are then further categorised according to domain-specific knowledge. This classification of functions regarding their criticality is handled in many industry stand-

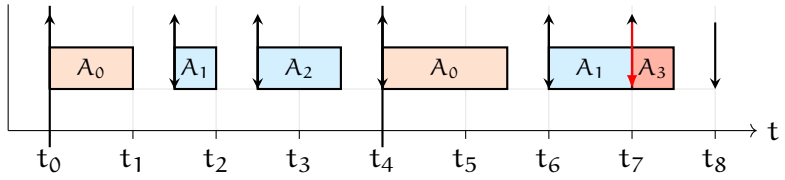


Figure 3.1: Example Time-Division Multiple Access schedule of five applications $A_0, \ldots, A_4$ on a common platform where each application is allocated a specific slot and an overall periodic execution behaviour.

ards, most derived directly or indirectly from the IEC 61508 [11], published by the *International Electrotechnical Commission*. The standard defines SILs to classify functions performed by electronic components. SILs are defined according to the consequences in case of a failure of such components, as listed in Table 3.1.

The classification and SIL determination of these components is performed by a domain-specific *risk analysis* in the context of an FMEA. The ISO 26262 [42] defines domain-specific Automotive Safety Integrity Level (ASIL) and provides risk analysis methods to determine the ASIL for each component. ASIL are closely related to the SIL described above. The aerospace domain provides the DO-178C [20] standard and defines Design Assurance Level (DAL) which are determined using similar risk analysis methods.

In the context of *mixed-criticality*, these classification levels are used to categorise functions mapped together on common platform resources. Such mixed-criticality systems adhere to different SIL at the same time and consequently need to implement proper segregation techniques. In order to integrate subsystems with different criticalities, the underlying platform needs to support *sufficient independence* [42] between such functions. This is the main consideration when integrating mixed-criticality on a common platform: if sufficient independence cannot be guaranteed, such combined systems need to be implemented according to the highest criticality level of all subcomponents, which often renders its development cost economically infeasible.

3.2 SEGREGATION APPROACHES

The safety standards require the design process to guarantee *sufficient independence* between mixed-criticality functionalities. But which isolation techniques and platforms satisfy this property? In the aerospace domain, the *ARINC 653* standard defines such a platform with Time-Division Multiple Access (TDMA) principles, where the system is temporally split into different execution partitions which mark an exclusively available slot to each mapped functionality.

Figure 3.1 depicts an illustration of this segregation technique. Each application $A_0, ..., A_4$ is given a fixed slot with start time and duration. To ease the transition between application-level requirements such as periodic execution behaviour, the overall schedule is often divided into periodic partitions which the applications can be allocated to. In this example, the first partition starts at t_0, the next at t_4, then at t_8 and so on.

Functionalities mapped to such a platform can be allocated time slices of these partitions and the underlying TDMA scheduler prevents erroneous overruns by preempting the system at the end of the defined application execution time. In the illustrated example, application A_1 executed at t_6 in the period starting at t_4 misses its deadline due to a fault in the internal behaviour. However, the preemptive scheduling based on application time-slices prevents the fault from propagating into the behaviour of application A_3, such that it can still operate as specified. Hence, TDMA prevents fault propagation across applications and can therefore provide *sufficient independence*, as requested by the industry standards.

In the aerospace domain, the IMA platform is aimed at providing a modular integration and execution environment for achieving flexibility regarding function composition and platform partitioning. It provides a modular platform which is able to execute functions of different criticalities due guaranteeing independence between applications. Such platforms therefore provide a solution in the context of the mixed-criticality challenges regarding platform-level segregation. Mixed-criticality systems can be integrated using these platforms, but at the loss of flexibility, since the static TDMA partitioning is unable to consider resource dynamics, especially considering performance-related, non-critical functions. Research has therefore focussed on utilisation aspects of mixed-criticality systems and provides novel approaches for the specification, scheduling, and enforcing segregation properties in mixed-critical embedded systems.

3.3 REAL-TIME MODELS & ANALYSIS

The real-time systems domain considers the issue of utilisation in mixed-criticality systems. The key motivation of mixed-criticality models is based on the observation that with the growing complexity of today's platforms, a static analysis of execution times results in very pessimistic, often never observed, upper timing bounds. This is because the actual WCET cannot be determined due to the complexity of the platform. As a consequence, while the design-time system scheduling and partitioning setups are configured such that this upper bound can be guaranteed, the overall system utilisation is reduced.

The idea of real-time mixed-criticality task models is to take into account the dynamic average- and worst-case behaviour of the executed functions and categorise them into different scenarios. Typically, a low-critical and a high-critical execution behaviour is defined which considers different run-time observed task execution time behaviours.

In general, the definition of an mixed-criticality task model τ consists of a *vector* of execution times $\vec{C}$, with an execution time for each defined criticality. This models the *expected execution time* on each criticality scenario level. Such real-time analysis models require the execution semantics of the underlying system to monitor the execution time and perform the scheduling decisions based on the observed task execution time. If a task exceeds its execution time budget, the system transitions into the higher criticality scenario. Otherwise, the schedule is performed as defined for the low-critical

scenario. Typically, in higher-critical scenarios, lower-critical functionality is either not considered at all or executed in a degraded mode, such that it is able to reduce its resource consumption and provide a feasible execution of the overall mixed-criticality system.

Different task *execution profiles* have been proposed by considering task execution time not just as a function of the underlying platform and the behaviour, but also by their *criticality scenario*. Typically, high-critical tasks require a thorough static execution time analysis which yield an upper-bound on the expected WCET. In the proposed real-time mixed-criticality task models, these tasks are additionally characterized according to a low-criticality estimation method such as a measurement-based approach. Since this approach determines an upper-bound on the maximum *observed* WCET, both approaches might differ in their result significantly. This is the main contribution of real-time mixed criticality task models, because it results in defining dynamic execution modes which are switched according to the actual run-time behaviour and utilisation of the tasks.

Based on the assumption that tasks can be assigned such different *criticality scenarios*, the real-time analysis community has provided different scheduling strategies for managing these execution scenarios. It has been shown that mixed-criticality real-time task models combined with run-time strategies for observing the dynamic execution time behaviour of mixed-critical functionality can reduce the utilisation gap while guaranteeing the temporal requirements specified at design-time [14].

4 THESIS CONTRIBUTIONS

Until now, this thesis covered the basic methods and tools of embedded system development and illustrated the challenges in the specification, modelling, development, and validation in system-level design. In particular, we have discussed the challenges regarding MPSoC platforms by focussing on the *platform-based design paradigm* proposed by Sangiovanni-Vincentelli and Martin [71] and presented OSSS which implements this methodology.

We have further provided an introduction to the domain of *mixed-criticality* by identifying two main research challenges. First, we have discussed methods of ensuring temporal and spatial *segregation* in mixed-criticality systems implemented on a common platform. Next, we have identified the *utilisation* issues stemming from safety-critical WCET estimation, and presented optimisation concepts of scheduling policies regarding average-case and worst-case behaviour in mixed-criticality systems.

4.1 CONTRIBUTIONS

This section presents the contributions of this thesis in the context of the scientific questions given in Section 1.3. Figure 4.1 provides an overview of the contributions embedded in the proposed integration flow. To continue the discussion on future mixed-criticality research efforts by Ernst and Di Natale [24], the following contributions aim at increasing the design process efficiency by integrating hypervisor-based segregation techniques with a software component specification model and by providing models for validating the dynamic behaviour of mixed-criticality scheduling policies and platform artefacts in the specification phase and during the integration process on the platform.

CONTRIBUTION C1: We propose a programming model for mixed-criticality software components that allows describing functional behaviour along with temporal constraints and mixed-criticality properties derived from a system specification.

The proposed design and integration flow starts with a system-level description of the functional behaviour in the context of a programming model. Contribution C1 is linked to our first scientific question on how mixed-criticality properties of safety- and performance-critical software applications can be considered along their functional behaviour in an embedded systems design flow. The programming model focusses on *dual-criticality* software components which can be categorised into two distinct sets of functions. Each software component is classified as *high-* or *low-critical*, according to some classification method which considers the component's temporal behaviour and criticality derived by the environmental requirements of the embedded system.

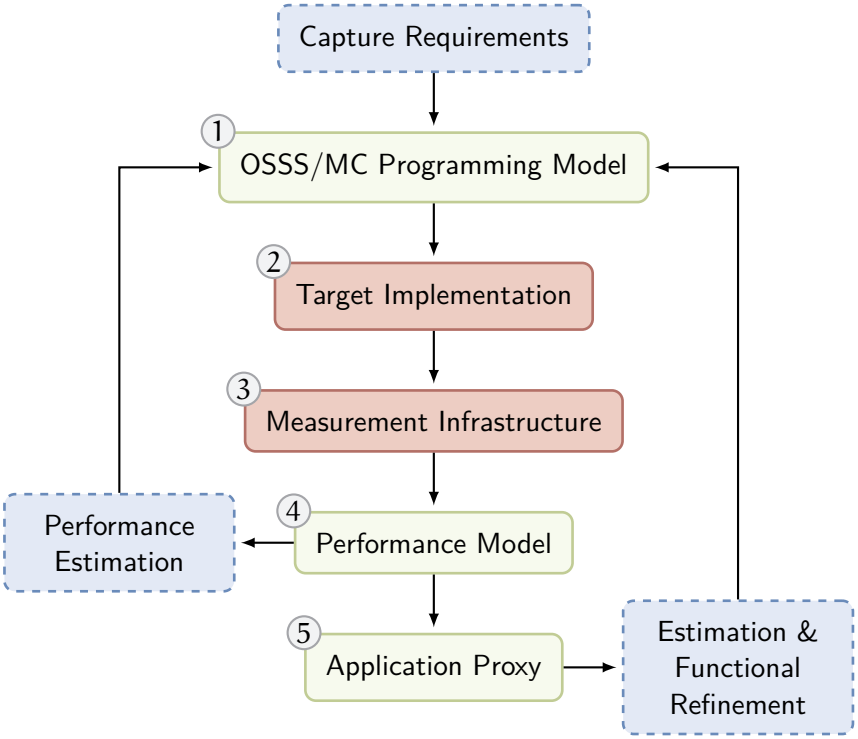


Figure 4.1: Overview of the proposed mixed-criticality design and integration flow along with the contributions of this thesis.

CONTRIBUTION C2: We provide a systematic way to derive an implementation of the programming model based on a virtual resource layer implemented on platform components of a MPSoC.

To answer the question of how to refine the functional description along with its mixed-criticality properties to derive an implementation on an MPSoC, and to determine which platform properties are required to implement temporal and spatial segregation in mixed-criticality systems, we provide an implementation of the programming model on platform components which leverage contemporary mixed-criticality *segregation* and *utilisation* techniques. The refinement step consists of mapping and partitioning decisions which can be expressed by the model. To achieve platform-level *segregation*, the programming model clusters the functional behaviour based on its annotated criticality properties. The function-level criticality annotations are considered in the mapping and partitioning steps such that two subsystems `HI`-critical and `LO`-critical are formed. For improving the system *utilisation*, we implement a contemporary mixed-critical scheduling policy and provide the necessary platform-level monitoring implementation to dynamically optimise mixed-criticality scheduling strategies.

CONTRIBUTION C3: We propose a measurement infrastructure along with the virtual resource implementation to capture dynamic application-level timing behaviour as well as platform segregation overhead.

The measurement infrastructure is embedded into the programming model implementation on the platform and addresses the question of what

dynamic timing properties arise from the application behaviour as well as the platform-level scheduling artefacts and the segregation overhead in an integrated mixed-criticality system.

CONTRIBUTION C4: Based on the timing behaviour of HI-critical applications and the mixed-critical scheduling policy exposed by the measurement infrastructure, we construct a performance model to provide early estimates of performance behaviour in LO-critical applications.

This contribution is linked to the question of how to generate models which represent the integration impact of dynamic mixed-criticality timing properties. The proposed performance model is based on the platform-level execution time measurements observed by the measurement infrastructure and considers the scheduling configuration as well as the segregation overhead of the specified mixed-criticality system. Combined with isolated execution time measurements of a LO-critical software component, the model estimates the expected end-to-end execution time of the component executed on the integrated mixed-criticality platform.

CONTRIBUTION C5: We propose a method for constructing *application proxies* based on the measurements to mimic the temporal behaviour of HI-critical subsystems and provide an integration environment for LO-critical applications on the platform without the need for integrating the functional HI-critical behaviour.

With contribution C5 we answer the question of how the generated timing models of HI-critical software components can be used to separate the individual integration process of HI- and LO-critical subsystems in a mixed-criticality system.

CONTRIBUTION C6: We evaluate the accuracy of the performance model and the application proxy by analysing the resulting timing predictions and the timing behaviour of the LO-critical application when integrated with the HI-critical application proxy.

Finally, to determine the timing accuracy of the models derived from the measured HI-critical timing behaviour in terms of their estimates for LO-critical applications, we evaluate the modelling accuracy by comparing it against measurements with the original applications integrated on the common platform.

4.2 ASSUMPTIONS

The contributions listed above contain the following assumptions about the application model and the targeted platform.

- In the context of our first contribution C1, we focus on exploring resource consumption behaviour of HI-critical real-time sensitive software functionality in combination with best-effort LO-critical behaviour without real-time constraints.

- We further assume that an external analysis tool is able to determine the Worst-Case Response Time (WCRT) of the HI-critical software functionality mapped to the processing element.
- To explore platform segregation techniques regarding temporal and spatial segregation, contribution C2 focusses on the effect of resource sharing of software components mapped to a single processing element.
- Since we incorporate an existing flexible, time-triggered scheduling policy, we assume that an externally provided scheduling analysis has determined whether the task mapping is feasible in the context of the application-level task priorities and deadlines.
- The timing measurement and modelling approach of the contributions C3-C5 consider the temporal behaviour of HI-critical software functions on a fixed set of input stimuli.

Section 10.2 provides an outlook and a discussion on initial concepts to soften these restrictions.

5 RELATED WORK

The work of this thesis covers many research topics due to the diversity of the proposed contributions. This chapter therefore presents the related research activities organised in two sections, representing the topics of the proposed contributions. In the first section, we consider real-time models, platforms, design flows, and industrial efforts in the realm of mixed-criticality systems and compare these efforts to the contributions of this thesis. The second section discusses related work regarding workload modelling and performance models for embedded real-time systems.

5.1 MIXED-CRITICALITY

Research considering mixed-criticality covers many aspects in the topic of embedded systems design. This section presents a top-down approach aligned at the level of abstractions in embedded systems design. The first section presents mixed-criticality real-time models where mixed-criticality is considered in the modelling and scheduling analysis of real-time task models and its criticality-dependant run-time behaviour. The challenges of implementing mixed-criticality functionality on a common platform and various approaches towards mixed-criticality platforms are discussed next. We will also take a look at industrial efforts regarding mixed-criticality platforms. Finally, approaches which combine behavioural models and consider platforms with a design flow are discussed.

5.1.1 *Real-Time Models*

Henzinger and Sifakis [37] discuss critical and best-effort engineering as property to classify systems engineering methodologies. They implicate a widening gap between both approaches, due to their nearly disjoint research communities and practices. In order to bridge this gap, the authors state that methods are needed for guaranteeing a sufficiently strong separation between critical and non-critical functionalities on the platform. At the same time, design techniques need to be provided which allow a flexible use of the platform resources. Real-time task models provide a means of defining these critical and best-effort requirements on the same model and evaluate the impact on scheduling analysis and dynamic run-time behaviour regarding computational resource usage.

The first proposal of considering run-time task criticality properties for scheduling decisions by Vestal [77] considered extending the real-time task model with multiple execution time estimates a task may exhibit at run-time. The intuitive idea was to define multiple *scheduling scenarios* which depend on the observed task execution time at run-time. In particular, Vestal proposed extending the task execution time from a single WCET to consider

multiple execution time estimates. Depending on their level of assurance, the method of determining task computation time can range from estimated, average case values derived by a simulative or measurement-based approach to a high-assurance, statically analysed upper bounded approach. Additionally, besides different computation times, each task contains a criticality level specified by the system designer.

The scheduling policy was then extended with a dynamic system criticality context that is chosen by taking into account the (static) task criticality and its (dynamic) execution time behaviour. If a *high-critical* task operates within its specified *lower-critical* execution time boundary, the scheduling policy exploits the resulting available resources to execute the lower-critical tasks. However, if a *high-critical* task exceeds its anticipated execution time estimates at run-time, the lower-critical task execution is discarded to provide the required resources to the high-critical task set. An extended scheduling analysis ensures that the high-critical tasks always meet their deadlines, regardless of the criticality context the system operates in.

The mixed-criticality extensions proposed by Vestal are based on the periodic task model [53]. The real-time research community further explored the impact on mixed-criticality model and scheduling extensions towards sporadic task models and their implications on scheduling analyses [8] and existing fixed-priority periodic task models [7]. Resulting implications on mixed-criticality scheduling analyses were extensively discussed in the contributions of Baruah et al. [5], Baruah et al. [6], Baruah, Burns and Davis [7] and Baruah and Vestal [8].

The contributions of Mollison et al. [60] and Li and Baruah [54] were among the first to consider mixed-criticality scheduling in the context of multi-processor execution models. The challenge in these extensions rely on the issue of synchronisation regarding criticality scenario switches across parallel running task instances.

Burns and Davis [14] provide an extensive overview of different mixed-criticality task models and the ongoing efforts regarding mixed-criticality scheduling analysis in the context of single- and multi-core systems. Based on the established models provided in the academic literature, this work proposes a specification and programming model based on mixed-criticality properties of the real-time task model to support the notion of *dual-criticality* [7, 54] tasks. Furthermore, the programming model implementation on an MPSoC supports *partitioned* mixed-criticality systems, where tasks are hierarchically scheduled according to their criticality and priority. In contrast to existing scheduling analysis contributions, this work focusses on modelling performance metrics of a `LO`-critical subsystem. As a consequence, this work explicitly considers the impact of `HI`-critical resource usage on the performance degradation in `LO`-critical applications and consequently does not disable the `LO`-critical subsystem when `HI`-critical budget overruns occur.

The focus of recent work on mixed-criticality task models shifted from scheduling independent task sets to communicating tasks and resource access protocols. The work of Giannopoulou et al. [30] and Yun et al. [81] considers criticality-dependent properties and dynamic behaviour regarding task dependencies and resource access protocols. Yun et al. propose platform

support for managing implicit shared resource access of interfering cores by throttling their cache misses. The contribution features a method of determining the task interference budget and a response time analysis incorporating these properties.

Giannopoulou et al. propose the Flexible Time-Triggered Scheduling (FTTS) policy and provide an analysis model for adaptive, mixed-critical systems. Their contribution consists of a flexible, resource-, and criticality-aware scheduling policy that aims at maximising the average run-time resource utilisation in mixed-critical application scenarios without compromising the safety-critical properties of the task set and is thus closely related to the work of this thesis. The FTTS scheduling analysis takes into account the communication dependencies when determining a feasible configuration. To achieve this, a task dependency graph is constructed based on the knowledge of implicit resource sharing between tasks.

The contributions of this thesis build upon the existing research on mixed-criticality scheduling techniques. In particular, this work implements a scheduling policy based on FTTS in the context of single-processor scheduling. The goal of the contributions in this work is to estimate the impact on re-assigned resources for LO-critical software components. While the scheduling policy already provides means for allocating additional resources to the LO-critical software subsystems, our main contribution in this context is the systematic integration approach for functional behaviour specified in the programming model and an implementation based on hypervisor-based segregation techniques.

Such a systematic integration workflow can augment the available mixed-criticality scheduling policies by providing a performance evaluation environment. As we have already stated, the coupling effects of any dynamic scheduling behaviour in mixed-criticality systems directly impacts the performance of the mapped functionality, but it is only visible when all components are integrated on the platform. Our evaluation will demonstrate how a systematic integration approach significantly reduces the effort for comparing different mapping, partitioning, and scheduling configurations.

5.1.2 *Platforms*

The mixed-criticality real-time task models discussed in the previous section make strict assumptions about the underlying platform capabilities. One of these is that the platform has the capability to restrict and monitor the execution to the boundaries of the given execution time estimates. This section provides an overview of the work on mapping mixed-criticality functionality to platforms in the academic literature.

The CompSOC approach proposed by Goossens et al. [32] enables mapping of mixed-criticality software task models on a custom MPSoC consisting of homogeneous *processor tiles* which support composable and predictable mappings. This approach solves the problem of handling post-integration interferences in the design flow, since the platform guarantees that mapped applications are composable regarding their timing behaviour. Resources are statically allocated to the applications depending on their design-time estim-

ates. To guarantee timing compositionality [80], the CompSOC approach features the concept of a virtual execution platform for each mapped application. Each virtual platform guarantees the same timing behaviour and is therefore independent of other parallel executing applications.

The time-triggered architecture approach [46] provides a synchronous mechanism for executing tasks on the platform. The resources are organised into application-specific slots which are executed through a global, time-triggered scheduling policy configured at design time. Due to this slot-based isolation, the mapped applications are unable to interfere their timing behaviour.

The focus of the efforts in the MERASA[1] project [63] lies on managing hard real-time applications and interferences in multi-core processing environments. In particular, regarding the shared cache infrastructure, the bus interconnect arbitration, and the access to the memory subsystems are considered. The resulting platform provides countermeasures against timing interferences on these components along with suitable WCET analysis tools.

The architecture proposed by Liu, Reineke and Lee [57] considers timing predictability by eliminating variable execution times on a software thread-level and proposes hardware-assisted mechanisms for restricting the timing interferences between different threads. A possible benefit of this approach is that processor hardware support for improving the software performance can still be exploited without reducing the analysability of parallel running threads.

The CompSOC approach and the time-triggered architecture by Kopetz and Bauer [46] provide timing guarantees and composability in the context of mixed-criticality systems. However, these static isolation approaches prevent the propagation of unused resources in mixed-criticality scenarios. The MERASA project and the architecture proposed by Liu, Reineke and Lee [57] focus on platforms for restricting timing interferences, but they do not consider reducing the resource utilisation gap in mixed-criticality systems.

The approach proposed by Kritikakou, Marty and Roy [47] provides a run-time mechanism to guarantee that high-criticality tasks can meet their deadlines, where low-criticality tasks can use the available resources detected by the on-line monitoring approach. This contribution is closely related to our approach since the FTTS scheduling policy we have implemented is based on the same principles. However, our design flow additionally features a systematic approach of refining the functional behaviour specified in the programming model and allows the designer to generate application proxies which can mimic the `HI`-critical application behaviour on the target.

Last, the PROXIMA project [38] considers the timing uncertainties regarding measurement-based timing analyses and the WCET. They identify so-called sources of jitters induced by the platform components which cause the results of measurement-base approaches and WCET estimates to significantly differ. Their proposed solution is to provide mechanisms which randomize the sources of jitter on the platform in order to provide a statistical model about the probability of execution time uncertainties. The contributions of this project are closely linked to our modelling approach. However,

1 Multi-Core Execution of Hard Real-Time Applications Supporting Analysability

the custom hardware extensions proposed in the project rule out their applicability in Commercial Off-the-Shelf (COTS) hardware which our contributions focus on.

5.1.3 *Industrial Efforts*

The underlying platform techniques in DREAMS and SAFEPOWER are based on the bare-metal hypervisor XTRATUM [17] which supports segregation techniques and minimising platform-level interferences.

Established industrial solutions for mixed-criticality include the IMA [69] platform which focusses on explicit resource access control and provides isolated partitions to allow executing functionality with different certification levels to the same platform.

Another commercial solution is PIKEOS [67], which supports mixed-criticality partitions, para-virtualisation for RTOSs, and dynamic re-allocation of unused computing resources. These hypervisor techniques are emerging due to their ability to control system partitioning and further provide support for legacy applications with hardware emulation, para-virtualisation, and the use of virtualisation extensions on complex MPSoC platforms.

Our contribution is based on the open source bare-metal hypervisor Xvisor [64]. The implementation provides a virtualised execution environment for applications mapped to the same processing unit. Based on the existing implementation, the technical contribution aims at extending the available application context management technologies towards a dynamic scheduling approach based on the FTTS policy. We have further extended the hypervisor scheduler implementation to support subsystem-independent integration scenarios using the proposed application proxy mechanism and integrated a measurement infrastructure to construct a timing model of the `HI`-critical application behaviour.

5.1.4 *Design Flows*

As discussed above, designing a mixed-criticality system deals with considering *sufficient independence* between functions of different criticality on the platform. A design flow for mixed-criticality systems therefore has to consider criticality on the function level and provide a way to specify the results of FMEA or other fault determination methods, such that the functionality can be mapped to the appropriate application slots on the platform. A mixed-criticality design flow can provide a systematic refinement process while considering the annotated criticalities on the functional units. In this section, we will discuss several design flows and their properties.

The mixed-criticality design flow specified as a part of the CompSOC approach [32] features an incremental refinement with automatic translation of applications defined in cyclo-static data flow programming models and further supporting semi-automatic refinement of applications described with less restrictive computation models such as Kahn Process Networks. This approach is built on top of the proposed composable SoC platform which features timing predictability at the level of clock cycles. The res-

ulting drawback of this design flow is obviously the limited application to other platforms, especially COTS MPSoCs due to the assumptions performed at the system-level and the required platform support to implement these assumptions. An overall goal of the CompSOC design flow is to provide a systematic refinement towards a composable platform, such that timing guarantees can be provided in an isolated manner and still hold in the integrated case. Although there are extensions of CompSOC towards scenario-based application-level scheduling, its primary goal is to provide a time-composable platform.

The mixed-critical design flow proposed by Poplavko et al. [68] features a concurrency language for expressing resource management policies. The design flow aims at integrating the dynamic criticality-aware scheduling policy behaviour into the application models to extend the analysis towards the dynamic criticality scheduling effects. On the platform level, a dynamic runtime manager enforces the timing properties specified in the model. A special focus of this approach is placed on tasks interfering on multi-core architectures. The approach restricts the expressiveness of the application model to enforce temporal segregation. In case of a specification written in a high-level language such as C/C++, the description has to be enriched with metadata to support the notion of criticalities along the functional behaviour. In contrast, out mixed-criticality specification model OSSS/MC combines the mixed-criticality properties with the functional behaviour in a homogeneous environment.

The DREAMS [49] project focusses on distributed mixed-criticality systems. It aims at consolidating platforms into nodes of MPSoC systems and therefore considers CPS and their mixed-criticality properties. The focus of this project lies in the communication across different platforms and their dependencies. The SAFEPOWER [52] project considers the challenge of low-power mixed-criticality systems and aims at providing a reference architecture for the integration and partitioning of mixed-criticality systems on a single device with the goal of reducing the overall power consumption [51].

In contrast to these approaches, our design and integration flow concentrates on the challenge of independently integrating and validating subsystem specifications starting from an initial executable description based on C/C++ to an implementation on a COTS MPSoC on one processing element. With this focus, we make assumptions about the existing specification and restrict the platform mapping capabilities to manage the design space complexity. Our implementation flow further assumes that external scheduling analyses methods can be applied to our programming model properties to verify the correctness and the overall scheduling configuration in relation to the specified performance metrics. Based on the target behaviour, our design flow contains resource consumption models which help in estimating the performance impact of integrated L0-critical applications and provide integration scenarios in a subsystem-independent manner.

5.2 PERFORMANCE & WORKLOAD MODELS

As we have discussed in the chapter on system-level design, the process of validation can be generally categorised into static and dynamic approaches. Several contributions for modelling architectural and platform-related timing properties have been proposed throughout the literature. The underlying characterization to gather appropriate runtime characteristics can be obtained either based on a static code analysis [12, 13, 73] or from a dynamic profiling phase [4, 10, 22, 27, 45, 76].

The limiting factor of the static approaches is that they require accounting for the dynamic properties on the platform at the source level of the input behavioural model. This is especially challenging in the context of MPSoC and shared resources on the platform. The reason for mixed-criticality utilisation enhancements is precisely the over-estimation of platform resource consumption that result from the static analysis approach.

Performance validation approaches can include more than just the computation resource usage of tasks on their mapped processing elements. The approach in [21], which is based on a pure source-level performance estimation, has been extended in [31] with a cache model based on the resulting host address distribution in a native source-level simulation which predicts memory accesses on the platform. A limiting factor of this approach is the challenge to derive the *target* platform behaviour from the high-level view on the input model. The cache model used in [40] is not utilised for traffic generation and instead adjusts the annotated delays based on miss rates. The characterization of the target binary code proposed by Stattelmann, Bringmann and Rosenstiel [72] especially supports optimised code through a path simulation of the target delay model in parallel to the native simulation of the functional code. While these contributions provide powerful early estimates of input behavioural models on the expected target platform, they are limited in their applicability due to their reliance on incorporating platform-specific knowledge in the modelling process.

We propose a data-driven performance model similar to statistical approaches [22, 39, 62]. Our performance modelling approach relies on measured data of the platform to train a statistical model which represents the execution time distribution. By constructing this statistical model based on the measured data, the model captures the execution time distribution of software components on a target processor. In the context of our design flow, we focus on the computational resource consumption of tasks since our hypervisor implementation provides platform segregation mechanisms. Thus, access to caches and the memory subsystem do not need to be represented by the model since they do not interfere the execution of other applications regarding their timing behaviour.

Besides a performance analysis targeting end-to-end execution time estimation for `LO`-critical software functions, we also construct an application proxy to mimic the temporal behaviour of the `HI`-critical software on the target platform. The application proxy behaviour also focusses on replaying the timing behaviour due to our platform-level restriction of timing interferences. The hypervisor-based isolation approach therefore avoids the need

for complex workload representations involving memory access or further usage patterns on other resources [4, 10, 27, 76].

In summary, the presented approaches allow the generation of benchmarks mimicking the spatial and temporal memory access behaviour on the target architecture beyond the CPU boundary and consequently include the memory hierarchy properties of the platform. They can therefore be used for a fast and early performance evaluation of memory hierarchy configurations including caches and page size settings. Our approach however does not need to incorporate architectural information from the memory subsystem or the cache infrastructure. The advantage of our application proxy approach therefore is the flexibility towards different platform configurations.

5.3 SUMMARY

This work builds upon the efforts regarding mixed-criticality real-time scheduling and analysis by incorporating mixed-criticality task model properties in its specification model and targeting the FTTS scheduling policy on a MPSoC. While these models and techniques aim at providing timing guarantees on each criticality level, this work instead focusses on modelling the timing impact of dynamic, high-critical applications on performance-critical, QoS applications. The contributions of this thesis focus on a systematic integration of functional behaviour specified in an executable model and a refinement towards a hypervisor-based implementation on a MPSoC.

As discussed, the difference to existing platform-based approaches is our focus on COTS MPSoC which provide greater flexibility regarding platform choices. The contributions are therefore based on a type-1 hypervisor implementation and extend it towards the dynamic mixed-criticality scheduling aspects captured in the C++ specification model.

Finally, the contributions regarding the `HI`-critical timing models extend the current state-of-the-art by exploiting the hypervisor-based segregation techniques implemented on the mixed-criticality platform. The temporal and spatial segregation implemented on the platform enables constructing timing models which accurately represent the resource utilisation on the platform, while ignoring additional resource consumption patterns such as memory usage and cache behaviour modelling. The proposed *application proxy* workload model abstracts away detailed platform information and enables the integration of `LO`-critical applications independent of the functional behaviour of a `HI`-critical application.

Part II

DESIGN FLOW & MODELLING APPROACH

6 MIXED-CRITICALITY SYSTEM DESIGN FLOW

The previous chapters have shown that the iterative process of designing a MPSoC relies on the feedback of design decisions as early as possible in the design flow. This chapter describes the proposed design flow for embedded mixed-criticality systems on MPSoCs. Its goal is to provide an evaluation environment during the refinement steps such that the designer can validate its mapping, partitioning, and implementation decisions. In the context of mixed-criticality system design, as has been discussed in the previous chapters, dynamic scheduling artefacts should be modelled and evaluated during the refinement of the system as well.

The first step in a model-based design flow is to capture the requirements of the application. The design process typically starts with a *specification model* to express the requirements of the embedded system in terms of its (functional) behaviour as well as its (physical) constraints. This behaviour is either manually specified using some high-level programming language, or provided as a model of computation which can be transformed to high-level C/C++ source code. In particular, *Matlab/Simulink* provides a modelling and simulation toolchain for developing control algorithms and modelling their environment. The models can be exported to a high-level language such as C/C++ and targeted towards software platforms or exported for a high-level synthesis for hardware designs. In the case of software platforms, the next step is to partition the high-level source code into tasks and communication objects of an executable specification, which define semantics that are used for a systematic refinement to a target platform.

This section describes such a specification model consisting of two layers. The Application Layer (AL) contains components that allow partitioning the functional behaviour in terms of *tasks* and *Shared Objects* to express computation and communication. Next, the Virtual Resource Layer (VRL) describes resource models to which the AL can be mapped. The specification model contains the application behaviour, its timing constraints, and captures mixed-criticality properties in the AL and VRL. Based on this specification model, the *OSSS/MC programming model* and its execution semantics are introduced and described in Section 6.3.

An untimed simulation of the programming model components on resource models then provides feedback on the functional correctness of the partitioning of the behaviour into tasks and Shared Objects. It therefore defines the first validation and analysis step in the SoC design flow. Section 6.1 proposes the specification model while Section 6.2 presents a resource model designed to provide further insight into the mixed-criticality scheduling behaviour when the specified AL components are mapped to the computation resource. Section 6.3 then discusses the behavioural semantics of the runtime model with mapped tasks and Shared Objects. After discussing the communication and computation refinement step, Section 6.4 describes the implementation of the modelling components on a target plat-

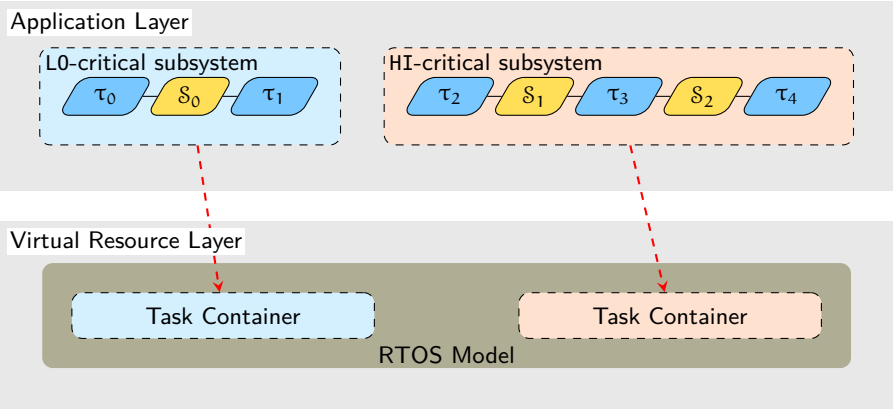


Figure 6.1: Application Layer with tasks and Shared Objects, clustered using two criticality levels and mapped to the Virtual Resource Layer.

form. This refinement step consists of computation and communication refinement as well as target code generation and platform configuration.

6.1 APPLICATION LAYER COMPONENTS

The specification model components are defined in the AL, as illustrated in Figure 6.1. The desired functional behaviour is expressed using active *tasks* which are based on actor-oriented design principles [50]. Communication between tasks is explicitly described via passive *Shared Objects*. The separation of communication and computation via Shared Objects and tasks is realised through task *ports* which are bound to matching Shared Object interfaces. Shared Objects then implement the desired functionality accessible through the ports. The binding relation of task ports and Shared Objects is derived by partitioning the functional behaviour into AL components. It is therefore part of the system specification.

The AL components capture the structure and functional behaviour derived from a descriptive system specification. We assume that the functions of this description have been categorised by their *criticality*, e. g. as a result of performing a domain-specific FMEA. The specification model components provide properties to capture the determined criticality.

Definition 1 (Application Layer configuration)**.** *An* Application Layer (AL) configuration $\mathbb{C}$ *is defined as a tuple* $(\mathcal{T}, \mathcal{S}, \mathcal{B}, \mathcal{L})$ *consisting of a set of tasks* $\mathcal{T}$*, Shared Objects* $\mathcal{S}$*, a binding relation* $\mathcal{B}$ *specifying the connections of task ports and Shared Object interfaces, and criticality levels* $L_i \in \mathcal{L}$. □

As already stated, this work considers functions classified by two criticality levels $\mathcal{L} = \{\texttt{LO}, \texttt{HI}\}$, representing performance-driven LO-critical- and safety-critical HI-critical real-time sensitive functions. As a consequence, the specification model components either describe LO- or HI-critical behaviour which we refer to as *subsystems*:

Definition 2 (LO-critical/HI-critical subsystem)**.** *A* LO-critical (HI-critical) subsystem *describes the subset of AL components attributed with the criticality LO (HI).* □

In a next step, the AL subsystems are mapped to resource models on the VRL, where they are encapsulated in *task containers*. Following the typical refinement flow in the design of embedded systems, as a first step towards an implementation, the mapping and functional behaviour can be validated using a SystemC-based simulation. In the next step, the AL components can be implemented on a target platform using a backend which implements the model semantics on top of RTOS primitives.

6.1.1 *Tasks*

Within the AL, a *task* represents an active, concurrent, periodically executing entity which can communicate via ports bound to Shared Objects. Each task is associated to a given criticality defined in the AL configuration. Additionally, certain dynamic criticality properties that depend on the run-time behaviour can be specified. This section provides the definition of task behaviour and its properties.

A general issue regarding timing estimation with a generic specification based on high-level languages such as C++ is their richness in expressing functional behaviour. To be applicable to WCET analysis tools, the behaviour has to fulfil properties which guarantee *bounded execution time*, which describes freedom from unbounded execution which can be caused by infinite loops or recursion patterns. We therefore restrict the task behaviour in our model by providing semantics which the C++-based implementation must adhere to. We use the notion of *activation traces* to express bounded computation and explicit synchronisation points in the execution trace of a task.

Definition 3 (Task execution trace)**.** *Given a task* τ_i*, an execution trace* $\vec{\xi}_i \subset \nu^*$ *is a finite sequence of* activity tuples *representing functional behaviour and an annotated timing property. An activity tuple* $\nu \in \pi_i \times \mathbb{R}^+$ *denotes communication through a port* π_i *or local computation* $(\emptyset, t)$ *with an execution time* $t \in \mathbb{R}^+$*.* □

The task can then be defined as follows.

Definition 4 (OSSS/MC task)**.** *Let* $\mathcal{I}$ *be the set of all Shared Object interfaces in the AL configuration* $\mathbb{C}$*. A* task $\tau_i \in \mathcal{T}$ *is a tuple* $\left(T_i, \pi_i, \ell_i, \vec{\xi}_i\right)$*, consisting of*

- *a period* $T_i \in \mathbb{R}^+$ *(equal to the task deadline),*
- *a set of ports* $\pi_i \subseteq \mathcal{I}$*, representing the binding to the associated Shared Object interfaces,*
- *a static criticality* $\ell_i \in \mathcal{L}$*, and*
- *a set of* execution traces $\vec{\xi}_i \subset \nu^*$ *as described in Definition 3.*

We assume that for each criticality level $L \in \mathcal{L}$*, there exists a way to determine the execution time of a task's execution trace set, which we represent with the function* $C_L : \vec{\xi}_i \to \mathbb{R}^+$*. As discussed, the timing analysis can therefore be performed with different methods depending on the level of assurance and the*

resulting required execution time guarantees. Throughout the paper, the notion $C_L(\tau_i)$ *refers to the maximum execution time of an execution trace set* $\vec{\xi}_i$ *of task* τ_i. □

In general, determining the task execution time requires the knowledge of the AL mapping and the complete system scheduling state due to the possible interferences induced by communication on the platform. Since this severely limits the overall system analysability, many models of computation therefore restrict the actor behaviour expressiveness by defining phases in which communication and computation is performed and upper-bounded [66] in the resource consumption. Although we are not focussing on scheduling and mapping analyses in this work, our proposed execution trace model limits the task behaviour expressiveness to support integrating such external response time analysis methods.

We support criticality-dependant task computation times by defining different ways of determining the worst-case execution time of an execution trace set $\vec{\xi}_i$. In our model, we assume that for a task, execution traces are *functionally equivalent* in all criticality levels but may differ in their *temporal behaviour* due to a more pessimistic execution time estimates: $L_i > L_j \Rightarrow C_{L_i}(\vec{\xi}_i) \geqslant C_{L_j}(\vec{\xi}_i)$. In contrast, a LO-critical task can exhibit different *functional behaviour* across criticalities, such that an execution trace $\vec{\xi}_i(\texttt{LO})$ represents resource-demanding behaviour whereas $\vec{\xi}_i(\texttt{HI})$ captures the resource-constrained behaviour. As a consequence, for low-critical tasks, $L_2 > L_1 \Rightarrow C_{L_2}(\vec{\xi}_i) \leqslant C_{L_1}(\vec{\xi}_i)$. In this work, we focus on modelling criticality-dependant temporal behaviour of HI-critical subsystems and fixed functional behaviour in LO-critical components. Our previous work in [43] demonstrated how to model and support criticality-dependant functional behaviour for LO-critical applications, such that they can dynamically adapt to changing resource usage scenarios. This work however focusses on the temporal differences and does not consider the functional criticality-dependant behaviour.

Example 1 (OSSS/MC HI-critical task)**.** *In this example we model a* HI*-critical task* τ_0 *which periodically performs a local computation. The task executes with a period of* 2 ms*. A measurement-based worst-case execution time has determined* 1 µs *while the statically derived, upper-bounded execution time is* 200 µs*. The properties of* τ_0 *are* $T_0 = 2\,\text{ms}$ *and* $\pi_0 = \emptyset$*. Due to different execution time estimate methods,* $\vec{\xi}_0$ *models the same functional behaviour, but its execution time varies across criticality levels:* $C_{LO}(\vec{\xi}_0) = 1\,\mu\text{s}$ *and* $C_{HI}(\vec{\xi}_0) = 200\,\mu\text{s}$.

6.1.2 *Shared Objects*

Communication within OSSS/MC is explicitly modelled using *Shared Objects*, which are based on the semantics of Ada's *protected objects* [16]. Each Shared Object implements one or more *interfaces* which tasks can access through a structural binding of their matching *port*. This separation enables a structural refinement of task communication, since the internal task behaviour and the interface methods it uses are separated from the underlying implementation of the communication channel. In the refinement step, where

tasks and Shared Objects are allocated to platform resources, ports serve as transactors which model the interface behaviour for accessing Shared Objects depending on their allocation. If a task is structurally bound to a shared object residing in the same resource, the port can implement simple memory access to forward the communication requests. On the other hand, if task and Shared Object model communication across resource boundaries, the port implements the necessary access to the interfaces for communicating to the other resource.

Definition 5 (Shared Object). *Let $\mathbb{C}$ be an AL configuration defining the criticality set $\mathcal{L}$. A* Shared Object $\mathcal{S}_i$ *is a tuple* $(\Sigma_i, M_i, \mathcal{I}_i)$ *consisting of an* internal state Σ_i*, describing user-defined, abstract data types, a set of* methods *or* services $M_i : \Sigma_i \to \Sigma_i'$*, representing operations on the object's internal state, and a set of* interfaces $\mathcal{I}_i \subseteq \mathcal{P}(M_i)$ *the object provides.* □

To structurally separate computation from communication, task communication is explicitly modelled using ports. Access to a particular Shared Object is furthermore defined statically via *port binding*. In the context of mixed-criticality task properties, the binding also adheres certain restrictions, which we define now.

Definition 6 (Shared Object binding). *A task port π_k can be* bound *to a matching Shared Object when the Shared Object implements the corresponding port interface, as defined in the $\mathcal{I}_i$ set. The function $\mathcal{B} : \pi \to \mathcal{S}$ defines the* binding *of task ports to Shared Objects. Note that multiple ports can be bound to the same Shared Object instance. In that case, a locking mechanism guarantees mutual access to the Shared Object state. Locking and access behaviour depends on the maximum criticality of all tasks bound to the object. We call this property the* ceiling criticality $\mathtt{ceil}(\mathcal{S}_i)$*. A task with criticality $L_k < \mathtt{ceil}(\mathcal{S}_i)$ can only access methods which do not modify the internal object state Σ_i. This allows* `LO`*-critical tasks to access information provided by* `HI`*-critical tasks without inducing temporal or spatial interference.* □

Example 2 (OSSS/MC Shared Object). *A Shared Object $\mathcal{S}_0$ for communicating block-wise image data can be modelled as follows. We describe the interface as $\mathcal{I}_0 = \{\{get\_block, set\_block\}\}$ such that the Shared Object provides two methods get_block $: \Sigma_0 \to \Sigma_0$ and set_block $: \Sigma_0 \to \Sigma_0'$ for block-wise reading and writing of image data (with a read-only and a write access to the internal state Σ_0). Suppose the port π_0 of τ_0 (from example 1) and the port π_1 of a second task τ_1 with $L_1 =$ `HI` are both bound to $\mathcal{S}_0$, i.e. $\mathcal{B} = \{(\pi_0, \mathcal{S}_0), (\pi_1, \mathcal{S}_0)\}$. Since $\mathtt{ceil}(\mathcal{S}_0) =$ `HI`, τ_0 can access both methods, while τ_1 can only access* `get_block` *due to the Shared Object binding constraints defined above.*

6.2 RUNTIME MODEL

As discussed in the foundations of mixed-criticality systems, mapping functions of different criticality imposes requirements on the underlying platform in terms of its segregation properties. In this thesis, we are interested in evaluating *segregation* properties regarding temporal and spatial behaviour. The following definition provides a distinction between *segregation* and *isolation*.

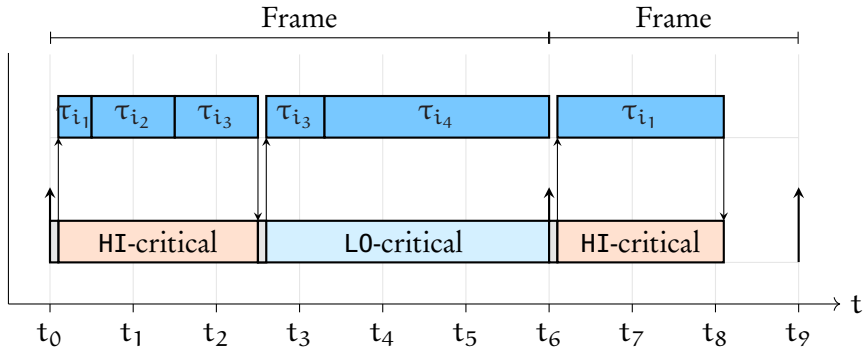


Figure 6.2: Illustration of the scheduling and execution semantics of tasks (top) and containers (below) of two subsystems representing HI-critical and LO-critical behaviour.

Definition 7 (Segregation and Isolation)**.** *Subsystems are* segregated *iff their interference on shared resources is bounded in both space and time. Furthermore, subsystems are considered to be* isolated *when their interference on shared resources is prevented.* □

While *isolation* can be achieved by statically assigning resource consumption slots to each subsystem and a TDMA scheduling policy, the result obviously stands in contrast to the goals of mixed-criticality scheduling policies. Their aim is to dynamically re-allocate unused resources from one subsystem to another. Since this dynamic resource re-allocation behaviour can change the temporal behaviour of another subsystem, a system featuring mixed-criticality scheduling policies aims at enforcing *segregation*.

6.2.1 *Modelling Scope*

This section proposes a runtime model which provides scheduling and execution semantics on a computational resource for mapped tasks and shared objects. The model provides *task containers* for each mixed-critical subsystem defined in the AL, which implement an isolated execution context for the tasks and a container-local scheduling policy. The runtime model scheduling semantics are based on the FTTS scheduling policy proposed by Giannopoulou et al. [29] which separates the computation resource temporally into *frames*. Each frame consists of a series of *slots* – execution blocks dedicated to each task container. Since our system model considers two criticalities LO and HI, each frame therefore consists of two slots.

The frame sequentially executes its slots according to their criticality in descending order. Thus, in the dual-criticality model of this work, the HI-critical slot is executed at the start of a frame, followed by the LO-critical slot. At the root, the spanning hierarchical scheduling policy therefore schedules a sequence of *frames* containing *slots* which encapsulate each task container. An illustration of the scheduling semantics is depicted in Figure 6.2. As can be seen, within each frame, task containers are executed sequentially, while each container implements the execution sequence of its mapped tasks. Task

containers are preempted after each frame such that it is guaranteed that the HI-critical container can start execution in the new frame. The flexibility of this scheduling approach – compared to a TDMA schedule – lies in its slot execution behaviour. The duration of the HI-critical slot depends on the WCRT of the mapped HI-critical task set in this frame. After execution the task set, the task container yields the execution control back to the runtime model, which in turn can directly switch the context to the LO-critical slot.

The allocation of tasks to the frames (and implicitly slots) is derived from the periodic execution and reaction requirements defined in the system specification. As an example, when designing control algorithms in Matlab/Simulink, the model of the generated C/C++ code requires a specific execution frequency to maintain its semantics. These requirements have to be mapped to the scheduling configuration of tasks within frames. We consider this task and frame allocation as an input to our approach and therefore allow the designer to define the allocation in the model. An algorithm for choosing the slot allocation based on such task parameters is presented in [29].

6.2.2 *Runtime Model Properties*

The *runtime model* represents a hierarchical two-level scheduling policy based on the FTTS policy proposed by Giannopoulou et al. [29]. In our implementation, mapped tasks are clustered by their criticality into *task containers*. On the first scheduling level, each task container is executed in a *slot* which represents an exclusive computation timespan within a *scheduling frame* of the top-level scheduling policy. On the second level, an *application-local* policy manages the execution order of the task set within the slot. Shared Objects can be mapped *container-local* or *-global*, depending on the interface binding properties of the Shared Objects. However, they can only be considered local when all bound tasks have the same criticality level. The *global* Shared Object container allows access from different criticalities, consequently supporting communication across task containers, under the restrictions defined above.

Summarising, the properties of the runtime model are expressed in the following definition.

Definition 8 (OSSS/MC runtime)**.** *A runtime* $\mathcal{R}_i = (L_i, \theta_i, \Pi_i, F_i)$ *consists of a dynamic criticality level* L_i*, a function* $\theta_i : \mathcal{L} \to \mathcal{T}$ *denoting the tasks mapped to the runtime (clustered by criticality), a function* $\Pi_i : \mathcal{L} \to \pi$ *assigning a scheduling policy for each criticality and a sequence of* scheduling frames $F_i = \langle f_0, \ldots \rangle$*. For any criticality* $L \in \mathcal{L}$*, the set* $\theta_i(L)$ *refers to the tasks of runtime* $\mathcal{R}_i$ *and* $\Pi_i(L)$ *denotes the scheduling policy of the task set. We further define the hyperperiod* H_F *as the sum of all frame durations* $\sum_{f_i \in F} \delta_k$*.* □

6.2.3 *Scheduling Configuration*

The frame *scheduling configuration* consists of a sequence of scheduling frames. Each frame is further divided into *slots* that execute the HI-critical

or LO-critical tasks in an order defined by the respective scheduling policy. The properties of the scheduling frame configuration are defined as follows.

Definition 9 (Scheduling frame)**.** *Given a runtime* $\mathcal{R}_k$*, a* scheduling frame *is a tuple* $f_i = (\mathcal{T}_i, \delta_i, M_i)$ *consisting of a set of* high-critical tasks $\mathcal{T}_i \subseteq \mathcal{P}(\theta_k(\mathit{HI}))$ *which are executed in the frame, a* frame duration $\delta_i \in \mathbb{R}^+$*, and a set of* margins $M_i : \mathcal{L} \to \mathbb{R}^+$ *representing the estimated execution time duration of the task set* $\mathcal{T}_i$ *for all criticalities* $L \in \mathcal{L}$*. The margin values depend on the end-to-end execution time of* $\mathcal{T}_i$*. After executing* $\mathcal{T}_i$ *in the slot, the remaining length* r *of the frame duration depends on the run-time behaviour of* $\mathcal{T}_i$*, where* $r \geqslant \delta_i - M_i(\mathit{HI})$ *in a* HI*-critical frame run and* $r \geqslant \delta_i - M_i(\mathit{LO})$ *in a* LO*-critical run. This duration is then allocated to the task container* $\theta_k(\mathit{LO})$*.* □

The margin values determine the execution mode for the LO-critical slot in each frame. The underlying concept is to expose the dynamic resource consumption behaviour of the HI-critical subsystem to the LO-critical tasks. When a LO-critical task is instantiated, it can act upon the information about the resource consumption of the HI-critical subsystem in this frame. Possible behavioural adaptions include reduced resource consumption scenarios or other degraded execution modes, which may prevent the LO-critical subsystem from missing deadlines. We will consider this extension in the outlook in Section 10.2, while focussing on the effects of the overall resource availability caused by dynamic scheduling effects in this thesis.

6.2.4 *Mapping*

The components in the AL are mapped to the resources via mapping relations. Within OSSS/MC, mapping restrictions are defined that apply to the AL components. This ensures spatial and temporal segregation between subsystems of different criticalities.

Definition 10 (Application Layer (AL) to Virtual Resource Layer (VRL) mapping)**.** *The* AL to VRL mapping *consists of two functions* $(\mu_\mathcal{T}, \mu_\mathcal{S})$*. A* task mapping *function* $\mu_\mathcal{T} : \mathcal{T} \to \mathcal{R}$ *assigns each task* $\tau_i \in \mathcal{T}$ *with the criticality* ℓ_i *to a runtime model* $\mathcal{R}_k \in \mathcal{R}$*. A* Shared Object mapping *is a function* $\mu_\mathcal{S} : \mathcal{S} \to \mathcal{R}$ *which assigns each Shared Object to a task container (*local *use) or to the runtime (*global *use) to manage access to the shared resource.* □

Intuitively, tasks are clustered into their corresponding criticality subsystems, and Shared Objects either exist as a means of communication within a criticality cluster, or across. However, if information should be transmitted across criticality clusters, the model ensures that LO-critical subsystems cannot modify the Shared Object state. Therefore, LO-critical subsystems are prevented from spatially interfering with HI-critical subsystems.

6.3 EXECUTION SEMANTICS & SIMULATION

At the start of a new frame, the runtime model $\mathcal{R}_k$ picks the next frame from the sequence F_k and configures a timeout according to the frame duration. In the first slot of the frame, the HI-critical task set executes. As soon as the

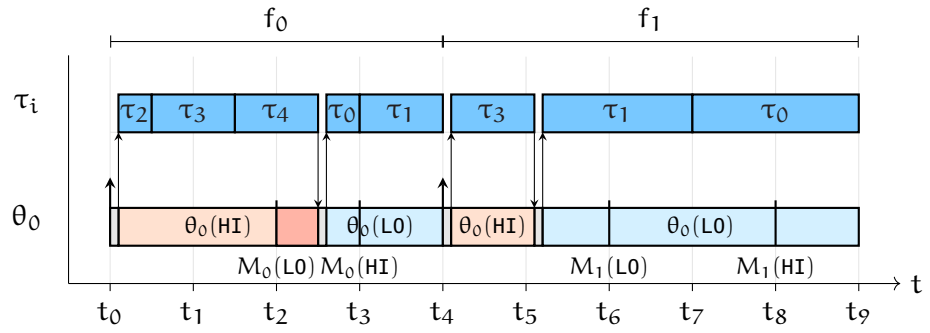


Figure 6.3: Example execution trace of two frames f_0, f_1 from a scheduling configuration of an OSSS/MC runtime $\mathcal{R}_0$.

set has finished, the criticality level L is updated according to the execution duration and the LO-critical task set executes. It will be preempted when the frame timeout has been reached. Intuitively, margin $M_i(\texttt{LO})$ represents a decision point which, at run-time, determines the current system criticality mode based on the execution time of the critical task set in the frame. If the task set takes longer than anticipated in a given criticality, the system criticality increases and the LO-critical task container may use this system state knowledge to adjust its computational demands. A task set executing for longer than $M_i(\texttt{HI})$ can be considered faulty and appropriate actions such as a system reset may be performed by the runtime.

Based on the properties and semantics of the specification model presented above, we have implemented the OSSS/MC programming model featuring AL task and Shared Object components as well as the VRL runtime model on top of SystemC. This enables describing or integrating the functional behaviour in C++ and provides a consistent refinement flow from a native, annotation-based executable specification to an implementation candidate which can be targeted towards the platform with the help of a cross-compilation toolchain. OSSS/MC therefore integrates with existing code generators such as *SimuLink Coder* as the missing link between the generated code and the partitioned functional behaviour. The programming model provides a component-based seamless refinement flow towards a target implementation. Furthermore, we make extensive use of design introspection by providing a configuration file that is generated at SystemC elaboration time. This configuration contains the application and VRL model properties and can be used as a basis for further analysis tools to check the system configuration consistency or evaluate its scheduling properties.

Example 3 (Runtime behaviour)**.** *Figure 6.3 depicts an example execution trace from a runtime model $\mathcal{R}_0$ with tasks mapped as shown in Figure 6.1. The scheduling policy configuration spans a hyperperiod consisting of two frames $F_0 = \langle f_0, f_1 \rangle$ configured as $\mathcal{T}_0 = \{\tau_2, \tau_3, \tau_4\}$ with $\delta_0 = t_4 - t_0$ and $\mathcal{T}_1 = \{\tau_3\}$ with $\delta_1 = t_9 - t_4$. With $t_0 = 0$, we assume that for f_0, a scheduling analysis has determined $M_0(\texttt{LO}) = t_2$ and $M_0(\texttt{HI}) = t_3$, as well as $M_1(\texttt{LO}) = t_6$ and $M_1(\texttt{HI}) = t_8$ for f_1. The LO-critical subsystem is defined as $\theta(\texttt{LO}) = \{\tau_0, \tau_1\}$.*

As can be seen in the example, the observed execution time of $\mathcal{T}_0$ is higher than $M_0(\texttt{LO})$ (at t_2) which results in an allocation of *more than* $M_0(\texttt{LO}) - t_0$ and *up to* $M_0(\texttt{HI}) - t_0$ time units for the high-critical task set $\mathcal{T}_0 \in \theta_0(\texttt{HI})$ to provide the necessary resources for completing the computation (since the scheduling policy must assume HI-critical temporal behaviour). $\theta_0(\texttt{LO})$ thus cannot use the slot duration of $\delta_0 - M_0(\texttt{LO})$ and instead executes in a degraded scenario when $\theta_0(\texttt{HI})$ yields, which is after the $M_0(\texttt{LO})$ mark. Starting with the new frame f_1 at t_4, the task container $\theta_0(\texttt{LO})$ is preempted and $\mathcal{T}_1$ executes, now within its $M_1(\texttt{LO})$ duration. This results in $\theta_0(\texttt{LO})$ being able to execute for the remainder of the frame duration which is at least $r = \delta_1 - M_1(\texttt{LO})$ since the scheduling policy has observed LO-critical temporal behaviour of $\mathcal{T}_1$.

6.3.1 *Application Layer Components*

The task and Shared Object models are provided as SystemC base class modules which contain methods and processes for executing the functional behaviour. A user-defined task inherits from the task base class and implements the behaviour in an abstract virtual method. An example task definition is shown in Listing 6.1. Each task contains its local context, enabling the defined subclass to specify local state (such as the `val` attribute in the example below). This state is preserved across invocations to the task behaviour. The task base constructor defines the properties specified in the task model. The notation uses the static C++ type system to enforce compile-time checks on the types passed, thereby minimising errors such as mixing deadline and period properties. Although not internally used for scheduling, the properties defined here are nevertheless exposed in the model introspection configuration and can be used as an input to external scheduling analysis tools.

The inheritance approach provides great flexibility in defining the behaviour since it encourages code re-use across multiple similar tasks. The designer can refactor similar tasks such as wrappers for external functions or generated code and thus greatly simplify the specification overhead and reduce the necessary code maintenance. Convenience macros can additionally be used to simplify the task definition.

The example in Listing 6.1 also illustrates the use of timing annotations in the task behaviour. Depending on the task state, either 2 or 3 seconds are consumed on the resource to which this task has been bound to. These high-level annotations are ment to provide early insights into the runtime model scheduling behaviour.

6.3.2 *Runtime Model*

The runtime model can be instantiated in a user-defined top-level module and provides methods for defining the scheduling configuration as well as the task binding. An example platform definition consisting of two tasks and a runtime instance can be seen in Listing 6.2.

```
#include "modelling/osssmc.h"

using namespace osssmc;

class MySafetyTask : application::task {
  int val = 0;

public:
  OSSSMC_TASK_CTOR(MySafetyTask, criticality::safety,
                   task::priority_type(2),
                   task::deadline_type(10_s),
                   task::period_type(10_s))
  {}

  OSSSMC_BEHAVIOUR(scenario) {
    (void)scenario;
    if (val % 2 == 0) {
      OSSSMC_CONSUME(2_s) {}
    } else {
      OSSSMC_CONSUME(3_s) {}
    }
    val++;
  }
};
```

Listing 6.1: OSSS/MC user-defined task behaviour.

The method `create_frame` defines the hierarchical scheduling setup of the runtime instance. The first argument to `create_frame` is the frame length, the second argument is a list of margins that define the relative time stamps in the frame when a scenario switch should take place. The third argument is the sequence of tasks specifying their mapping to this frame. Each task mapped to the frame is executed in the appropriate slot depending on its criticality.

In this example, both task instances are mapped to the `HI`-critical slot of the frame, since they are defined with the safety criticality property, as shown in Listing 6.1. The execution order in this slot is defined by the mapping order. Therefore, `safety_task` will be executed as the first task in the slot, followed by `safety_task2`. Multiple frames can be defined by subsequent calls to `create_frame`. Once all frames have been executed by the scheduler, the runtime repeats the configured schedule with the first frame.

6.3.3 *Instantiation & Simulation*

The OSSS/MC model can be simulated based on the SystemC simulation semantics. In the elaboration phase, the system models are initialised according to the user-defined hierarchy. This is achieved by instantiating the user-defined modules, such as the `Top` module in our example. Subsequent constructors invoked by this instantiation will perform the instantiation of

```
#include "modelling/osssmc.h"

using namespace osssmc;

SC_MODULE(Top)
{
  MySafetyTask safety_task, safety_task2;
  platform::runtime rt;

public:
  SC_CTOR(Top)
    : safety_task("MySafetyTask")
    , safety_task2("MySafetyTask2")
    , rt("runtime")
  {
    rt.create_frame(10_s, { 5_s, 8_s },
                          {safety_task, safety_task2});
  }
};
```

Listing 6.2: OSSS/MC platform definition and runtime scheduling configuration.

```
#include "modelling/osssmc.h"
#include "Top.h"

int osssmc_main() {

  Top t{"Top"};

  return osssmc_start();
}
```

Listing 6.3: OSSS/MC platform instantiation and simulation start.

the tasks and runtime as well as the setup of the scheduling frames. The simulation finally starts with a call to `osssmc_start`.

By wrapping the SystemC simulation mechanism with custom methods, we can replace the underlying functionality depending on different implementation backends. With the SystemC simulation backend, the function `osssmc_main` is called from `sc_main` directly. A possible target implementation instead can initialise the platform before calling `osssmc_main`. Similarly, all classes presented in this section can have a corresponding backend for an implementation on a target. The next section discusses the implementation of each OSSS/MC component on an ARM-based embedded target platform.

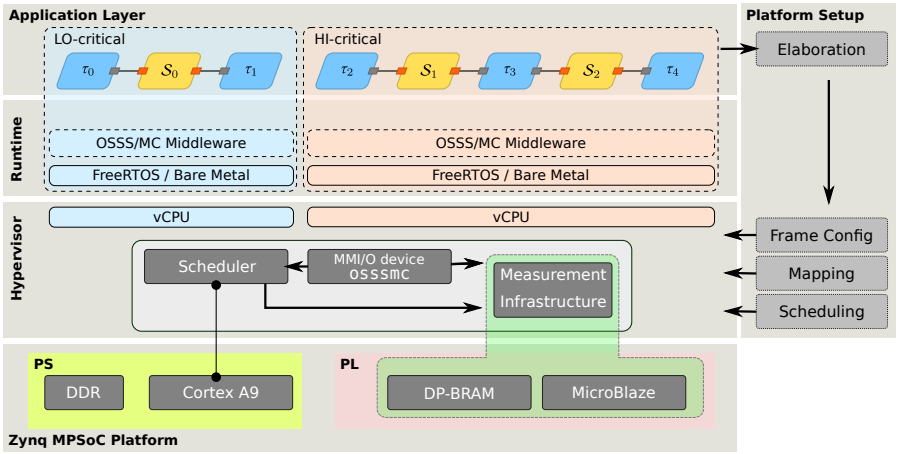


Figure 6.4: Overview of the Programming Model implementation layers on a target platform.

6.4 TARGET IMPLEMENTATION

The previous section defined the OSSS/MC programming model components and their properties. This section describes an implementation of the modelling components on a Zynq 7000 series MPSoC target platform consisting of a dual-core ARM Cortex-A9 processor and a FPGA for implementing custom logic.

The MPSoC (as depicted in Figure 2.4 on page 16) is separated into two parts. The Processing System (PS) consists of the dual-core Cortex A9 processing unit, a two-level cache hierarchy, a DDR memory controller, a bus infrastructure, and various other peripherals for accessing the external interfaces such as USB, Ethernet, HDMI, etc. The Programmable Logic (PL) can be used to implement user-defined custom logic such as additional soft-cores, Block-RAM (BRAM) memory components, and hardware accelerators. A routing mechanism in the PS enables the use of external peripherals in the PL by redirecting their pins into the logic part. Furthermore, bus interfaces exist for communicating with the PS, which can be realised either by accessing target memory components from the PL or by implementing BRAM targets in the PL which can then be mapped to the memory address space of the PS bus masters.

The programming model and the measurement infrastructure have been implemented using both the PS and the PL. Figure 6.4 provides an overview of the different implementation layers of the OSSS/MC software stack. Starting at the top, each mixed-criticality subsystem is implemented on an isolated software stack managed by the hypervisor. Within each instance, an OSSS/MC runtime stack implements the application-local scheduling policy through a dedicated RTOS. Together, the hypervisor and the RTOS instances containing the tasks form a hierarchical two-level scheduling configuration. The hypervisor is running on one of the Cortex-A9 dual-core processor and maps the mixed-criticality subsystems exclusively to the other processor core. The measurement infrastructure, described in Section 7.1, is instantiated on a MicroBlaze soft-core in the PL and is equipped with a

dedicated dual-port block RAM accessible to the hypervisor via dedicated drivers.

6.4.1 *Platform Components & Constraints*

The mixed-criticality programming model makes several assumptions about the underlying platform capabilities. First, it is assumed that task computation times can be estimated (upper-bounded) and that subsystems can be executed independently of their mapping relation. While the first requirement imposes restrictions on the expressiveness of the model of computation, as discussed in Section 6.1.1, the second constraint requires platform-level support for *temporal* and *spatial* segregation. As a consequence, the hypervisor implementation needs to provide both functional and temporal isolation techniques for the mixed-criticality subsystems.

Managing the state of mixed-criticality subsystems therefore needs to be handled by an abstraction layer which explicitly controls resource access to the underlying platform. This is implemented by *virtualising* the platform resources and control the access from *guest instances* to the state of the virtualised platform resources. A common technique in hypervisors for implementing this feature is to utilise the processor's privilege modes to distinguish between the supervisor execution which the management instance is running in, and the unprivileged execution mode for the guest instances. As soon as a guest instance accesses a virtualised peripheral, a hardware exception handler allows the underlying hypervisor implementation to catch and emulate the desired access, thereby ensuring control over the physical device. Hardware platforms with support for virtualisation provide additional instructions for natively supporting this concept. Alternatively, these mechanisms can be emulated (with an obvious performance penalty) on platforms with a Memory-Management Unit (MMU) by restricting access on a memory address basis.

Enforcing the independence of mixed-criticality temporal behaviour furthermore requires the use of platform timers which need to be configured according to the application-level time slots. The timers allow the hypervisor management instance to regain control by preempting the processing unit execution once the desired slot duration expires. However, as discussed previously, simply re-assigning the processor resource may not ensure temporal isolation of mixed-criticality subsystems, since the platform state is not necessarily in a neutral state after a context switch to another subsystem. Therefore, it is desired to control the state of shared resources on the platform, specifically the cache behaviour, before executing the next mixed-criticality subsystem slot. Since disabling the cache infrastructure completely is undesirable due to the resulting performance penalty, the underlying implementation platform can instead flush the cache component to avoid temporal dependencies between critical and non-critical subsystems.

The programming model provides different scheduling policies in each mixed-criticality subsystem instance. While the `HI`-critical tasks are scheduled statically, the `LO`-critical subsystem components can be dynamically scheduled to increase the response time for certain use-cases. Therefore, the

target implementation needs to provide a hierarchical scheduling approach, with the slot-based scheduling at the root level and the application-specific scheduling policies at the guest-level.

To implement these requirements, we chose to extend the light-weight, open-source hypervisor Xvisor [64] which provides virtualisation support and manages the system criticality state. The hypervisor has been ported to the Zynq ARM Cortex-A9 dual-core in an SMP configuration with one core dedicated to executing the `HI`- and `LO`-critical vCPU. The second core is used for booting the system and running the hypervisor maintenance console. Note that this decision was taken to eliminate any potential L1 cache intereferences which can occur when switching mixed-criticality contexts on the same core. The overhead of executing the hypervisor on a second core can be mitigated by a more restrictive cache partitioning setup.

Xvisor provides an isolated execution environment for each subsystem by managing a virtual CPU (vCPU) state and its transitions. Furthermore, peripheral access and interrupts can be either configured as pass-through or emulated, making it possible to gain complete control over the resources assigned to the guest instances. The hypervisor implements the requirements stated above by providing guest instances which can be configured, controlled, and executed independently. Further extensions have been implemented to ensure a predictable cache state when switching mixed-criticality subsystem contexts. The Xvisor scheduler, implementing a static time-triggered scheduling policy, has been extended to support the execution semantics of mixed-criticality subsystems as presented in the previous sections.

6.4.2 *Mapping & Scheduling Configuration*

Xvisor provides a configuration infrastructure for design-time parameters which is bundled with the deployed binary in a post-link stage, avoiding the need for recompilation or re-linkage of the target binary executable code when changing the configuration settings. The configuration is written in the *device-tree* specification format [55], a versatile domain-specific language for describing the structure, its features, and the address mappings of (embedded) platforms and their peripherals. There exists a top-level device tree description for the underlying platform used by the hypervisor, and a per-subsystem configuration which contains the configuration options of the virtualised and emulated peripherals. The advantage of these descriptions is the comparatively low overhead when testing different configurations and it further enables providing pre-compiled and linked binaries which can be configured as needed in a later step of the integration process.

We have extended the top-level device tree configuration and the per-subsystem specification to allow configuring the mixed-criticality scheduling policy and the mapping of tasks to subsystems. The scheduling frame configuration for the target software stack as well as settings related to the measurement infrastructure are encoded in the top-level device tree. The device tree description of each subsystem contains the mapping relation $\mu_{\mathcal{T}}$ and VRL runtime properties associated to each instance.

Technically, there are different approaches for refining the model description to a target implementation. One approach is to use code generators to provide input for the target compiler which will then emit the executable binaries representing the specified behaviour and the configured mapping/scheduling policies for each target processing unit. The other approach is to provide a runtime environment on the target and execute the system according to a configuration read at initialization of the target system. The second approach therefore contains a dynamic initialization phase which sets up the execution environment, configures the scheduling policy according to the parameters, and determines which components are scheduled, based on the given mapping information. It is therefore more flexible due to the possibility to modify the configuration without the need to invoke the complete code generation and cross-compile toolchain. The refinement flow implements the latter approach through the use of the device tree configuration mechanism.

The frame configuration F_i is specified in the executable simulation model. As discussed, these properties are extracted at elaboration time. In the refinement step, they are used for configuring the hypervisor implementation stack on the target dynamically at boot time. One of the benefits of this approach is a design-time consistency check of the user-supplied configuration settings, because the settings are specified as API calls into the OSSS/MC implementation of the native, C++-based simulation environment. Due to the use of expressive compile-time type checks for specifying the frame durations as well as the task mapping, the exposure to system misconfigurations are minimised. The elaboration step and underlying assertions will check if the mappings $\mu_{\mathcal{T}}$ and $\mu_{\mathcal{S}}$ are consistent with the specified resource model $\mathcal{R}_i$ and its task list $\mathcal{T}_i$ specified in the frame configuration F_i. Additionally, the model properties are specified with the help of a type-safe compile-time unit library to prevent conversion errors between different units in the configuration layers. As a further simplification of deployment, we provide the same payload binary for both criticalities by creating an identical copy and perform the criticality-dependant configuration during the guest boot process. As a consequence, the integration process is simplified with the low-cost overhead of an extended initialization routine.

6.4.3 *Application Context Management*

The hypervisor guest partitions are implemented with the help of an abstraction layer consisting of vCPU, emulated devices, and interrupt handlers for each guest system. The hypervisor prevents access to the underlying physical peripherals as well as special per-instance processor registers. Upon using these privileged instructions or accessing the (virtual) peripheral address space, the platform raises an exception due to insufficient access rights, and allows the hypervisor implementation to emulate the operation on the virtual representation of the resource.

The hypervisor scheduler has been modified to implement the time-triggered frame scheduling policy FTTS according to the configured dura-

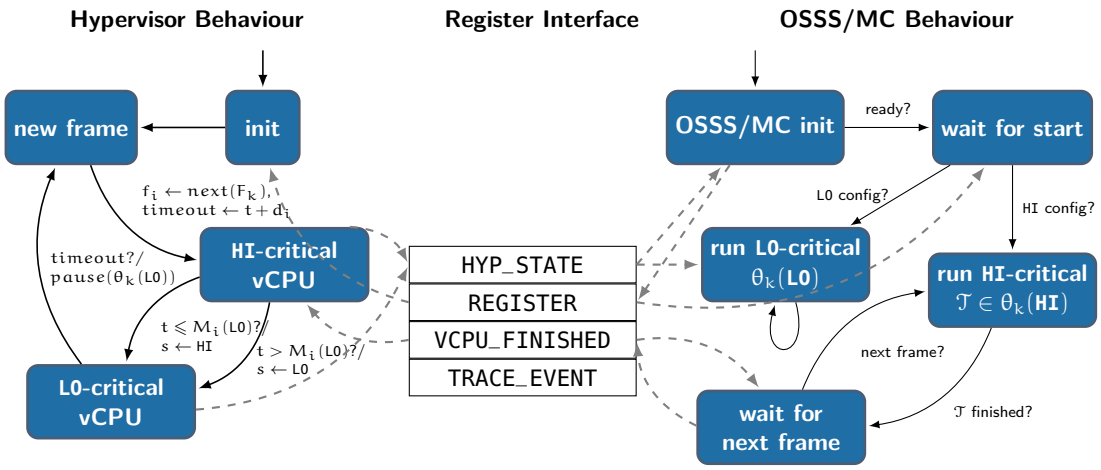


Figure 6.5: Overview of the hypervisor and RTOS execution semantics with communication across the register interface (dashed arrows).

tions δ_i. Each vCPU priority is then chosen to reflect the defined subsystem criticality.

The task scheduling behaviour is implemented within each OSSS/MC runtime instance. Note that the top-level scheduling of guest instances is based on the task execution behaviour inside the guest instances, since the scheduling semantics allow LO-critical subsystems to execute as soon as the tasks of a HI-critical slot have completed their execution. To construct this global scheduling view, the application-local task execution state needs to be communicated to the hypervisor. The hypervisor architecture provides a communication interface in the form of an emulated device, which is represented to the guest instance by a virtual peripheral mapped to its address space. The register interface of the device is accessible via MMIO. This interface was not only used for communicating the scheduling state, but also serves as an interface to communicate task-level timing measurements.

The hypervisor receives payload data through the MMIO interface of the emulated device, such as task start and end events along with their identifiers. Figure 6.5 illustrates the hypervisor and OSSS/MC runtime behaviour across the register interface. Listing 6.4 additionally provides pseudo-code that illustrates the hierarchical scheduling implementation of vCPUs in the hypervisor and tasks in the runtime.

The left part of Figure 6.5 and the pseudo-code depicted in Listing 6.4 illustrate the hypervisor behaviour. Upon booting the system, after all guest instances have been initialised, the hypervisor activates the frame timer and resumes all vCPUs. The HI-critical vCPU executes the task list of the current frame in the defined task order. When finished, it yields control back to the hypervisor. The hypervisor then pauses the HI-critical instance, determines the remaining frame duration, and adjusts L_i according to the execution time duration of the slot and the margin M_i(LO). When the frame duration δ_i elapses, the hypervisor prepares the next frame by resuming all HI-critical vCPUs in the new frame F_i.

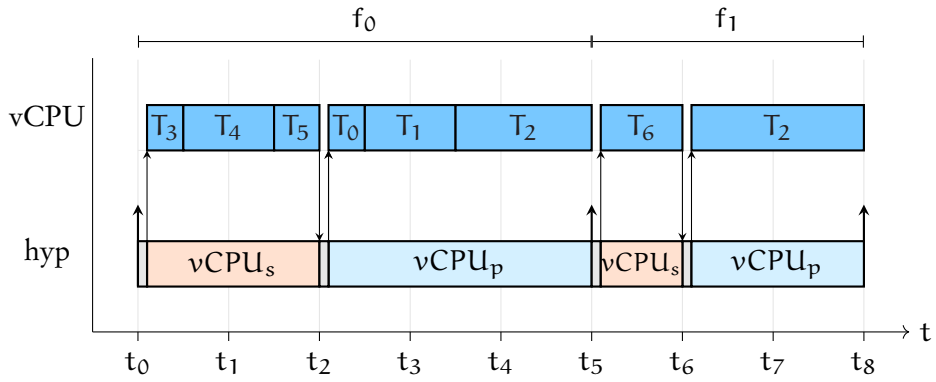


Figure 6.6: Example hypervisor frame and OSSS/MC task execution behaviour of a HI-critical νCPU_s and LO-critical νCPU_p application container.

Algorithm 1 : vCPU scheduling behaviour.

```
FrameConfig ← frame configuration;
wait for all vCPUs to be initialised;
while true do
    NextFrame ← NextFrame + 1  mod size(FrameConfig);
    for c ∈ vCPUList(NextFrame) do
        ResumeVCPU(c);
    end
    WaitForNextFrame(NextFrame);
end
```

Listing 6.4: Pseudo-code illustrating the hypervisor frame scheduling policy behaviour.

The underlying priority-based scheduling policy ensures that the vCPU which is ready and has the highest priority is always chosen to run at the start of a frame. Orchestrating the FTTS schedule is then performed by selectively pausing or resuming the execution of vCPU instances according to their criticality and the state of the execution within a frame.

After the HI-critical vCPU has finished executing its task list $\mathcal{T}_i \in \theta_k(\texttt{HI})$, it will send an event via MMIO to the hypervisor which then determines the remaining frame duration and adjusts L_i according to the execution time duration of the slot and the margin $M_i(\texttt{LO})$. The HI-critical vCPU will be paused such that the LO-critical vCPU is chosen by the priority-based scheduling policy. Note that the HI-critical container will always yield the processor. However, if a LO-critical vCPU does not yield, it will be preempted at the end of a frame such that the HI-critical application can be executed again in the next frame.

Algorithm 2 : vCPU-local task scheduling behaviour.

```
Criticality ← ReadReg(HYP_STATE);
initialise environment;
WriteReg(REGISTER) ; // blocks until first frame starts
if Criticality == LO then
    // execute dynamic LO-critical schedule
    while true do
        t ← GetNextTask;
        crit ← ReadReg(HYP_STATE);
        ResumeTask(t, crit);
    end
else
    // execute HI-critical static schedule
    while true do
        frame ← ReadReg(HYP_STATE);
        for t ∈ T_frame do
            ResumeTask(t);
        end
        wait for all tasks to finish;
        WriteReg(VCPU_FINISHED);
    end
end
```

Listing 6.5: Scheduling policy behaviour of the mixed-critical subsystems implemented as guest instances in the hypervisor.

6.4.4 *Programming Model Implementation*

The OSSS/MC programming model tasks and Shared Objects are implemented on top of FreeRTOS. Structurally, each OSSS/MC task is represented by a FreeRTOS task, while Shared Objects wrap memory which is shared between tasks. They provide the Shared Object implementation which is exposed through interface methods and accessible through matching ports defined in the task. Each FreeRTOS task contains a private execution context that stores the task state across its invocations.

The underlying RTOS scheduling policy has also been modified to implement the execution semantics of the runtime model Π_i presented in Section 6.2.3. The state machine on the right side of Figure 6.5 and the algorithm depicted in Listing 6.5 illustrate the OSSS/MC initialisation and run-time behaviour.

Note that the design decision to provide a configuration-based initialization requires an extended initialisation phase of the RTOS. Figure 6.5 depicts the behaviour upon system initialisation. In the state *OSSS/MC init*, the runtime initializes either the HI- or LO-critical task container, depending on the configuration of the system. To achieve this, the implementation requests information from the system configuration through the HYP_STATE register, which contains the current global system state in terms of the cur-

rently active frame, the vCPU-local criticality, and the dynamic criticality state. With this information, the per-application scheduling policy configuration Π_i can initialize the system corresponding to the criticality of the subsystem instance.

Once all guest instances have initialised the underlying RTOS and are ready to execute the given subsystem schedule, they reach the *wait for start* state, where they register themselves to the hypervisor via the `REGISTER` offset. A barrier has been implemented in the hypervisor to synchronise the next steps in the subsystems. After both the `HI`- and `LO`-critical subsystems have reached the barrier, the hypervisor frame scheduler starts scheduling the first frame and switches the context to the guest instance with the highest priority in this frame.

To implement the static scheduling order defined in the runtime model, the `HI`-critical instance instantiates a high-priority FreeRTOS task. This task coordinates the static, frame-based scheduling order of the task instances $\mathcal{T}_i \in \theta_i(\texttt{HI})$ executed in the frame (state *run HI-critical*). After the tasks have been executed, the OSSS/MC runtime yields the control back to the hypervisor by switching to the state *wait for next frame* and accessing the `VCPU_FINISHED` register. When a new frame starts and the `HI`-critical vCPU resumes in state *wait*, it reads the global system state register again and executes the tasks $\mathcal{T}_i$ mapped to the frame determined by the updated content of the register value.

The dynamic schedule of the `LO`-critical subsystem is implemented using the scheduling policy available in the RTOS. We have chosen a preemptive priority-based scheduling policy that executes the mapped tasks on a best-effort basis to maximise their throughput. `LO`-critical tasks are therefore executed according to their dynamic scheduling policy $\Pi_i(\texttt{LO})$ and will be preempted by the expired frame timer of the hypervisor after the frame duration has elapsed.

The `LO`-critical subsystem implementation reads the current frame and system criticality f_i after each task invocation by accessing the `HYP_STATE` register. This information is used to update the dynamic scenario property L_i. Note that `LO`-critical tasks can be preempted at the end of a frame in which case no update to the scenario state will be performed when resuming the task in the next frame. For future activities regarding resource availability feedback on the system criticality state, it needs to be considered that the criticality state may only be updated on a task boundaries which may not necessarily be aligned to the slot execution windows. Section 10.2 contains a more detailed discussion of this extension.

6.5 SUMMARY

This chapter presented a design flow starting from a functional behaviour of software components which have been categorised into `LO`- and `HI`-critical subsystems. First, a specification model has been proposed to capture the unique mixed-critical properties for the functions which should be integrated on a common platform. The functional behaviour as well as their extra-functional timing properties are specified in the AL and partitioned

into tasks for executing the computation, and Shared Objects representing their communication channels.

Next, an implementation of the programming model as an executable specification based on SystemC has been presented, allowing the designer to functionally test the behaviour and ensure proper functional partitioning. The executable specification allows integrating C/C++ based descriptions of functional behaviour, which can be either manually specified or extracted from high-level system models such as Matlab/Simulink through the use of a code generator.

We have furthermore proposed a resource model for a platform executing mixed-criticality behaviour. The VRL contains a runtime model to which the AL tasks and Shared Objects are mapped. The runtime model then performs a hierarchical scheduling according to the semantics of a mixed-criticality scheduling policy. Coupled with user-defined timing annotations, the mapping of tasks and Shared Object to the runtime can be evaluated in a SystemC-based environment, allowing the designer to gather initial performance metrics of the partitioning and mapping decisions.

Finally, we have provided a systematic way of implementing the proposed components of the programming model. The proposed refinement flow aids the designer in implementing the functional behaviour and make use of the identified techniques for temporal and spatial isolation on the target MPSoC.

7 PERFORMANCE MODELLING

The previous chapter presented the OSSS/MC programming model target components enabling a systematic implementation of the executable specification model on an MPSoC target platform. This section describes the modelling approach integrated into the design flow. The proposed measurement infrastructure serves as the data source for evaluating the temporal behaviour of the application model components. The first section introduces the measurement infrastructure implementation on the target and its integration into the hypervisor implementation and the platform components.

Next, a performance modelling approach is presented which allows to perform an analytical estimation of the integration impact on the `LO`-critical subsystem in terms of its timing behaviour. Finally, we propose the construction of application proxies, which are based on the gathered timing behaviour from the measurement infrastructure and mimic the resource consumption of `HI`-critical subsystems while omitting the execution of their functional behaviour on the target platform.

7.1 MEASUREMENT INFRASTRUCTURE ARCHITECTURE

The measurement infrastructure captures two different data sources and thus consists of two components. The first component collects execution time durations of `HI`-critical slots by constructing a *dynamic histogram* from a series of measurements on the target. A second module extracts average durations of task, slot, and frame execution time in the system. Additionally, the module collects average performance-related metrics on a per-slot basis from the hardware platform. This section describes both components, their configuration and run-time interfaces. It furthermore discusses the data post-processing along with the performance model and application proxy construction based on the collected data.

The OSSS/MC measurement infrastructure is depicted in Figure 7.1. Its target software implementation consists of two independent hypervisor modules which can be configured and used separately. The `osssmc_dist` module gathers slot execution time durations and constructs a distribution as a histogram from the collected data. The second module, `osssmc_trace`, collects durations and events which can be annotated across the full software stack, ranging from task-level execution times over bare-metal hypervisor events to hardware performance counter metrics. Both modules use the existing hypervisor configuration infrastructure, such that each one can be mapped independently to available memory address ranges and their parameters can be modified without the need for recompiling the hypervisor binary.

Interference-free measurement data extraction has been implemented by a dedicated MicroBlaze soft-core accessing a dual-ported block RAM which

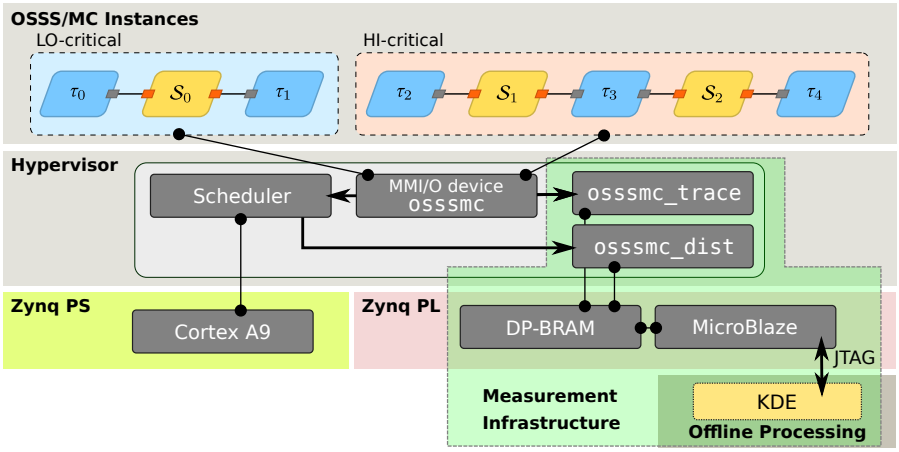


Figure 7.1: Overview of the HW/SW measurement infrastructure components in the hypervisor and programmable logic.

stores the data gathered by the `osssmc_trace` and `osssmc_dist` modules running on the ARM core. Note that the MicroBlaze instance is only needed for the JTAG access and does not execute any functions. Since the data extraction may alter the temporal behaviour, platforms which do not support custom logic can alternatively stop the measurement setup before accessing the data in the main memory.

Internally, each module provides a method-based interface for tracking the distribution-related durations and events. Apart from the event type, the interface provides means for gathering meta-data such as performance counter values or vCPU identifiers, which are then stored in the mapped BRAM along with the event type. The `osssmc_trace` module interface is additionally accessible via MMI/O by an emulated device `osssmc` mapped to each OSSS/MC instance address space. A simple driver integrated into the OSSS/MC runtime forwards annotated events in the task implementation to the hypervisor. The `osssmc_dist` module implements the functionality for storing the hypervisor-managed slot duration of a given frame.

7.1.1 *Online Pre-Processing*

The `osssmc_trace` module measures two types of events: the occurrence of absolute events and the duration between two events. The latter can represent the start and end of a state, such as the duration of a running task or slot. For this duration-based measurement, the module continually updates the minimum, maximum, and running average durations. A weighted moving average across the event durations increases the stability of the determined average value. The weighted average is calculated as $a(t) = \frac{1}{8}x(t)+\frac{7}{8}x(t-1)$, such that new values have an impact of 12.50 % on the moving average.

The module supports tracing the vCPU duration, task execution times, frame, slot, and hyper-period start/finish events as well as platform performance counter values. Each absolute event and event duration has a predefined memory location in the external memory buffer which can be specified when configuring the module. Furthermore, the events are annotated

in the provided target implementation of the AL components, such that it is not required for the designer to insert them manually.

The execution time of a slot is recorded by an on-line dynamic histogram mechanism to gather as many data points as possible on the target. The histogram captures the number of execution time observations of a particular slot, which can be selected by a measurement infrastructure configuration setting. A histogram typically is parametrised with a *bin size* which defines the interval in which observed values are grouped. The bin size can also be interpreted as the resolution of the histogram data. To maximise the histogram resolution, the chosen bin size and data range therefore are a direct property of the available memory space for storing the histogram (which defines the total number of bins).

To increase the data resolution of the measurement infrastructure, the bin size and the chosen range needs to match the expected range of values which should be observed. The measurement infrastructure therefore dynamically adjusts the bin size and the range. This *dynamic histogram* avoids the need for an initial measurement and a per-deployment configuration of the histogram bin size and data range settings. It ensures that a minimum bin size and appropriate data range is chosen which ultimately results in a maximum histogram resolution for the available memory to which the module is mapped to.

Listing 7.1 illustrates the adjustment. Initially, the bin size and data range are set to provide the highest resolution possible in the statically configured memory range. When a `HI`-critical slot finishes, the module checks the current bin size, minimum value, and range of the histogram to determine whether the new value fits into the configured value range. The histogram bin size calculation uses the maximum and minimum data points observed so far as well as the available memory size to determine the new minimum bin size. When a new duration is stored in the histogram, the mechanism checks if the data point can be mapped to the current histogram configuration, consisting of the minimum duration and the range covered by the histogram. If the new data point is not within the current range of the data points, the histogram needs to be extended to allow storing the new data point. After readjusting the parameters, all previously stored values are also converted to the new histogram bins.

Note that the distribution range will never be decreased; it will start with the highest possible resolution (constrained by the available memory region) which consequentially provides only a small range of values to be captured. Next, the mechanism then will gradually increase the range to match the observed execution time distribution range. Along with the histogram bin data, the necessary metadata to reconstruct the histogram after extracting the BRAM contents are also stored in memory.

7.1.2 *Data Extraction & Post-Processing*

The measured event durations and histogram data from the `osssmc_trace` and `osssmc_dist` modules are stored in a block RAM in the programmable logic part of the Zynq. To extract the data from the memory without gen-

Algorithm 3 : Dynamic histogram resizing.

```
Data : the measured duration and the current Histogram
Result : the updated Histogram
// get the current histogram properties
min ← Histogram(min) ;
max ← min + Histogram(range) ;

// Adjust bounds of current histogram properties
if duration < min then min ← max(duration − SafetyMargin, 0) ;
if duration > max then max ← duration + SafetyMargin ;

range ← max − min ;
// check if the data layout needs to be adjusted
if min ≠ Histogram(min) ∨ range ≠ Histogram(range) then
    // keep the old data
    Histogram' ← Histogram ;

    // update histogram properties
    Histogram(min) ← min ;
    Histogram(range) ← range ;
    Histogram(binsize) ← 1 + (range−1)/AvailBins ;

    // copy old data to new layout
    CopyHist(Histogram', Histogram) ;
end
return Histogram ;
```

Listing 7.1: Pseudo-code illustrating the dynamic histogram resizing.

erating interference on the active measurements, we use a MicroBlaze soft-core that is accessed via JTAG and instructed to read a given memory location over JTAG into a local file on the host machine. In this setup, the MicroBlaze core is only used to enable JTAG-based access to the memory; it does not execute any functions. Note that this does not restrict the use of our measurement infrastructure to platforms with programmable user-logic, since the online-extraction only served as a convenience method for extracting the data. An equivalent process can be implemented by stopping the measurements on the platform before downloading them through an external connection.

After gathering the `osssmc_trace` data, they are further processed in a python-based setup where the raw memory image extracted from the board is parsed according to the event memory layout specified in the `osssmc_module` configuration. Each duration contains the average, min, and max value and is exported as a CSV file for further inspection.

7.2 DATA-DRIVEN PERFORMANCE MODELLING

This section describes the performance model construction based on the measurements presented in the previous section. The extracted measure-

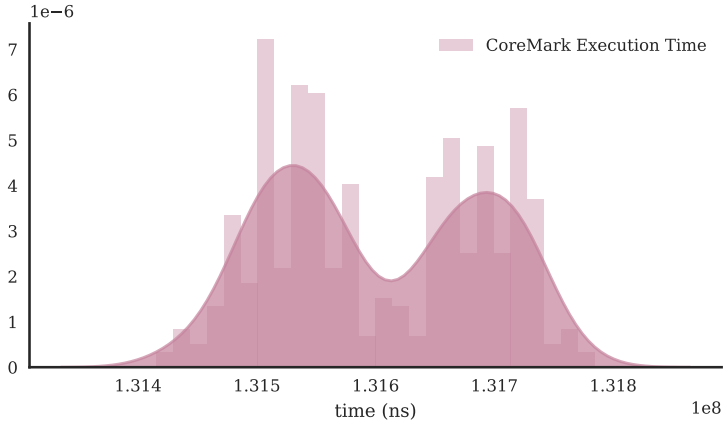


Figure 7.2: Probability Density Function constructed from the measurements the CoreMark benchmark.

ment results of the `osssmc_dist` distribution are used to construct a performance model to estimate the utilisation impact on the `LO`-critical software subsystem resulting from the overall frame and slot management overhead, the scheduling configuration, and the temporal behaviour of the `HI`-critical slot execution.

7.2.1 *Assumptions*

The performance modelling approach uses the measured histogram data and constructs a statistical model by determining the Probability Density Function (PDF) of the measured data. Figure 7.2 provides an example measurement result of the `osssmc_dist` module and the constructed PDF. The underlying necessary assumption for constructing a PDF is that the observed execution time durations follow an independent and identically distributed random process $X = (x_1, x_2, \ldots, x_n)$. The data being *independent* in our context means that the execution time does not depend on any previous measurements. This is true due to our assumption on the temporal segregation enforcement of `HI`- and `LO`-critical slots on the platform. The data is also assumed to be *identically* distributed. In our context, this means that we assume that we manage the platform state in a way that each measures slot execution time duration is modelled by the same distribution.

These assumptions allow us to determine the PDF $f(x)$ by calculating the Kernel Density Estimation (KDE) $\hat{f}(x)$ from the histogram data. The KDE is defined as

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

and can be parametrised with the *kernel* K (a distribution) and a *bandwidth* h. Intuitively, each measured data point x_i represents a kernel as the point's own probability distribution. As a consequence, the quality of the resulting PDF is determined by the histogram resolution. The KDE sums up the weighted contribution of all kernels for a particular probability of point x

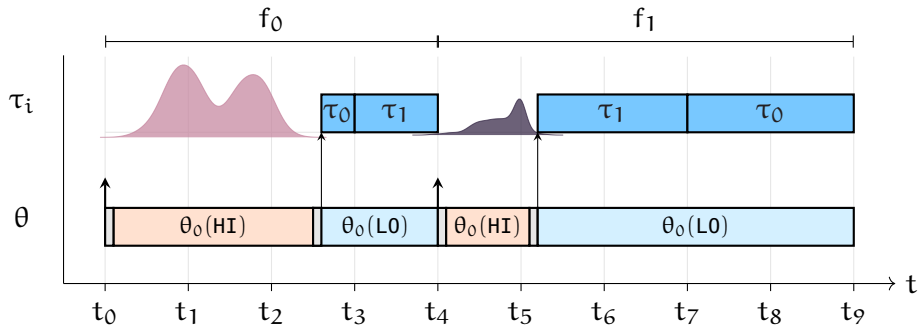


Figure 7.3: Illustration of HI-critical slot execution time distributions in the mixed-criticality scheduling policy implementation. Each distribution models the overall probability of a single slot duration.

according to the estimated PDF. We have chosen the Gaussian kernel and the data-dependant bandwidth taken from the experiment histogram results.

7.2.2 *Model Construction*

The PDF represents a model of the timing behaviour of a set of tasks executing in a HI-critical slot. Assuming that the measurements provide a reasonable coverage of the timing behaviours of the subsystem in the slot, we can therefore use the timing behaviour model to analytically estimate the integration cost in terms of performance impact to LO-critical subsystems, since the slot behaviour of the HI-critical subsystems directly determines the available resources of any LO-critical subsystem due to the dynamic mixed-criticality scheduling approaches. Figure 7.3 illustrates the general idea of the modelling approach.

The PDF represents the per-frame utilisation distribution by modelling the HI-critical execution time of a single slot. Extending the approach for all frames then results in a representation of the overall system utilisation caused by the HI-critical subsystem. We have implemented the histogram data post-processing and visualisation with the python `seaborn` [78] and `numpy` [75] libraries. Constructing the PDF has been achieved with the KDE implementation provided by the `sklearn` [65] library.

Definition 11 (LO-critical application performance model)**.** *Given a frame configuration* F *and its hyperperiod* H_F*, we define* $U_{f_i,\mathcal{O}}$ *to represent the per-frame utilisation caused by the HI-critical slot execution time of* $\mathcal{T}_i$ *modelled with the PDF* $d_{f_i}(x)$*, its expected value* $E[d_{f_i}]$ *to realise the distribution, and a context switch overhead* $\mathcal{O}$ *for each slot:*

$$U_{f_i,\mathcal{O}} = \frac{E[d_{f_i}] + 2\mathcal{O}}{\delta_i}$$

The utilisation over a complete frame set $U_{F,\mathcal{O}}$ *is defined as*

$$U_{F,\mathcal{O}} = \frac{1}{H_F} \sum_{f_i \in F} \delta_i U_{f_i} = \frac{1}{H_F} \left(2\mathcal{O}|F| + \sum_{f_i \in F} E[d_{f_i}] \right)$$

Determining the context switching overhead $\mathcal{O}$ *and* d_{f_i} *for each* $f_i \in F$ *denoted as the set* d_F *allows us to instantiate the performance model for our application configuration and platform properties. After measuring the* LO*-critical execution time* C_k *of a task in isolation, we can then determine the expected response time* R_k *on the platform, given the system scheduling configuration and the resource consumption models of the* HI*-critical application:*

$$\begin{aligned} R_k &= C_k \frac{1}{1 - U_{F,\mathcal{O}}} \\ &= C_k \frac{H_F}{H_F + 2\mathcal{O}|F| - \sum_{f_i \in F} E[d_{f_i}]} \end{aligned}$$

□

Note that depending on the complexity of the scheduling configuration, multiple measurement runs may have to be performed to determine d_{f_i} for all $f_i \in F$ since the measurement infrastructure can only be configured to capture the HI-critical slot execution time durations of a particular slot in a hyperperiod. Each measurement configuration therefore provides the HI-critical execution time distribution of one particular frame. However, if the underlying platform can provide sufficient memory, this approach can be easily extended to capture histograms of multiple slots at the same time.

The following example shows how the performance model can be applied after performing measurements of the slot duration of a HI-critical subsystem and using these with the isolated response-time measurements of a LO-critical subsystem which should be integrated on a mixed-criticality platform.

Example 4 (Performance model example with the `compress` benchmark)**.** *In this example we define a VRL configuration with a scheduling frame* $f_0 = (\mathcal{T}_0, \delta_0, M_0)$ *where* $\mathcal{T}_0$ *contains a task running for an expected execution time of* $E[d_{f_0}] = 34\,280.76\,\mu s$ *and the frame setup has been defined as* $\delta_0 = 171\,048\,\mu s$ *with* $M_0(LO) = 34\,297.56\,\mu s$ *and* $M_0(HI) = 102\,629\,\mu s$ *(configuration taken from the* `compress` *benchmark of the evaluation). There is only one frame, such that* $H_F = f_0$ *and the determined platform overhead is* $\mathcal{O} = 20.52\,\mu s$*. Assuming that the isolated measurements of a video frame encoder determined an average end-to-end execution time of* $C_0 = 4\,331\,585.41\,\mu s$*, we can now estimate the expected response time* R_0 *in an integrated environment:*

$$\begin{aligned} R_0 &= C_0 \frac{H_F}{H_F + 2\mathcal{O}|F| - E[d_{f_0}]} \\ &= 4\,331\,585.41\,\mu s \cdot \frac{171\,048\,\mu s}{171\,048\,\mu s + 41.04\,\mu s - 34\,280.76\,\mu s} \\ &= 5\,415\,792.58\,\mu s \end{aligned}$$

Under the given scheduling configuration and HI*-critical execution time distribution, the model therefore estimates the video frame encoder end-to-end execution time of* $\approx 5.42\,s$.

The performance model uses the expected value of the distribution to determine the average execution time estimation analytically. An alternative

approach is to incorporate the generated PDF into the simulation model presented in Section 6.3. This extension is further discussed in the outlook in Section 10.2.

7.3 APPLICATION PROXY

The PDF model constructed in the previous chapter allows us to analyse the expected performance behaviour of the LO-critical application, given an initial, isolated end-to-end execution time of the LO-critical application. This section describes the construction of a dynamic execution time model based on the determined PDF of the HI-critical subsystem. The idea is to use the model data on the target to mimic the execution time behaviour of HI-critical slots. Such a timing model on the platform can provide an integration environment for LO-critical subsystems due to the combination of the HI-critical timing behaviour with the dynamic mixed-criticality scheduling policy. As a consequence, LO-critical subsystems can be integrated on the platform without needing to integrate the functional behaviour of the HI-critical subsystem. For each frame f_i, instead of executing the mapped HI-critical tasks, the underlying scheduling policy consumes the duration d_{f_i} which is determined by the PDF model.

7.3.1 *Target Requirements*

To replay the modelled execution time of a slot on the target, the implementation needs to provide a mechanism for generating the slot duration intervals as defined in the constructed PDF. There are multiple ways to achieve this. The first option is to implement the PDF on the target. This option requires representing the PDF properties and its semantics in the target implementation. Therefore, all measurement points used to construct the PDF as well as the KDE functionality, which generated the distribution, need to be available on the target. Recall that the KDE accumulates the Gaussian distributions for each measurement point. This operation may therefore infeasible to be performed on the limited computing resources of an embedded target. While the model generation may be performed at the target initialisation phase, extracting an execution time value for the HI-critical slot at the start of each frame might induce more timing overhead than the actual execution time that should be consumed. A possible mitigation for this effect (depending on the platform capabilities) is the use of a special hardware random number generator [1] for reducing the software overhead when generating the execution time duration samples from the model.

An alternative way to implement the model on the target is to precompute a sequence of execution time samples on the host computer and replay them on the target platform. To achieve this, the target platform only needs to provide a timer which can be set to intervals in the resolution of the measured slot durations. Then, the timer can be configured at the start of each frame to produce a timeout event after the duration of the next execution time sample of the pre-computed sequence.

To account for the potential context switching overhead of the hypervisor layer, the input duration given to the timer has to be calibrated such that the desired execution time duration is produced. This implies determining the context switching overhead and subtracting it from the sample before configuring the timer. This can be achieved by measuring the application proxy execution time durations on the platform and compare them with the specified sample durations. The resulting offset then needs to be considered when setting up the timeouts. Note that this offset is independent of the subsystem behaviour and solely depends on the platform and the overhead of the timer and hypervisor infrastructure.

As a result of the discussion above, we have decided to pre-compute the execution times to avoid inducing any timing uncertainties and delays which can impact the timing resolution of the application proxy by a software implementation of the PDF model on the target processor itself. Although the approach is therefore limited by the sequence length of the application proxy samples due to memory requirements, we were able to successfully run configurations that provided execution time durations for our 10 min experiments, where some slots only lasted for an average of 320 µs. Even in that case, the total memory footprint is less than 21 MB. However, if the storage capacity is a limiting factor, the durations can also be repeatedly executed, as we will discuss in the in Section 9.4. The next sections describe the implementation of the sample pre-computation and replay approach for the HI-critical slot execution time distribution.

7.3.2 *Implementation*

As shown in the previous section, the PDF model d_{f_i} represents the distribution of the HI-critical slot execution time observed in the frame f_i. Implementing this model on the target consists of two steps. The first part is to generate a sequence of n samples $(\sigma_1, \ldots, \sigma_n)_i$ for each frame $f_i \in F$. A sample in the sequence is a duration according to the determined slot distribution d_{f_i} and therefore represents a possible execution time duration of a HI-critical slot. As discussed, the sequence is constructed at design time to reduce the complexity of the target implementation needed to mimic the resource usage behaviour. The next step is to iterate through the sample list at run-time and configure the timer infrastructure accordingly.

The hypervisor implementation has therefore been extended with a new execution mode to support replaying the sample sequence. The new behaviour is illustrated in Listing 7.2. When the mode is configured, the hypervisor implementation executes each frame schedule f_k as usual but skips the HI-critical subsystem slot execution. Instead, it configures a timer to wait for the specified sample duration σ_i, thereby effectively replacing the functional behaviour of the HI-critical subsystem with the resource consumption behaviour based on the samples extracted from the PDF model.

Each frame invocation of f_k results in selecting the next sample σ_{i+1} of the pre-computed sequence $(\sigma_1, \ldots, \sigma_n)_k$ and instead of executing the HI-critical subsystem, a timer is configured to expire after the specified duration

```
FrameConfig ← frame configuration;
wait for all vCPUs to be initialised;
while true do
    NextFrame ← NextFrame + 1 mod size(FrameConfig);
    for c ∈ vCPUList(NextFrame) do
        if CriticalityOf(c) == HI then
            WaitForTimeout(NextSample(NextFrame));
        else
            ResumeVCPU(c);
        end
    end
    WaitForNextFrame(NextFrame);
end
```

Listing 7.2: Pseudo-code illustrating the hypervisor frame scheduling policy behaviour under the HI-critical application proxy behaviour.

has elapsed. The LO-critical slot is executed for the remainder of the frame duration.

7.4 SUMMARY

This chapter presented the measurement infrastructure integrated with the programming model implementation on a hypervisor-managed MPSoC platform. The measurement infrastructure contains two modules. The `osssmc_event` module provides measuring one-shot events and durations, while the `osssmc_dist` module measures the slot execution times. It also contains a dynamic approach for storing the measurements in a histogram while ensuring the maximum possible time resolution at the given memory constraints. This infrastructure forms the basis of the modelling approach proposed in this thesis.

The measurement infrastructure is subsequently used to extract execution time duration for constructing an analytical model of the HI-critical subsystem timing behaviour. The purpose of this model is to predict the integration impact of LO-critical subsystems in terms of its timing behaviour. In combination with an (isolated) timing estimation of a LO-critical subsystem, the model can estimate the impact on the end-to-end execution time caused by the HI-critical subsystem resource consumption in combination with the mixed-criticality scheduling policy on the platform.

Based on the extracted PDF, a method for implementing the generated HI-critical model on the target platform has been proposed. To achieve this, a list of *samples* is extracted from the constructed model and a modification of the implementation replays the scheduling behaviour with the samples. This *application proxy* mimics the HI-critical computational resource consumption behaviour. As a result, a LO-critical subsystem can be integrated into the mixed-criticality target platform and the dynamic mixed-criticality

scheduling policy artefacts can be imitated without requiring a functional implementation of the HI-critical subsystem.

Part III

RESULTS

8 INTEGRATION FLOW EVALUATION

The following evaluation sections are structured along the contributions of this thesis. The first section provides a definition of a mixed-criticality system with the components introduced in Chapter 6. In the next section, their execution time characteristics are measured with the measurement infrastructure in order to gather initial timing estimates of the software components. The results are then used to derive a feasible setup for the dynamic mixed-criticality scheduling policy.

After configuring the scheduling policy, the evaluation in Section 8.3 examines the performance impact of the segregation approaches in the implementation by considering the overhead of the subsystem context as well as the platform component state management. In particular, the evaluation focusses on the hypervisor context switch timing penalty and the impact of cache flushes, both regarding their timing penalty on the subsystems, and their contribution in mitigating application-level interferences. We further demonstrate the potential gains of the dynamic mixed-criticality scheduling policy for `LO`-critical subsystems due to its adaption to the resource consumption behaviour of `HI`-critical subsystems.

The results of the segregation overhead analysis then allows us to instantiate the proposed performance model and compare its timing estimates with the target measurements. Chapter 9 then further considers the accuracy of the application proxies by evaluating their timing behaviour on the target and comparing it to the original `HI`-critical subsystem timing behaviour. Finally, we demonstrate how a `LO`-critical subsystems can be refined and validated on the target without requiring the implementation of the `HI`-critical subsystem. The restrictions of the application proxy implementation regarding their sample size are considered by providing an estimate of the minimum samples required for achieving a given behavioural accuracy on the target.

8.1 SETUP

This section describes the evaluation setup using the system specification models introduced in Chapter 6. The overall mixed-criticality system consists of a `LO`-critical and a `HI`-critical subsystem. The `HI`-critical subsystem features a benchmark task executing the *Märladalen* WCET analysis tool benchmark suite [36] and the *CoreMark* benchmark [23]. These benchmarks provide different synthetic workloads capturing complex memory- and control-flow workload scenarios for `HI`-critical applications and allow exploring the execution time distribution behaviour on the target platform. In contrast, the `LO`-critical subsystem contains a task and a Shared Object dedicated to executing a video encoding algorithm. This section first defines

the overall system configuration and then details the task properties of each subsystem.

The system configuration $\mathbb{C}_b$ consists of the task set $\mathcal{T}_b = \{\tau_b, \tau_e\}$, the Shared Objects $\mathcal{S}_b = \{\mathcal{S}_0\}$, the binding relation $\mathcal{B}$, and the criticality levels $\mathcal{L}_b = \{\texttt{LO}, \texttt{HI}\}$. Each system configuration $\mathbb{C}_b$ features one benchmark b as the HI-critical task. *Instantiating* the system configuration for a benchmark b_i results in a specific instance of the system where the HI-critical task executes the behaviour (and exhibits the temporal properties) of benchmark b_i.

Each benchmark is wrapped by an OSSS/MC task executing the benchmark 1000 times per invocation. For each benchmark b, the task τ_b consists of the properties $\left(T_b, \pi_b, \ell_b, \vec{\xi}_b\right)$ with

- the period T_b,
- the set of ports $\pi_b = \emptyset$,
- the criticality $\ell_b = \texttt{HI}$, and
- the execution trace $\vec{\xi}_b = ((\emptyset, t_b)^{1000})$, representing the execution of the benchmark behaviour with duration t_b for 1000 times.

Note that T_b is typically specified in the timing requirements of the behaviour executed in the task context and can therefore be annotated to tasks of the programming model. This property can be used by external analysis tools along with the task timing requirements to determine a feasible scheduling configuration. In the first step of the evaluation, we determine the configuration empirically based on the overall slot duration and therefore do not require specifying the period in the task model. A possible extension of the design flow is to incorporate scheduling analysis tools which check the consistency of the specification requirements and the given scheduling analysis.

The LO-critical subsystem consists of a performance-critical function representing a video encoder. In the chosen evaluation setup, it contains an MPEG4 video processing algorithm embedded in a task which encodes one video second (24 frames) per iteration on a synthetic byte stream read from a Shared Object. The video encoding task τ_e thus consists of

- the period $T_e = 0$,
- the set of ports $\pi_e = \mathcal{I}_0$,
- the criticality $\ell_e = \texttt{LO}$, and
- the execution trace $\vec{\xi}_e = ((\pi_e, t_{e0}), (\emptyset, t_{e1}))$, representing the read from the Shared Object using the port π_e with duration t_{e0} and the encoding operation with the duration t_{e1}.

The period T_e is defined as 0 since the encoding should be executed whenever possible to maximise the overall frame throughput. Therefore, the task restarts as soon as it has finished encoding the frame.

The Shared Object $\mathcal{S}_0$ containing the synthetic frame data is defined as follows:

- Σ_0 contains the synthetic frame data,
- $M_0 = \{\texttt{getFrame}\}$ with $\texttt{getFrame} : \Sigma_0 \rightarrow \Sigma_0$ is the method for retrieving the frame data, and
- $\mathcal{I}_0 = \{M_0\}$ denotes the methods exposed via the Shared Object's interface, marking the method $\texttt{getFrame}$ accessible through task ports bound to the Shared Object.

The port binding relation is defined as $\mathcal{B} = (\pi_e, \mathcal{I}_0)$. Since only the LO-critical task port is bound to the Shared Object, its *ceiling criticality* is $\texttt{ceil}(\mathcal{S}_0) = \texttt{LO}$ and the task can access all shared object interface methods, regardless of whether they modify the Shared Object state or not.

The *LO-critical subsystem* therefore consists of the task τ_e and the Shared Object $\mathcal{S}_0$, while the *HI-critical subsystem* contains the task τ_b (for each instance of b). The remainder of the evaluation considers the effects on scheduling, segregation techniques, and performance modelling with the set of benchmarks by *instantiating* the system configuration $\mathbb{C}_b$ for each benchmark b.

8.1.1 *Frame & Slot Configuration*

This section considers deriving the VRL parameters of each system instantiation $\mathbb{C}_b$. Throughout the following evaluation, the overall scheduling policy structure consists of one frame in which the task of each subsystem is executed. Therefore, the scheduling configuration is defined as $F_b = \langle f_b \rangle$ with $f_b = (\mathcal{T}_b, \delta_b, M_b)$. For each benchmark b, the parameters of this configuration will be empirically estimated in this section.

To simplify the process of determining the VRL scheduling properties for each benchmark, the calibration measurements were performed by configuring the hypervisor scheduling frame with a duration of 500 ms to allow all benchmarks b to finish their execution loop within one scheduling frame. The initial scheduling configuration $F_{init,b}$ thus consists of one scheduling frame $f_{init,b} = (\mathcal{T}_b, \delta_{init} M_{init})$ with the properties:

- $\mathcal{T}_b = \{\tau_b\}$
- $\delta_{init} = 500\,\text{ms}$
- $M_{init} = \{500\,\text{ms}, 500\,\text{ms}\}$

For each benchmark b, we measured the execution time durations of the slot executing τ_b using the proposed measurement infrastructure in the hypervisor implementation. Each benchmark behaviour b was executed on the target board as the task τ_b, as specified in the system configuration, for a total duration of 600 s. The resulting slot execution time distribution allows configuring the scheduling policy of F_b by deriving appropriate values for $M_b(\texttt{LO})$, $M_b(\texttt{HI})$ and the frame lengths δ_b, each based on the observed measurement results of benchmark b.

Table 8.1: Average execution time $C_{mean}(\mathcal{T}_b)$ and frame configuration for all benchmarks based on the maximum observed execution time C_{LO} (in µs).

τ_b	$C_{mean}(\tau_b)$	$C_{LO}(\tau_b)$	$C_{HI}(\tau_b)$	δ_b
compress	34 280.76	34 297.56	102 629	171 048
coremark	131 602.82	131 784.56	395 353	658 922
cover	10 734.97	10 749.85	32 170	53 617
crc	5983.31	6029.76	18 115	30 192
expint	348.03	369.16	1054	1757
fdct	11 793.76	11 822.20	35 466	59 110
fft1	24 534.60	24 628.85	77 129	128 549
insertsort	4902.51	4948.62	14 872	24 786
lcdnum	957.71	1010.81	3032	5054
ludcmp	49 523.89	49 644.44	161 564	269 273
qsort-exam	16 935.59	16 955.40	50 839	84 733
qurt	23 610.31	23 749.87	70 801	118 002
statemate	18 801.09	18 818.83	130 261	217 101
ud	9511.25	9624.76	161 564	269 273

The measurement-based approach provides the estimation method C_{LO}, which we use as the mechanism for determining the maximum observed execution time $C_{LO}(\tau_b)$ for each benchmark b. Following contemporary industrial and academic practices for determining task execution times on complex HW/SW platforms, we further approximate the WCET upper bound by multiplying the maximum observed execution time by a factor of three [29]. As a consequence, we define $C_{HI}(\tau_b) = 3C_{LO}(\tau_b)$ to represent the upper bounded execution time determined by the C_{HI} estimation method.

Based on these execution time durations, we define the per-benchmark scheduling configurations with a HI-critical slot duration of $M_b(\mathtt{HI}) = 3C_{LO}(\tau_b) = C_{HI}(\tau_b)$ and an overall frame length of $\delta_b = 5C_{LO}(\tau_b)$ to provide some leftover computation time for the LO-critical subsystem. As a result, with the HI-critical benchmark consuming at most $M_b(\mathtt{HI})$ processor time, the LO-critical subsystem should theoretically be able to consume at least $\frac{\delta_b - M_b(\mathtt{HI})}{\delta_b} = 40\,\%$ of the processor time within one frame. However, due to the dynamic mixed-criticality scheduling policy, the actual processor time depends on the HI-critical execution time behaviour, as discussed in the previous sections.

Table 8.1 lists the average execution time (denoted as C_{mean}), the maximum observed execution time C_{LO} on the platform, the derived value for C_{HI} as well as the frame duration settings δ_b of each benchmark b. Based on the results, $M_b(\mathtt{LO})$ is set to C_{LO} and $M_b(\mathtt{HI})$ is set to C_{HI}. This concludes the per-benchmark frame scheduling settings.

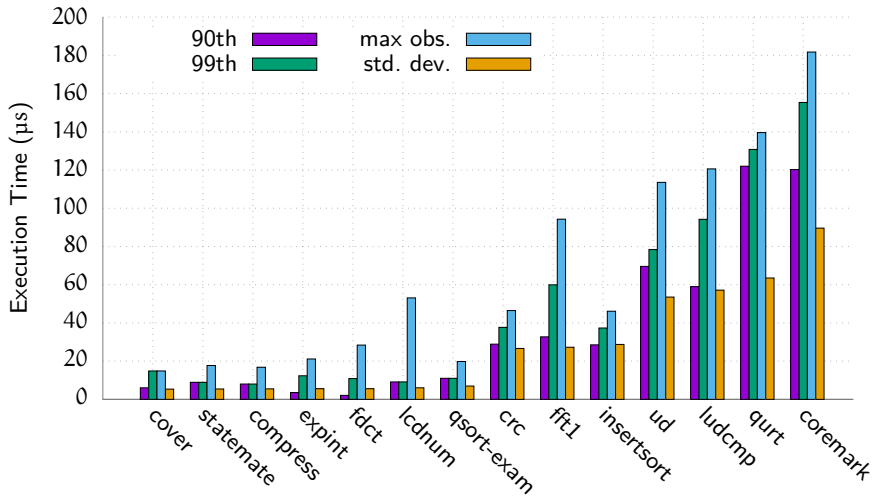


Figure 8.1: Absolute execution time difference at the 90th, 99th percentiles, and maximum observed execution time, sorted by standard deviation (baseline: average execution time).

8.1.2 *Benchmark Timing Behaviour*

To compare the measured execution time distribution between the benchmarks, we have determined the average and maximum observed execution time, the standard deviation, as well as the execution times observed at the 90th and 99th percentiles. Figure 8.1 depicts the relative execution time differences between those values and the average execution time for each benchmark. A low relative difference denotes a narrow execution time distribution, while a higher difference in the execution time indicates a wider execution time distribution.

The data in Figure 8.1 show that the execution time distribution of most benchmarks is relatively narrow, with a standard deviation of at most 89.58 µs and – except for the `coremark` and `qurt` benchmarks – only small differences in the higher percentiles. Thus, it seems that the argument for exploiting the execution time distribution characteristics of `HI`-critical tasks with a dynamic scheduling approach is impractical. But, the main motivation for these scheduling policies stems from the difference between the *upper-bounded execution time* $M_b(\texttt{HI})$ which is either determined based on a factor of the maximum observed execution time (as discussed in the evaluation setup) or by a static WCET analysis. Due to the execution time uncertainty demonstrated by the measurements above, the resulting upper-bounded WCET can easily exceed the observed maximum execution time by a significant factor [18]. Additionally, small variations in the distribution may alter the resulting upper-bound considerably since they indicate an execution time uncertainty on the platform. Hence, even in the case of this benchmark suite, there exists a significant execution time variance on complex MPSoC platforms. The results therefore indicate the need for modelling this dynamic behaviour in analysis models and simulation methods to

provide an accurate performance estimation in the mixed-criticality design flow.

8.2 DYNAMIC MIXED-CRITICALITY SCHEDULING

A major benefit of the dynamic mixed-criticality scheduling policy is to increase the overall resource utilisation for LO-critical subsystems without sacrificing the scheduling and timing guarantees given by a comparable static scheduling configuration. In this section, we compare both approaches and measure the resulting utilisation improvements on the platform. Based on our previously determined scheduling configurations, we define a typical *static* time-triggered scheduling approach with fixed slot durations and compare its run-time behaviour against the *dynamic* scheduling configuration where slot lengths can vary depending on the HI-critical subsystem's dynamic resource usage.

8.2.1 *Scheduling Configuration*

As discussed in the introduction of this chapter, we follow contemporary industrial and academic practices for defining an upper-bound on the WCET. Recall that we have used the maximum observed task execution times $C_{max}(\mathcal{T}_b)$ of each benchmark b to allocate a HI-critical slot duration of $M_b(\texttt{HI}) = 3C_{max}(\mathcal{T}_b)$, representing our WCET upper bound. We have further configured the frame length $\delta_b = 5C_{max}(\mathcal{T}_b)$ such that the LO-critical video encoding application can be executed in the same frame for $2C_{max}(\mathcal{T}_b)$. As a result, in the *static* setup, the HI-critical application is allocated to the $M_b(\texttt{HI})$ interval and the LO-critical application is allocated $\frac{\delta_b - M_b(\texttt{HI})}{\delta_b} = 40\,\%$ of the processor time within one frame. The specific configuration settings depend on each benchmark and are listed in Table 8.1 on Table 8.1. As in the previous sections, the measurements were performed for each benchmark b by instantiating the system configuration $\mathbb{C}$ with the corresponding scheduling configuration.

In the *static* scheduling setup, we have executed the scheduling behaviour according to a static time-triggered scheduling approach: the hypervisor performs a context switch after the fixed duration given by the configured margin $M_b(\texttt{HI})$ which was determined at design time. This setup therefore resembles a typical static time-triggered scheduling approach. In contrast, the dynamic configuration allows the HI-critical subsystem to yield the control back to the hypervisor when it has finished its computation. In this case, the LO-critical slot is able to consume the rest of the frame duration δ_b directly. Therefore, in the *dynamic* setup, if the HI-critical subsystem would yield the processor after the average observed execution time $C_{avg}(\mathcal{T}_b)$, it would increase the frame encoding throughput for the LO-critical application by 80 %. Depending on the chosen margin and frame length which are influenced by the HI-critical slot execution time distribution, the dynamic approach could therefore result in significantly higher resource availability for the LO-critical subsystem.

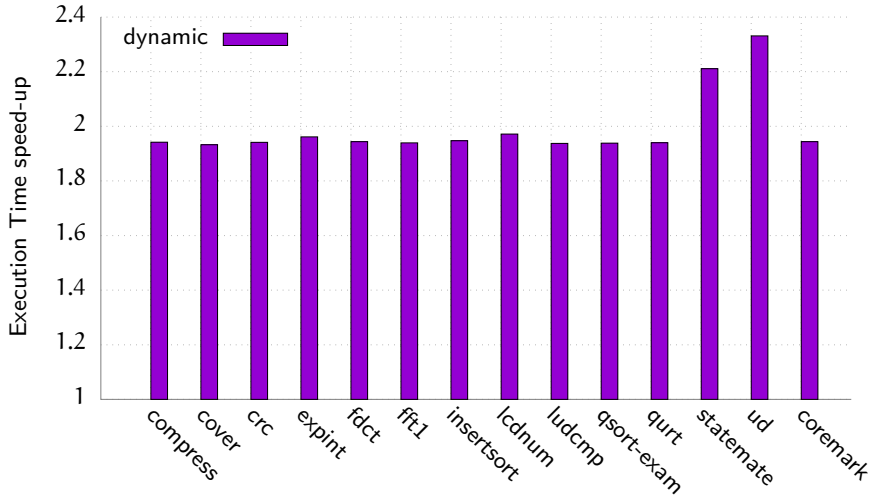


Figure 8.2: LO-critical video encoding end-to-end execution time speed-up in the dynamic scheduling configuration.

8.2.2 *Static & Dynamic Scheduling Comparison*

We have evaluated both the static and dynamic setup by measuring the average end-to-end execution time of the video encoding task τ_e executed within the LO-critical slot. As we can see in Figure 8.2, the dynamic configuration is in fact able to decrease the response time of the LO-critical task by an average factor of 1.99, suggesting that the HI-critical application indeed consumed much less time than assumed when configuring the execution time upper bound $M_b(\mathtt{HI})$.

As an example, in the `coremark` benchmark, the average execution time of the HI-critical task was 131.05 ms. In the dynamic case, the LO-critical task was hence able to execute for an average duration of 525.50 ms in each scheduling frame, which is 80 % of the total frame duration. In contrast, the static configuration only allowed for an execution of 262.61 ms ($\approx$ 40 % of the frame duration) due to the static time-triggered scheduling approach.

We can conclude from the results that with dynamic scheduling policies in mixed-criticality scenarios, the necessity for specifying a pessimistic upper-bound, especially when targeting complex platforms with high execution time uncertainties, does not consequently force a reduced overall resource utilisation for lesser critical subsystems. When integrating subsystems incorporating real-time constraints, such as the HI-critical subsystem in our evaluation, with best-effort functionality such as the video encoder of the LO-critical subsystem, such a dynamic scheduling approach can therefore be beneficial because it may considerably increase the LO-critical subsystem throughput.

However, the dynamic temporal behaviour of these mapped applications in the context of the dynamic scheduling policy imposes challenges on evaluating such potential resource utilisation gains in scenarios where HI- and LO-critical software subsystems are developed independently of each other. Since the dynamic resource utilisation depends on the temporal behaviour

of all mapped software components, its impact is thus only revealed in a fully integrated evaluation scenario. Chapter 9 focusses on the contribution for providing integration scenarios through *application proxies*, where HI-critical software components are either modelled or even replaced by proxies which mimic their temporal behaviour, such that it is possible to evaluate run-time performance metrics for LO-critical subsystems without the need for developing an integrated prototype.

8.3 SEGREGATION OVERHEAD

This section considers the overhead induced by the segregation mechanisms of the platform and the hypervisor implementation. The proposed implementation features two mechanisms for mitigating subsystem-level interferences on the integrated mixed-criticality platform.

The first mechanism considers the implicit sharing of resource state across mixed-critical subsystems. In Section 6.4.3, we have discussed how hypervisors implement support for spatial and temporal segregation in terms of virtualisation mechanisms for the platform resources. Processors, peripherals, and memory components are *virtualised* and the hypervisor manages their physical state such that they can be transparently shared across different application instances. While these mechanisms provide sufficient isolation such that the state of the virtual component cannot be read or modified by another subsystem, this mechanism does not prevent leaking *temporal* behaviour across virtual machine instances, since it depends on the internal state of the resource. This circumstance is especially prevalent on processors with a cache hierarchy, where the internal state of the cache determines the execution time of the software. Other mechanisms in modern processor architectures, such as branch prediction buffers, pipelines, etc., also leverage keeping internal state to increase the average execution time. As a consequence, a hypervisor should also manage the *internal* context of the processor and switch the state accordingly when scheduling the subsystems. However, current MPSoC platforms do not feature the proper hardware support for this level of virtualisation and the processor Instruction-Set Architecture (ISA) does not support managing the internal state of all processor components directly. Nevertheless, this section considers the most influential source of execution time uncertainty and as a first step towards managing internal processor state, the hypervisor implementation used in this thesis has therefore been expanded with the mechanism of flushing the cache before a slot is executed, such that its cache state is predictable across slot invocations.

Besides managing the processor state, the hypervisor also emulates each access to privileged memory regions (such as peripherals) by intercepting the execution and updating the corresponding *virtual state* of the accessed component. Thus, each access to these specially marked memory regions will force a context switch to the hypervisor.

Considering these sources of overhead, we first evaluate the impact on slot-based cache flushes, which is our technical contribution towards *processor state management*. In the next section, we then evaluate the overhead

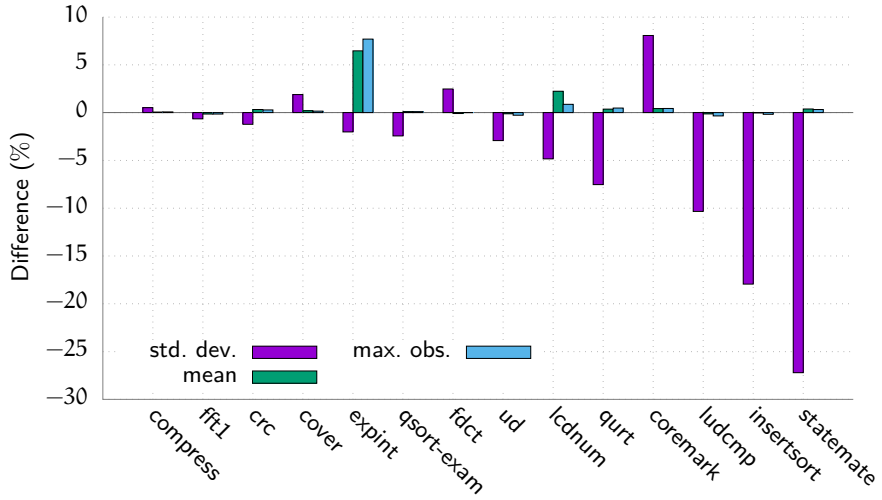


Figure 8.3: Relative change of the execution time distribution in the `flush` scenario compared to `noflush`, sorted by the relative difference in their standard deviation.

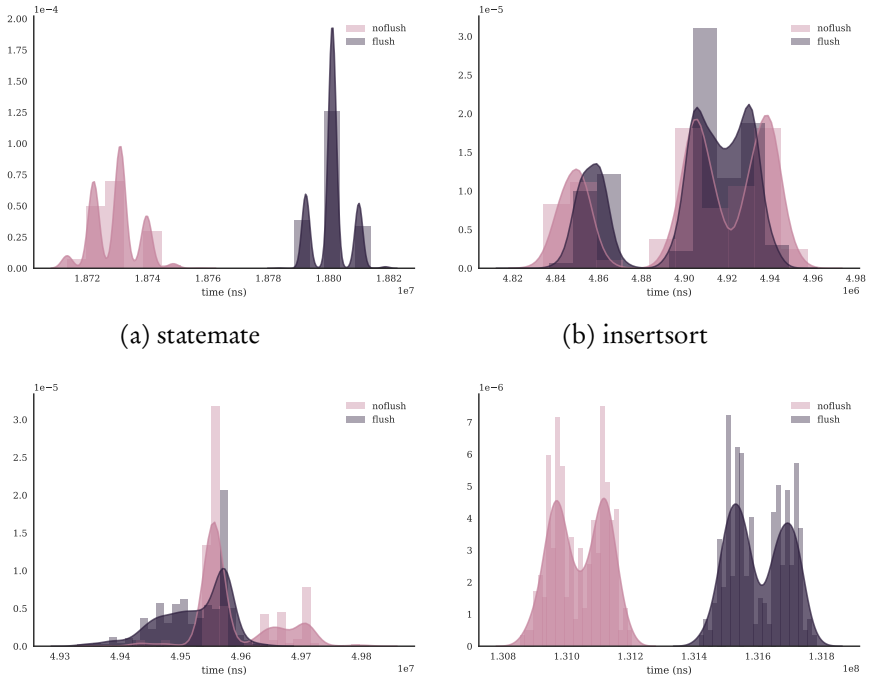
induced by managing the subsystem context as well as the integrated monitoring and measurement infrastructure, representing the *hypervisor platform state management*.

8.3.1 *Cache Flush*

In this section, we define two evaluation setups to measure and compare the performance impact of the cache flush mechanism. The system configuration $\mathbb{C}_b$ is then evaluated for each setup. In the `noflush` setup, the hypervisor switches between different subsystems directly. In the `flush` setup, a data cache flush is performed before switching to the next slot within a frame. We then compare the effects of the cache flush in both setups on the `HI`-critical execution time for each benchmark by measuring its slot execution time for each system configuration $\mathbb{C}_b$.

The relative differences between the `flush` and `noflush` setups for each benchmark are depicted in Figure 8.3. As can be seen, on average, the cache flush overhead increases the mean run-time duration across the benchmark suite by 0.72 % and reduces the standard deviation by 4.58 %. However, these trends are inconsistent across the benchmark suite results. The varying control flow dominance and memory footprint size characteristics of each benchmark presumably play a significant (and expected) role in the impact of the cache flush operations.

An additional test comparing the `flush` and `noflush` setups by considering cache refill behaviour has shown that on average, a benchmark running under the `flush` configuration performs 243.71 cache refill operations, while the average number of refills in the `noflush` setup is only 6.43. The number of cache refill operation varies throughout the benchmark suite from 84 for the `expint` benchmark to 401 for the `coremark` benchmark.



(a) statemate (b) insertsort

(c) ludcmp (d) coremark

Figure 8.4: Execution time probability distribution of the four benchmarks with the highest standard deviation differences between the `flush` and `noflush` configuration.

In the `expint` benchmark, we can see that the difference of the `flush` setup manifests mainly in the mean and maximum observed execution time. This is expected, since the benchmark itself is not heavily utilising the cache. Hence, the cache flush operation overhead impact is more significant due to the absence of any narrowing effects. The number of cache refill operations also correlates with the impact on the benchmark's timing distribution. We can therefore conclude that the segregation mechanism is effective, but its impact on the execution time distribution highly depends on the application characteristics.

Figure 8.4 illustrates the distributions of the four benchmarks with the highest standard deviation. The visual comparison of the distributions observed in both setups demonstrate the effect of cache flushes on the execution time behaviour. As we can see, the general observation is that the flush operations induce a constant overhead to the overall execution time. This effect is especially visible for the `coremark` and the `statemate` benchmark. Both benchmarks perform multiple memory operations and therefore profit from keeping the cache state across invocations which manifests in their average execution time.

Besides this constant overhead, we can also identify that the distribution shape is transformed to be more narrow. An interesting effect of both of these transformations can be seen when comparing the distributions of `insertsort` and `ludcmp` in both setups. Even in the context of the cache flush penalty, the distribution of the `flush` setup has been narrowed

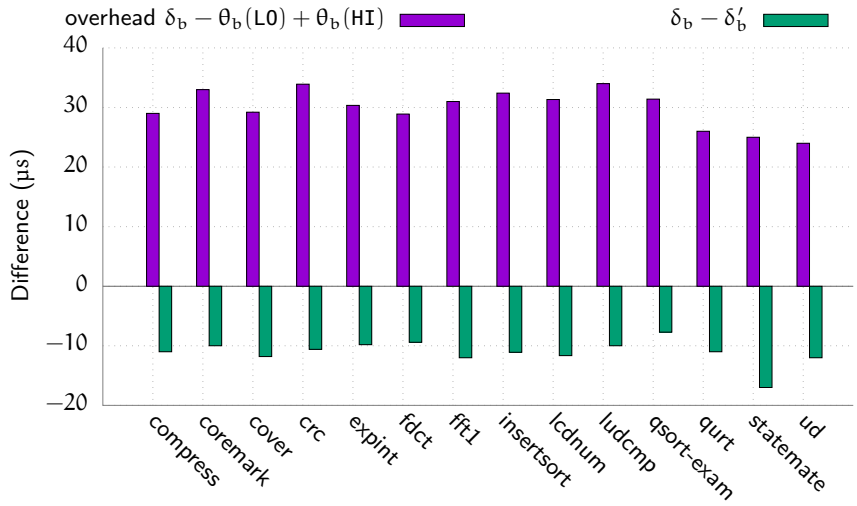


Figure 8.5: Absolute frame durations compared to the configurations and the summarised slot execution duration inside a frame.

such that the maximum observed execution time is lower compared to the `noflush` setup.

8.3.2 *Application Context Switching*

We have further measured the context switching overhead (including cache flushes) by comparing the slot execution time duration with the overall frame execution time duration. To demonstrate the overall hypervisor context switching overhead with the actual subsystem execution time duration, the measured frame execution time duration δ'_b and the difference to the configured duration δ_b were observed for each system configuration $\mathbb{C}$. The results were furthermore compared to the execution time duration of both slots $\theta_i(\texttt{LO})$ and $\theta_i(\texttt{HI})$ which are active within a scheduling frame.

The results depicted in Figure 8.5 show that on average, the measured frame execution δ'_b takes 11.08 µs longer than the configured duration δ_b. There is also a relatively constant offset between the configured frame and observed summarised slot execution time duration averaging at 29.96 µs which includes the measurement infrastructure overhead and the overhead of managing the virtualised state of the platform. This relatively high overhead is caused by the lack of hardware virtualisation support. Since the evaluation was performed on an ARMv7 architecture without virtualisation extensions, the hypervisor needs to emulate the virtualisation, which includes managing the context switch for the processor as well as the MMU using software routines. Each context switch therefore involves storing the general-purpose and control registers to a dedicated memory structure and retrieving the old state from the corresponding internal structures. Furthermore, it involves issuing a context switch on the MMU.

In total, from the perspective of the configured frame duration δ_b, the observed per-frame overhead averages at 41.04 µs. This duration covers the hypervisor context switches, the cache flush operation described in the previ-

ous section, the measurement overhead during a frame duration, and the offset of the configured and the measured frame duration. In the next chapter, this overhead serves as an input to our performance model.

9 MODELLING ACCURACY

This chapter considers the modelling accuracy of the performance model introduced in Section 7.2 and the application proxy described in Section 7.3. The previous sections of the evaluation serve as a basis for the input parameters of both models. After having determined F and d_F in Section 8.1.1 as well as the scheduling overhead $\mathcal{O}$ in Section 8.3, we can now apply the results to our proposed design and integration flow.

In an isolated experiment, we have determined that the video frame encoding duration of 24 frames requires 4.33 s of computation time on the platform. Using this information, we now evaluate the modelling accuracy by comparing the predicted QoS penalty for the `LO`-critical video encoding task with its implementation and integration with the `HI`-critical subsystem on the platform. We also compare the measurement results of the `LO`-critical end-to-end execution time when instantiated with an application proxy of the `HI`-critical subsystem. Finally, we estimate the *goodness of fit* regarding the `HI`-critical timing model and the generated samples used in the application proxies.

The evaluation in this chapter consists of three different measurement configurations. First, we estimate the temporal behaviour of the `LO`-critical application with the performance model (configuration `model`) and then evaluate the temporal behaviour in a setup where the `HI`-critical application is substituted by an application proxy (configuration `proxy`). The results are compared against a full integration of both `HI`- and `LO`-critical subsystems on the target (configuration `original`). In all three configurations we compare the resulting end-to-end execution time of the `LO`-critical video encoding task to evaluate the modelling accuracy. As in the previous sections, each of these three evaluation configurations were measured under the system specification $\mathbb{C}$ instantiated for each benchmark `b`.

9.1 PERFORMANCE MODEL

The results of the measured end-to-end execution time of encoding 24 frames in the `LO`-critical subsystem were compared to the predicted execution time penalty of the performance model and are depicted in Figure 9.1. Across all benchmarks, the duration for encoding one video frame spanned 4.50 s to 5.53 s. As can be seen in the results, on average, the `model` estimates are within 0.37 % of the measured results. The median deviation for the `model` estimates is −0.02 %.

The `expint` benchmark exhibits a maximum deviation of 4.97 % from the target measurements. Compared to the other benchmarks, `expint` has the smallest average slot duration of 348.03 µs, which allows for comparatively minor variations in the target measurements to accumulate over time and result in higher differences when estimating the end-to-end exe-

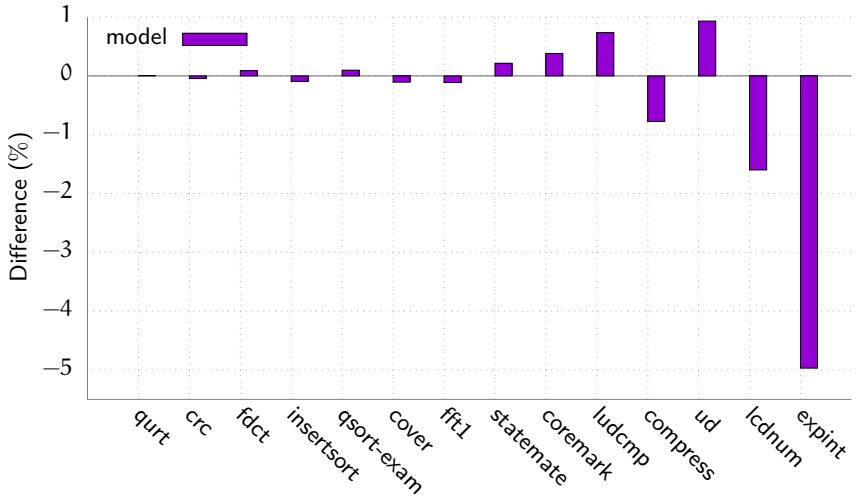


Figure 9.1: `LO`-critical video encoding duration estimates from the `model` estimates compared to the `original` measurements.

cution time of the `LO`-critical subsystem. Furthermore, the scheduling configuration in the system configuration instantiated for the `expint` benchmark (listed in Table 8.1 on Table 8.1) shows that the frame length is only 1757 µs. This length combined with an average `LO`-critical encoding duration of 5.52 s indicates that in this particular configuration, the `HI`-critical subsystem interrupted the `LO`-critical execution approximately 3144 times during its encoding operation. The difference of execution times by more than three orders of magnitude explain the relatively high error of the performance model.

In contrast, with the `coremark` benchmark containing an average slot duration of 131 602.82 µs, the highest of the benchmarks, the performance model result error is only 0.38 %. The `coremark` scheduling configuration has a frame length of 658 483 µs, and with an end-to-end execution time of 5.39 s, the `LO`-critical encoder was only interrupted approximately eight times during execution. This demonstrates the performance model sensitivity in the presence of short `HI`-critical slots and a long `LO`-critical end-to-end execution time.

9.2 APPLICATION PROXY

The measurement results of the configuration `proxy` in relation to the `original` measurements are shown in Figure 9.2. In the `proxy` configuration, the maximum deviation from the `original` video encoding duration is 1.14 % for the `ud` benchmark. Overall, the average error is 0.37 %, with a median of 0.44 %.

When comparing the results of the `expint` benchmark run to the corresponding performance model estimates, we can see that the `proxy` provides better estimates with an error of 0.17 % than the performance model, which produced an error of −4.97 %. In fact, the `expint` benchmark has the smallest error of all `proxy` configuration measurements. The increased accuracy

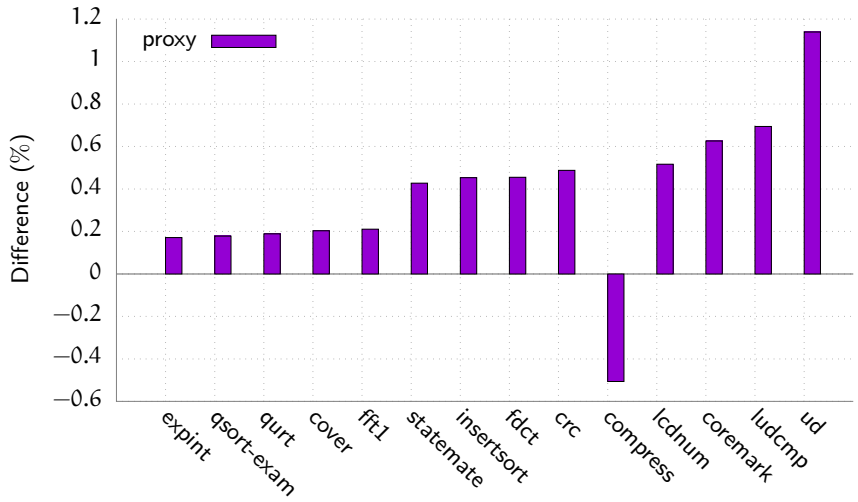


Figure 9.2: Video frame encoding duration estimates of the LO-critical task from the proxy subsystem compared to the original measurements.

is a result of measuring the actual overhead on the platform instead of using the abstracted value 0 in the performance model. Variations in the overhead are therefore more accurately represented in the proxy approach.

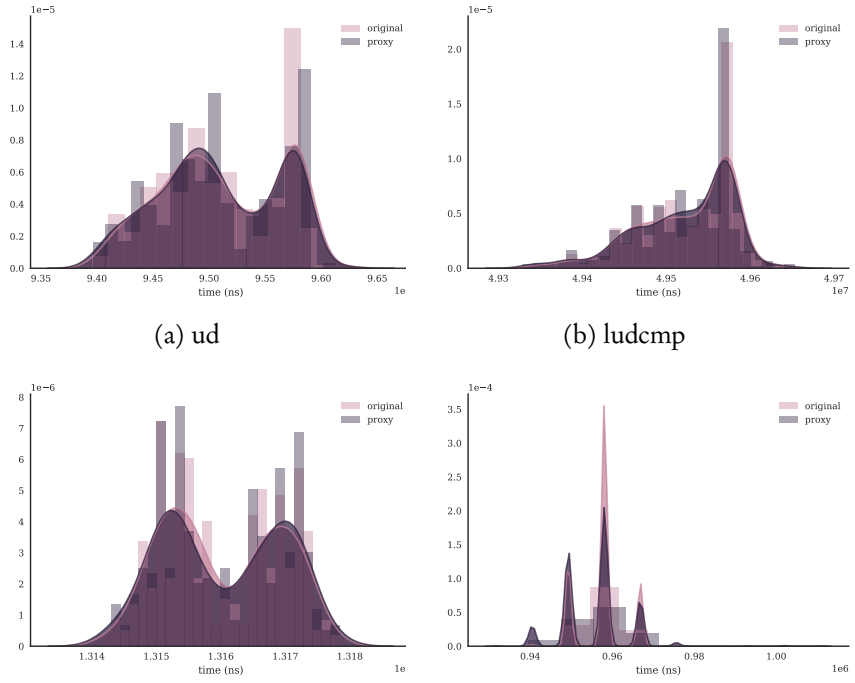
While the ud benchmark exhibits the highest error of $1.14\,\%$ in the proxy configuration, it is still within the range of the corresponding error of $0.93\,\%$ in the model configuration. The results therefore demonstrate that application proxies can provide accurate results for evaluating the performance metrics of LO-critical subsystems, and they are in fact even more accurate when representing slots with a short duration.

9.3 PROXY & MODEL DISTRIBUTION PROPERTIES

In addition to the LO-critical behaviour in terms of its execution time, we have also examined the distribution properties of the HI-critical distribution of the original implementation and the application proxy. To achieve this, we have used the measurement infrastructure again in the proxy mode to determine the overall slot duration distribution on the platform while the samples were replayed on the target, as discussed in Section 7.3.

Figure 9.3 illustrates the results of the original and proxy distributions for ud, crc, insertsort, and qurt, the four benchmarks with the highest error in the proxy configuration. The depicted PDF of the original and the sampled measurements show that the application proxy is able to represent the characteristic target execution time behaviour of the original measurements. This demonstrates a clear benefit in terms of modelling details regarding the run-time dynamics with a PDF compared to the expected execution time in the performance model.

A detailed view on the relative differences in the target measurements of the original and the proxy distribution is depicted in Figure 9.4. The plot shows the relative difference between observed average and maximum observed execution times as well as the execution time at the 90th and 99th



(a) ud (b) ludcmp

(c) coremark (d) lcdnum

Figure 9.3: Visualisation of the `proxy` and `original` Probability Density Functions of the four benchmarks with the highest error in their `L0`-critical performance estimation.

percentiles. In all measurement points, the error is less than 1 % with the highest deviating value being the 99th percentile of the `lcdnum` benchmark at 0.91 %.

Figure 9.5 additionally illustrates the difference between the `proxy` and the `original` measurements regarding their standard deviation. Interestingly, despite the relatively low error in the `L0`-critical estimates in Figure 9.4, the difference in the standard deviation averages at 11.09 % with the highest deviation at the `expint` benchmark at 33.96 %. Looking at the PDF visualisation of `expint` in Figure 9.3 however, we can see that the extracted PDF still closely matches the distribution of the `original` configuration. However, it seems that measurement points at the margin are overrepresented in the `proxy` case, while the `original` measurements produced more samples around the distribution mean value. The reason for this behaviour is – again – the duration of the `expint` slot. Since one `expint` slot duration only takes 1757 µs, the number of samples needed for accurately representing the distribution is the highest across the benchmark suite. Due to the fact that the experiment lasted ten minutes, we would have to generate at least 341 491 samples for the `proxy` measurements, which consumes 2.73 MB and exceeds the current technical limitation of 1 MB in the hypervisor configuration structures. Therefore, we have chosen to limit the number of samples in the `expint` configuration to 5000 (only 1.46 % of the full sample duration) and replay the sample list from the start when the list has been consumed. However, in the case of the `expint` benchmark, this

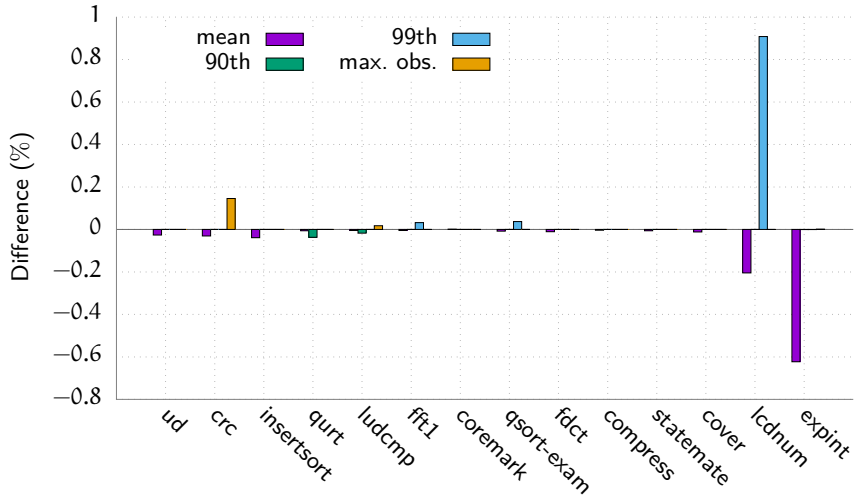


Figure 9.4: Relative change of the HI-critical proxy distribution characteristics compared to the original measurements.

limitation in the proxy approach still yields more accurate results than the model estimates.

Another reason for the differences in the standard deviation is the dynamic bin size adjustment performed by the measurement infrastructure at run-time. Since the distribution range and bin size will be dynamically chosen, they depend on the sample resolution. The standard deviation is especially sensitive to the quantization error introduced by the histogram bins and therefore shows the highest deviation from the original measurements in the results. The quantization error can be seen in the similar patterns of the distributions in Figure 9.6, illustrating four benchmarks with the highest standard deviation error. Due to the dynamic properties of the histogram storing mechanism, any execution time outliers can have a more significant impact on the overall data quality, since they influence the overall histogram resolution.

However, the results have also shown that the error of the standard deviation does not correlate with the modelling accuracy. The average relative differences in the mean duration, the percentiles and the maximum execution time of the measured distributions compared to the original distributions in Figure 9.4 are negligible with less than 0.10 %.

Although at first glance the overall estimates in the perf and proxy configurations are similar and presumably both model the HI-critical timing behaviour with a maximum error of 5 %, the measured distribution characteristics of the proxy configuration not only show a more complex temporal behaviour in their distribution characteristics, but they also have the advantage of a more accurate representation of subsystem-agnostic implementation artefacts such as the overhead for the mixed-criticality scheduling. These artefacts are abstracted away when modelled using the expected value and an average overhead, as performed in the perf model.

While the proxy approach can yield better results, it is still more complex in terms of the integration effort for measuring performance values, since

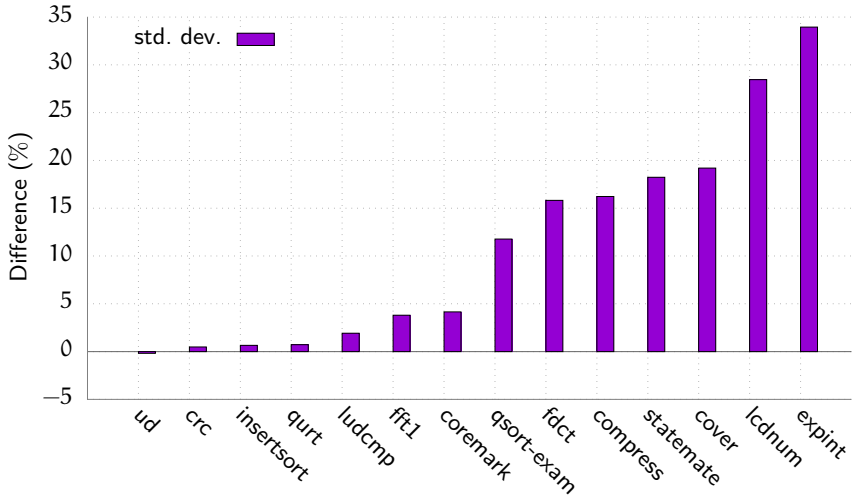


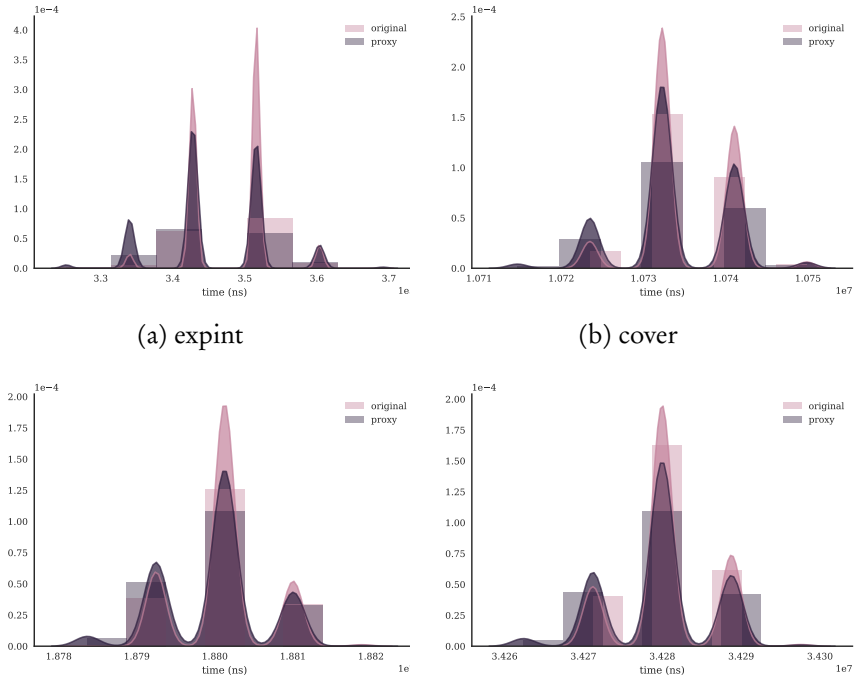
Figure 9.5: Relative change of the `proxy` standard deviation compared to the `original` measurements.

the `LO`-critical subsystem needs to be implemented on the platform. This is however countered with the possibility to integrate the subsystem without the `HI`-critical subsystem and simply using isolated measurement results as samples for the `proxy` configuration. This approach therefore is a clear benefit in the context of mixed-critical integration scenarios where `LO`-critical functional behaviour should be integrated into an existing platform along a `HI`-critical subsystem: the possibility to synthesise the `HI`-critical behaviour into a timing model on the platform removes the need for integrating it with the `LO`-critical subsystem in order to gain performance impact results.

9.4 APPLICATION PROXY SAMPLE SIZE

In Section 7.3.1, we noted that the possible number of application proxy samples is constrained by the memory size of the dedicated buffer. We concluded that for small sample sizes, the necessary number of samples to perform an experiment for a given amount of time might be limited. We proposed that by repeating the sequence, the experiment can run for a duration independent of the number of samples. This section considers the question of how many samples are needed to be drawn from the `original` distribution to sufficiently represent the measured timing behaviour on the platform while repeating the samples. To answer this question, we first have to specify what *sufficiently* means in this context.

The problem of determining this *sufficiency* is that we cannot determine whether the generated `proxy` distribution matches the measured `original` distribution with a statistical test such as the Anderson-Darling [3] test or the Kolmogorov-Smirnov [59] test, since they can only help in providing significant evidence for disproving their H_0 hypothesis, which states that two sample sets are drawn from the same distribution. The reason for this issue is that one can never gather enough data to prove that the `proxy` samples are drawn from the `original` distribution. Instead, the evaluation in this sec-



(a) expint (b) cover

(c) statemate (d) compress

Figure 9.6: Visualisation of the `proxy` and `original` Probability Density Functions of four benchmarks with the highest standard deviation error. The benchmark `lcdnum` is shown in Figure 9.3.

tion focusses on what constitutes a *representative* distribution in our setup and what its possible impacts are on the estimation of a `LO`-critical application. We therfore first determine which properties of the `proxy` distribution are relevant the context of this work.

Regarding the dynamic scheduling policy behaviour, a sufficiently accurate description of the software timing behaviour provides equal probabilities of exceeding certain timing margins in both the `original` and the `proxy` configuration. Since the duration of the `HI`-critical slot directly defines the `LO`-critical slot length, we choose to compare the probabilities at a set of quantiles of both distributions. If they are similar, we can deduce that the `LO`-critical slot will be allocated a comparable slot duration under both the `original` and `proxy` distributions.

This experiment is therefore constructed as follows. First, we derive the PDF from the `original` data, as previously done in the evaluation on the application proxy behaviour. Each `original` measurement data consists of N measurement points and the benchmarks were executed for $10\,\mathrm{min}$. From the constructed PDF, we then extract n samples which should be used for the application proxy. Note that when executing the `proxy` for $10\,\mathrm{min}$, the sequence of n samples might be repeated, depending on the number of samples. Thus, a `proxy` sample list can be identified by its ratio of $\frac{n}{N}$ *unique* samples. In the experiment, $1 - \frac{n}{N}$ of the replayed samples are repeated by re-iterating the sample list, as discussed in the application proxy implementation in Section 7.3.1.

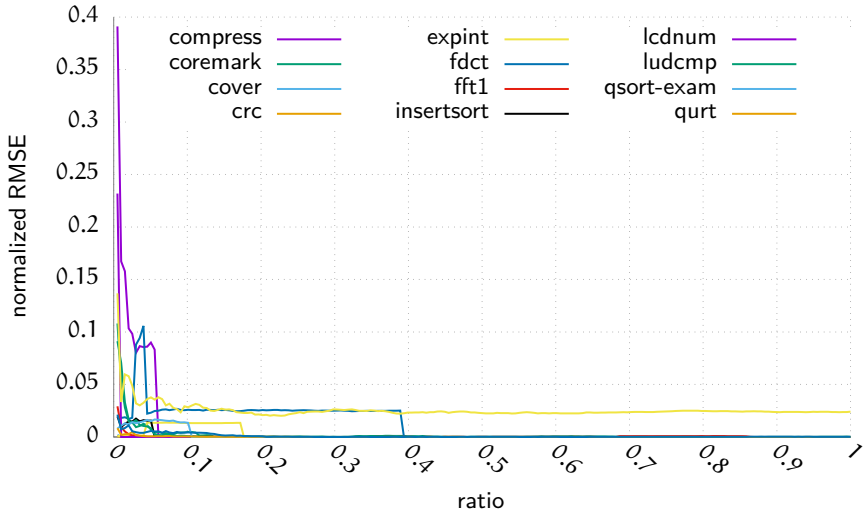


Figure 9.7: Comparison of different ratios for samples.

The goal of this experiment is to determine how many samples n in relation to the measured sample count N are needed to represent a similar behaviour at the quantiles as the original distribution. To evaluate this *goodness of fit* for each sample list, we construct a PDF from each sample list and compare their percentiles at quantiles 50, 90, 99, and 100 to the percentiles of the original measurement data. The error at these quantiles was calculated by a normalized root-mean-square error (RMSE). The resulting value provides an estimate on the error induced by repeating the sample list with ratio $\frac{n}{N}$ in terms of modelling the distribution quantiles of the `original` application.

As we can see in Figure 9.7, in all benchmarks, a low ratio (high replay rate) results in a higher error regarding the quantile values. However, as the ratio of unique samples in the sequence increases, the error gradually decreases. Starting with a ratio of 0.40, the error does not decrease further in any benchmark. This indicates that only about 40 % of data was needed to construct an application proxy from the measurement data and that repeating the sample subset with can provide similar results to the original measurements in terms of their quantile values.

9.5 SUMMARY

In this chapter, we have evaluated the programming model, the performance model, and the application proxy mechanism proposed in this thesis. First, Section 8.1 discussed the evaluation setup in terms of the OSSS/MC models and their properties introduced in Chapter 6 along with the evaluation method and the experimental setup. The evaluation considered various WCET benchmarks as the `HI`-critical subsystem and a video encoder application as the `LO`-critical subsystem. We described the process of empirically estimating suitable scheduling properties for the different system configurations and produced a system configurations $\mathbb{C}_b$ for each benchmark b.

Next, we have compared the distributions of the slot duration of each benchmark and identified the potential benefits of dynamic mixed-criticality scheduling policies in terms of the overall platform resource utilisation. These dynamic effects were evaluated in Section 8.2, where we could observe that in our scheduling setup, the overall resource availability for LO-critical subsystems under a dynammic mixed-criticality scheduling policy increases by a factor of 1.99.

Section 8.3 then considered the overhead of the hypervisor implementation and its mechanisms for ensuring the segregation of mixed-critical subsystem in terms of the temporal and spatial integration artefacts. In particular, we focussed on the effects of cache flushing after each subsystem invocation and the general overhead induced by the hypervisor processor and platform context management.

Empirically setting up the scheduling configuration and measuring the overhead then allowed us to evaluate the estimates of our performance model in Section 9.1. On average, the model estimates produced an error of 0.37 % compared to measurements of the actual implementation on the mixed-criticality platform. The results of the application proxy mechanism for modelling the HI-critical subsystem timing behaviour were presented in Section 9.2. They indicate that the application proxy provides a more accurate timing model of the HI-critical application when comparing the estimates of the LO-critical behaviour with both approaches.

While the average estimation error with the application proxy method is nearly identical to the results of the performance model estimates, we could identify several improvements regarding the representation of the timing behaviour using the proxy application approach and identified shortcomings regarding the performance model approach. The discussion in Section 9.3 revealed some limitations of the proxy approach regarding the number of samples required for accurately representing the HI-critical subsystem timing behaviour. We concluded that the application proxy provides a more accurate representation of the implementation artefacts on the platform and thus yields more accurate results for a performance estimation. The downside of the application proxy approach is clearly the requirement to implement and integrate the LO-critical subsystem on the target platform. However, this process is greatly simplified since the HI-critical application does not need to be integrated as well. The application proxy, which can be constructed once and then re-used across different implementation and integration stages for the LO-critical subsystem, enables separating the development process for both subsystems which can be exploited by increasing the parallelism in embedded mixed-criticality systems design processes.

10 CONCLUSION & FUTURE WORK

10.1 CONCLUSION

This thesis started with the observation of the importance of considering mixed-criticality scheduling and resource utilisation effects in the design flow of embedded CPS consisting of safety-critical and performance-critical, QoS software applications. The contributions of this thesis address the challenges of designing and integrating embedded mixed-critical software components on a common platform, where the functionality can be categorised along their criticality into *high-* and *low-critical*.

The first contribution aimed at answering the question of considering the mixed-criticality properties of safety- and performance-critical software applications along their functional behaviour in an embedded systems design flow. This thesis answers this question with OSSS/MC, a programming model and integration flow for mixed-criticality software components. The programming models allows considering the criticality of functionality along its functional behaviour in the embedded systems design flow. It additionally provides structural and behavioural components to perform function partitioning into computation and communication units, and an overall clustering of functional behaviour by their criticality.

A major focus of this thesis has been the systematic refinement from the programming model OSSS/MC towards an implementation on a MPSoC platform to answer the question of how to use the functional description along with the mixed-criticality properties to derive an implementation on a MPSoC. The thesis provides a method for mapping the programming model components to a runtime model of the VRL. This yields an executable functional specification in a SystemC-based simulation environment and allows checking the functional correctness of the partitioned behaviour during simulation. The configuration of virtual platform resources is embedded in the specification model as well. A systematic deployment approach can therefore extract all relevant platform configuration properties from the model, including the scheduling configuration and the task mapping. The proposed implementation of the runtime model on the platform features a hypervisor-based subsystem approach that implements spatial and temporal segregation. A systematic refinement of the runtime model then considers the application-level model structures and their resource binding properties and implements its semantics based on a flexible, time-triggered mixed-criticality scheduling policy on the platform.

The evaluation of the target implementation considered the impact of reducing the inter-application interference through explicit cache invalidation when switching between subsystems. The differences in the timing behaviour of this approach demonstrated that the effect on the timing behaviour is highly application-dependant and needs to be considered when estimating timing behaviour on the platform. While there is a noticeable impact of the

cache invalidation and the context management with 20.52 µs on our platform, modern embedded target platforms already contain hardware support for these segregation techniques which reduce the overhead significantly.

The second part of this thesis considered the performance estimation of low-critical subsystems mapped to the mixed-criticality platform. The question which dynamic timing properties arise from the application behaviour as well as the platform-level scheduling behaviour in an integrated mixed-criticality system has been answered by providing a measurement infrastructure embedded in the hypervisor-based implementation which allows the designer to capture dynamic application-level timing behaviour as well as the platform segregation overhead. The measurement infrastructure allows exploring the dynamic timing properties that arise from the application behaviour as well as the platform-level scheduling behaviour on the mixed-criticality platform.

Gathering the data from the measurement infrastructure lead to the contribution of two performance estimation methods that aim at representing the integration impact of dynamic mixed-criticality timing properties. The first estimation features a performance model derived from the measurement results and the scheduling configuration where the timing models were constructed from the measurements on the platform and represented as the Probability Density Function of the timing behaviour. The performance model considers the dynamic behaviour of a high-critical subsystem on the platform and yields the integration impact in terms of a timing penalty for best-effort, low-critical subsystems. These models can be parametrised with the scheduling setup and the isolated timing properties of the low-critical subsystem, thereby enabling performance estimation of the system before integrating it on the platform.

Given an end-to-end response time as the performance characteristic of the low-critical application, the proposed performance model uses the estimated value of the distribution and the platform-dependant segregation overhead measurements to provide an estimate about the expected end-to-end execution time when integrated with the modelled high-critical application on the flexible, time-triggered platform. The discussion of the results lead to the conclusion that this modelling technique can be especially valuable in early analysis steps when implementing a low-critical application. Results of the modelling accuracy have shown that on average across the benchmark suite, the model estimates only exhibit an error of 0.37 % when compared to the measured results.

The second approach of estimating the performance impact considers a typical mixed-criticality integration scenario, where both high- and low-critical software components are to be integrated on the platform featuring the dynamic mixed-criticality scheduling policy. The presented *application proxies* are implementations of the high-critical subsystem's timing model on the platform and therefore resemble the temporal behaviour of the high-critical subsystem. As a result, the performance impact on a low-critical subsystem can be measured on the target platform without requiring to integrate the functional behaviour of the high-critical application as well.

The results of the application proxy approach only yielded an error of 0.37 % compared to the measurement of the fully integrated setup with

both mixed-criticality subsystems running on the platform. This allows for two conclusions. First, the results indicate that the application proxy is able to provide accurate timing estimates for a performance estimation of the low-critical subsystem. Seond, since the high-critical functionality was not executed, this demonstrates that the segregation efforts implemented by the hypervisor layer indeed enforce the subsystem independence. In the context of a performance evaluation, the difference of whether executing the original high-critical subsystem or the timing model is negligible. The clear benefit of the application proxy approach however is that it provides an integration context for low-critical subsystems in the absence of the functional behaviour of the high-critical subsystem. The application proxies are therefore a viable approach for decoupling the development and integration process on mixed-criticality platforms with dynamic scheduling artefacts.

10.2 FUTURE WORK

Support for parallel execution on multi-core processors.

This work considers *multi-processor* platforms by implementing timing interference mitigation techniques for isolating mixed-criticality software applications mapped to the same processing element. We have demonstrated our approach by managing processor state segregation through a hypervisor implementation and mitigate timing interference caused by the processor cache subsystem through explicit cache flushes. As concluded in the evaluation, the techniques effectively prevent timing interferences of sequentially executing mixed-criticality subsystems executed on the same processing element. When moving towards supporting multi-core systems with parallel executing tasks, there are two main challenges to consider regarding timing segregation.

The first issue is that parallel executing processor cores cause interference on platform resources which are implicitly shared on the architecture, such as memory subsystems, peripherals, and shared cache hierarchies. The work in this thesis assumes that the complexity and the optimisations of the processor core itself and the cache state can be captured in the timing uncertainties modelled by the PDF. A restriction of this approach therefore is that it does not consider timing effects induced by parallel executing tasks on these shared resources. As a consequence, executing tasks on multiple processor cores in parallel invalidates the timing assumptions currently captured in the model. Therefore, the first step towards supporting parallel executing tasks is to prevent the tasks from causing timing interferences. The locally observed timing properties on a processing element need to be composable in the sense that they cannot depend on the behaviour of other tasks running in parallel on other processing elements on the platform.

An application model which is consistent with these timing isolation requirements therefore cannot allow arbitrary synchronisation of tasks across the cores, since that would result in communication delays that depend on the exact execution time of each task, which cannot be determined exactly, as demonstrated by the PDF modelling approach and the discussions on de-

termining execution times on complex platforms. Therefore, the second requirement of supporting parallel task execution is to explicitly model communication in the application model and consider these synchronisation points when mapping the tasks on the parallel running cores.

The proposed mixed-criticality application model already provides explicit communication through Shared Objects, such that performing required dependency analysis is straight-forward. Extending this work towards supporting multi-core systems first requires combining this dependency information with the scheduling analysis. The analysis can then restrict the mapping decisions based on the task dependency information. Additionally, the targeted platform needs to guarantee composability regarding parallel task execution time. Considering the MPSoC platform used in this thesis, a possible technique for limiting cache behaviour interferences is to partition the cache, such that parallel executing tasks do not interfere with the state (and therefore the timing) of both cores and are instead limited to the core where the task is mapped to. Our implemented cache flush operations would then again guarantee freedom from interference for tasks running on the same core (in the same cache partition) while the partitioning isolates the parallel running tasks from each other.

Integrating scheduling analyses in the application model.

In the current setup, the scheduling configuration chosen by the designer is assumed to be consistent with the overall task requirements on the specified periodicity and deadlines. This step currently requires the designer to consider the WCRT of each task mapped to the slots of the scheduling policy manually. A straight-forward future extension of the proposed configuration approach is to provide the periodicity and deadline information of the specification model along with the desired scheduling configuration to an external analysis step and check for inconsistencies. With this additional step, the refinement setup could reject scheduling policy misconfigurations automatically.

As a further improvement, the configuration approach presented in this thesis could be used to skip the manual scheduling configuration entirely by incorporating the scheduling analysis results into the specification model again, thereby automatically deducing a feasible schedule for the FTTS policy during the specification phase.

Support for data-dependant timing models.

This work demonstrated how to construct PDF-based timing models from tasks without considering any possible data dependencies regarding their temporal behaviour. Currently, data-dependency is only supported by generating different timing models in isolation, given that the designer specifies different workload scenarios for representative clusters of input parameters of the high-critical application. Each workload scenario would then require a full integration test with the proposed application proxies, since the im-

plementation is unable to dynamically change to different PDFs during the execution on the platform.

To model data-dependent timing behaviours with the PDF approach, a possible extension is to categorise different input stimuli into clusters which are expected to show similar timing behaviour. Constructing a PDF for each of these clusters with the existing measurement infrastructure then yields multiple PDFs for each slot. Furthermore, switching between these behavioural clusters can be modelled and simulated using a stochastic process, e. g. with the help of Markov Chains [15].

The limitation of this approach is that the designer needs to provide expert knowledge for clustering the application timing behaviour and derive the parameters of the stochastic process. In the simulation and application proxy, the implementation would then switch between different timing behaviours by simulating the Markov Chains to determine the distribution set for the next slot invocation. However, in case of highly data-dependant high-critical applications, this extension could provide a more refined timing model and improve the performance estimation of low-critical timing behaviour.

Timing distribution simulation in the executable specification model.

We have shown that the current simulation backend provides methods for functional validation of the task partitioning and supports single-value timing annotations. However, our measurement and modelling results have shown that a single timing annotation is a highly abstract approach when trying to reflect the actual timing behaviour, as seen by the constructed PDF models.

A possible future extension is to allow simulating the extracted PDF from the measurements. The generated samples of the PDF could be integrated back into the SystemC-based simulation model in a straight-forward manner, either by a list of samples in the same manner as the timing model implementation on the target, or by implementing the PDF in the simulation models. With this approach, the simulation environment would be able to provide performance estimates of low-critical behaviour, even if their annotation is still based on a single estimated execution time annotation.

Functional behaviour scenarios in low-critical applications.

The contributions of this thesis focussed on the integration impact of a low-critical application when integrated with a high-critical application. We have shown that in the case of a purely performance-driven low-critical application, we can estimate the integration impact with models derived from the timing behaviour of a high-critical application. In our scheduling setup, the scheduling policy can preempt the low-critical slot and resume its execution in the next slot dedicated to the low-critical application.

As discussed in the related work section, most mixed-criticality scheduling policies allow executing low-critical behaviour in different *scenarios* depending on the resource availability on the platform. If the high-critical

application was able to finish execution below a certain threshold, the low-critical application is allowed to operate in a more resource-demanding scenario. In the other case, the low-critical application may operate in a degraded scenario and adjust its resource consumption according to the available time in its slot.

A necessary requirement for such a low-critical scenario mode switching is to communicate the resource state at each slot invocation and the possibility of the functional behaviour to switch between these scenarios at the slot boundary. To support this dynamic scenario switching, the designer needs to configure the scheduling policy in such a manner that all low-critical tasks finish within their slot. Then, the invocation of a task at a slot boundary can take the current resource consumption into account and adjust its scenario to the available slot length in the remaining scheduling frame.

The proposed implementation already provides support for communicating the available length of the low-critical slot in the frame with the introduction of the M_i frame property. On the platform, the underlying hypervisor implementation manages a dynamic scenario variable which is set according to the duration consumed by the high-critical slot in the current frame. A low-critical task can read this information upon invocation. However, since the remaining frame duration and therefore the scenario variable is only valid for the current frame, preempting a low-critical task and resuming it in the next slot results in outdated information for the low-critical task and may not have the desired effects on its resource consumption behaviour.

The challenge in supporting multiple low-critical scenarios therefore is how to manage mode switches after a task resumes in the next slot. Contemporary mixed-criticality scheduling policies such as FTTS restrict low-critical task preemption such that all tasks of all criticalities require finishing within their assigned slot. This restriction limits the possible low-critical application behaviour, as it would prevent use-cases where short periodic high-critical tasks are integrated with resource-consuming, best-effort low-critical functionality which cannot be trivially partitioned into the available slots of the scheduling frames. Another approach would be to integrate *checkpoints* into the control flow graph of a low-critical task where it is able to adjust its resource behaviour in coordination with its functional behaviour. Assuming that low-critical tasks or their control flow graphs can be partitioned at such a level of granularity, our proposed simulation model may support the designer in identifying issues with functional behaviour and criticality scenario switches.

APPENDIX

BIBLIOGRAPHY

[1] Irune Agirre, Mikel Azkarate-Askasua, Carles Hernandez, Jaume Abella, Jon Perez, Tullio Vardanega and Francisco J. Cazorla. 'IEC-61508 SIL 3 Compliant Pseudo-Random Number Generators for Probabilistic Timing Analysis'. In: *Proceedings of the 2015 Euromicro Conference on Digital System Design*. Washington, DC, USA: IEEE Computer Society, 2015, pp. 677–684. ISBN: 978-1-4673-8035-5. DOI: 10.1109/DSD.2015.26.

[2] Rajeev Alur and David L. Dill. 'A theory of timed automata'. In: *Theor. Comput. Sci.* 126.2 (25th Apr. 1994), pp. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8.

[3] Theodore W. Anderson and Donald A. Darling. 'Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes'. In: *Ann. Math. Stat.* (1952), pp. 193–212. DOI: 10.1214/aoms/1177729437.

[4] Amro Awad and Yan Solihin. 'STM: Cloning the spatial and temporal memory access behavior'. In: *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 237–247. DOI: 10.1109/HPCA.2014.6835935.

[5] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster and Leen Stougie. 'Mixed-Criticality Scheduling of Sporadic Task Systems'. In: *Algorithms – ESA 2011*. Ed. by Camil Demetrescu and Magnús M. Halldórsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 555–566. ISBN: 978-3-642-23719-5. DOI: 10.1007/978-3-642-23719-5_47.

[6] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster and Leen Stougie. 'Preemptive Uniprocessor Scheduling of Mixed-Criticality Sporadic Task Systems'. In: *J. ACM* 62.2 (May 2015), pp. 1–33. ISSN: 00045411. DOI: 10.1145/2699435.

[7] Sanjoy Baruah, Alan Burns and Robert Ian Davis. 'An Extended Fixed Priority Scheme for Mixed Criticality Systems'. In: *Workshop on Real-Time Mixed Criticality Systems (ReTiMics)*. 2013.

[8] Sanjoy Baruah and Steve Vestal. 'Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications'. In: *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, July 2008, pp. 147–155. ISBN: 978-0-7695-3298-1. DOI: 10.1109/ECRTS.2008.26.

[9] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi and Martijn Hendriks. 'UPPAAL 4.0'. In: *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*. QEST '06. Washington, DC, USA: IEEE Com-

puter Society, 2006, pp. 125–126. ISBN: 0-7695-2665-9. DOI: 10.1109/QEST.2006.59.

[10] Robert H. Bell Jr. and Lizy K. John. 'Improved Automatic Testcase Synthesis for Performance Model Validation'. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. ACM, 2005, pp. 111–120. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088164.

[11] Ron Bell. 'Introduction to IEC 61508'. In: *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*. Vol. 55. SCS'05. Darlinghurst, Australia: Australian Computer Society, Inc., 2006, pp. 3–12. ISBN: 1-920682-37-6. URL: http://dl.acm.org/citation.cfm?id=1151816.1151817.

[12] Aimen Bouchhima, Patrice Gerin and Frédéric Pétrot. 'Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-simulation'. In: *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 546–551. ISBN: 978-1-4244-2748-2. URL: http://dl.acm.org/citation.cfm?id=1509633.1509762 (visited on 11/04/2018).

[13] Carlo Brandolese, Simone Corbetta and William Fornaciari. 'Software Energy Estimation Based on Statistical Characterization of Intermediate Compilation Code'. In: International Symposium on Low Power Electronics and Design (ISLPED'11). IEEE, 2011, pp. 333–338. ISBN: 978-1-61284-659-0. DOI: 10.1109/ISLPED.2011.5993659.

[14] Alan Burns and Robert I. Davis. 'A Survey of Research into Mixed Criticality Systems'. In: *ACM Comput Surv* 50.6 (Nov. 2017), 82:1–82:37. ISSN: 0360-0300. DOI: 10.1145/3131347.

[15] Kai Lai Chung. *Markov chains with stationary transition probabilities*. OCLC: 682058891. Berlin: Springer, 1960. ISBN: 978-3-642-49686-8.

[16] Norman H. Cohen. *ADA As a Second Language*. 2nd ed. McGraw-Hill Higher Education, 1995. ISBN: 978-0-07-011607-8.

[17] A. Crespo, I. Ripoll and M. Masmano. 'Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach'. In: *2010 European Dependable Computing Conference*. Apr. 2010, pp. 67–72. DOI: 10.1109/EDCC.2010.18.

[18] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoıt Triquet and Reinhard Wilhelm. 'Predictability Considerations in the Design of Multi-Core Embedded Systems'. In: *Proc. Embed. Real Time Softw. Syst.* (2010), pp. 36–42.

[19] H. J. Curnow and B. A. Wichmann. 'A synthetic benchmark'. In: *Comput. J.* 19.1 (1976), pp. 43–49. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/19.1.43.

[20] Dewi Daniels. 'Are we there yet? A Practitioner's View of DO-178C/ED-12C'. In: *Advances in Systems Safety*. Ed. by Chris Dale and Tom Anderson. London: Springer London, 2011, pp. 293–313. ISBN: 978-0-85729-133-2.

[21] Álvaro Díaz, Héctor Posadas and Eugenio Villar. 'Obtaining Memory Address Traces from Native Co-Simulation for Data Cache Modeling in SystemC'. In: *XXV Conf Des. Circuits Integr. Syst. DCIS10* (2010).

[22] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere and Lizy K. John. 'Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies'. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA '04. Washington, DC, USA: IEEE Computer Society, June 2004, pp. 350–361. ISBN: 978-0-7695-2143-5. DOI: 10.1109/ISCA.2004.1310787.

[23] *EEMBC - CoreMark - Processor Benchmark*. URL: https://www.eembc.org/coremark//index.php (visited on 11/04/2018).

[24] Rolf Ernst and Marco Di Natale. 'Mixed Criticality Systems — A History of Misconceptions?' In: *IEEE Des. Test* 33.5 (Oct. 2016), pp. 65–74. ISSN: 2168-2356. DOI: 10.1109/MDAT.2016.2594790.

[25] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega and Francisco J. Cazorla. 'Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art'. In: *14th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Heiko Falk. Vol. 39. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014, pp. 31–42. ISBN: 978-3-939897-69-9. DOI: 10.4230/OASIcs.WCET.2014.31.

[26] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. 1st ed. Springer Publishing Company, Incorporated, 2009. ISBN: 978-1-4419-0503-1.

[27] Karthik Ganesan and Lizy Kurian John. 'Automatic Generation of Miniaturized Synthetic Proxies for Target Applications to Efficiently Design Multicore Processors'. In: *IEEE Trans. Comput.* 63.4 (Apr. 2014), pp. 833–846. ISSN: 0018-9340. DOI: 10.1109/TC.2013.36.

[28] Andreas Gerstlauer, Rainer Dömer, Junyu Peng and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Springer Science & Business Media, 6th Dec. 2012. 264 pp. ISBN: 978-1-4615-1481-7.

[29] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang and Lothar Thiele. 'Scheduling of mixed-criticality applications on resource-sharing multicore systems'. In: *Proceedings of the International Conference on Embedded Software (EMSOFT'2013)*. IEEE, Sept. 2013, pp. 1–15. ISBN: 978-1-4799-1443-2. DOI: 10.1109/EMSOFT.2013.6658595.

[30] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang and Lothar Thiele. 'Mapping mixed-criticality applications on multi-core architectures'. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE Conference Publications, 2014, pp. 1–6. ISBN: 978-3-9815370-2-4. DOI: 10.7873/DATE.2014.111.

[31] Pablo González, Pablo Pedro Sánchez and Álvaro Díaz. 'Embedded software execution time estimation at different abstraction levels'. In: *XXV Conf Des. Circuits Integr. Syst. DCIS10* (2010).

[32] Sven Goossens, Benny Akesson, Martijn Koedam, Ashkan Beyranvand Nejad, Andrew Nelson and Kees Goossens. 'The CompSOC design flow for virtual execution platforms'. In: *Proceedings of the 10th FPGAworld Conference*. ACM Press, 2013, pp. 1–6. ISBN: 978-1-4503-2496-0. DOI: 10.1145/2513683.2513690.

[33] Patrick John Graydon and Iain John Bate. 'Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling'. In: *Proceedings of the Workshop on Mixed-Criticality Systems*. 2013, pp. 19–24.

[34] Kim Grüttner, Andreas Herrholz, Philipp A. Hartmann, Andreas Schallenberg and Claus Brunzema. *OSSS – A Library for Synthesisable System Level Models in SystemC(TM) – The OSSS 2.2.0 Manual*. Sept. 2008.

[35] Kim Grüttner and Wolfgang Nebel. 'Modelling Program-State Machines in SystemC'. In: *2008 Forum on Specification, Verification and Design Languages*. 2008 Forum on Specification, Verification and Design Languages. Sept. 2008, pp. 7–12. DOI: 10.1109/FDL.2008.4641413.

[36] Jan Gustafsson, Adam Betts, Andreas Ermedahl and Björn Lisper. 'The Mälardalen WCET Benchmarks: Past, Present And Future'. In: *10th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010, pp. 136–146. ISBN: 978-3-939897-21-7. DOI: 10.4230/OASIcs.WCET.2010.136.

[37] Thomas A. Henzinger and Joseph Sifakis. 'The Embedded Systems Design Challenge'. In: *Proceedings of the 14th International Conference on Formal Methods*. FM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1–15. ISBN: 978-3-540-37215-8. DOI: 10.1007/11813040_1.

[38] Carles Hernández, Jaume Abella, Francisco J. Cazorla, Alen Bardizbanyan, Jan Andersson, Fabrice Cros and Franck Wartel. 'Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study'. In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Ed. by Marko Bertogna. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, 16:1–16:23. ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.16.

[39] Clay Hughes and Tao Li. 'Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis'. In: *IEEE Intl. Symposium on Workload Characterization (IISWC'08)*. IEEE, 2008, pp. 163–172. ISBN: 978-1-4244-2777-2. DOI: 10.1109/IISWC.2008.4636101.

[40] Yonghyun Hwang, Samar Abdi and Daniel Gajski. 'Cycle-approximate retargetable performance estimation at the transaction level'. In: *Proc. of the Design, Automation & Test in Europe Conf. (DATE'08)*. 2008, pp. 3–8. ISBN: 9783981080. DOI: 10.1145/1403375.1403380.

[41] 'IEEE Standard for Standard SystemC Language Reference Manual'. In: *IEEE Std 1666-2011 Revis. IEEE Std 1666-2005* (2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[42] ISO. 'ISO26262 Road vehicles – Functional safety'. In: *Int. Organ. Stand. Geneva Switz.* (ISO 26262 2011).

[43] Philipp Ittershagen, Kim Grüttner and Wolfgang Nebel. 'Mixed-Criticality System Modelling with Dynamic Execution Mode Switching'. In: *Proceedings of the Forum on Specification and Design Languages (FDL'15)*. Barcelona, Spain, 2015. DOI: 10.1109/fdl.2015.7306356.

[44] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner and Achim Rettberg. 'Hierarchical Real-time Scheduling in the Multi-core Era – An Overview'. In: *Proceedings of the 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'13)*. IEEE, June 2013. DOI: 10.1109/isorc.2013.6913241.

[45] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers and H. Meyr. 'A SW performance estimation framework for early system-level-design using fine-grained instrumentation'. In: *Proc. of the Design Automation & Test in Europe Conf. (DATE'06)*. IEEE, 2006, pp. 468–473. ISBN: 3-9810801-1-4. DOI: 10.1109/DATE.2006.243830.

[46] H. Kopetz and G. Bauer. 'The time-triggered architecture'. In: *Proc. IEEE* 91.1 (Jan. 2003), pp. 112–126. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805821.

[47] Angeliki Kritikakou, Thibaut Marty and Matthieu Roy. 'DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems'. In: *ACM Trans. Des. Autom. Electron. Syst.* 23.2 (Oct. 2017), 13:1–13:26. ISSN: 1084-4309. DOI: 10.1145/3110222.

[48] Leslie Lamport. 'An Axiomatic Semantics of Concurrent Programming Languages'. In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 77–122. ISBN: 978-3-642-82453-1. DOI: 10.1007/978-3-642-82453-1_4.

[49] Asier Larrucea, Imanol Martinez, Vicent Brocal, Salvador Peirò, Hamidreza Ahmadian, Jon Perez and Roman Obermaisser. 'DREAMS: Cross-domain mixed-criticality patterns'. In: *Workshop on Mixed-Criticality Systems*. 2016, p. 6.

[50] Edward A Lee, Stephen Neuendorffer and Michael J Wirthlin. 'Actor-Oriented Design of embedded Hardware and Software Systems'. In: *J. Circuits Syst. Comput.* 12.03 (2003), pp. 231–260.

[51] A. Lenz, T. Pieper and R. Obermaisser. 'Global Adaptation for Energy Efficiency in Multicore Architectures'. In: *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'17)*. Mar. 2017, pp. 551–558. DOI: 10.1109/PDP.2017.46.

[52] A. Lenz et al. 'SAFEPOWER Project: Architecture for Safe and Power-Efficient Mixed-Criticality Systems'. In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 294–300. DOI: 10.1109/DSD.2016.64.

[53] Joseph Y.-T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 27th Apr. 2004. 1215 pp. ISBN: 978-0-203-48980-2.

[54] Haohan Li and Sanjoy Baruah. 'Global Mixed-Criticality Scheduling on Multiprocessors'. In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. July 2012, pp. 166–175. ISBN: 978-1-4673-2032-0. DOI: 10.1109/ECRTS.2012.41.

[55] Grant Likely and Josh Boyer. 'A symphony of flavours: Using the device tree to describe embedded hardware'. In: *Proceedings of the Linux Symposium*. Vol. 2. 2008, pp. 27–37.

[56] C. L. Liu and J. W. Layland. 'Scheduling algorithms for multiprogramming in a hard-real-time environment'. In: *J. ACM* 20.1 (1973), pp. 46–61.

[57] I. Liu, J. Reineke and E. A. Lee. 'A PRET architecture supporting concurrent programs with composable timing properties'. In: *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*. 2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers. Nov. 2010, pp. 2111–2115. DOI: 10.1109/ACSSC.2010.5757922.

[58] Charles C. Mann. 'The end of Moore's law?' In: *Technol Rev* 103.3 (2000), pp. 42–42.

[59] Frank J. Massey Jr. 'The Kolmogorov-Smirnov test for goodness of fit'. In: *J Am Stat Assoc* 46.253 (1951), pp. 68–78. DOI: 10.1080/01621459.1951.10500769.

[60] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, John Scoredos et al. 'Mixed-criticality real-time scheduling for multicore systems'. In: *IEEE 10th International Conference on Computer and Information Technology (CIT)*. IEEE. IEEE, 2010, pp. 1864–1871. ISBN: 978-1-4244-7547-6. DOI: 10.1109/cit.2010.320.

[61] D. B. Noonburg and J. P. Shen. 'A framework for statistical modeling of superscalar processor performance'. In: *Proceedings Third International Symposium on High-Performance Computer Architecture*. Proceedings Third International Symposium on High-Performance Computer Architecture. Feb. 1997, pp. 298–309. DOI: 10.1109/HPCA.1997.569691.

[62] S. Nussbaum and J. E. Smith. 'Modeling superscalar processors via statistical simulation'. In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 15–24. DOI: 10.1109/PACT.2001.953284.

[63] M. Paolieri et al. 'Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability'. In: *IEEE Micro* 30.5 (Sept. 2010), pp. 66–75. ISSN: 0272-1732.

[64] A. Patel, M. Daftedar, M. Shalan and M. W. El-Kharashi. 'Embedded Hypervisor Xvisor: A Comparative Analysis'. In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Mar. 2015, pp. 682–691. DOI: 10.1109/PDP.2015.108.

[65] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.

[66] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo and Lui Sha. 'Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems'. In: *Real-Time Systems Symposium*. IEEE, Nov. 2008, pp. 221–231. DOI: 10.1109/RTSS.2008.42.

[67] *PikeOS Hypervisor - SYSGO - Embedding Innovations*. URL: https://www.sysgo.com/products/pikeos-hypervisor/ (visited on 11/04/2018).

[68] Peter Poplavko, Rany Kahil, Dario Socci, Saddek Bensalem and Marius Bozga. 'Mixed-Critical Systems Design with Coarse-Grained Multi-core Interference'. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 605–621. ISBN: 978-3-319-47166-2. DOI: 10.1007/978-3-319-47166-2_42.

[69] P. J. Prisaznuk. 'Integrated Modular Avionics'. In: *Proceedings of the IEEE National Aerospace and Electronics Conference*. IEEE. May 1992, pp. 39–45. DOI: 10.1109/NAECON.1992.220669.

[70] Karl Rupp. *42 Years of Microprocessor Trend Data*. Feb. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[71] Alberto Sangiovanni-Vincentelli and Grant Martin. 'Platform-based Design and Software Design Methodology for Embedded Systems'. In: *IEEE Des. Test Comput.* 18.6 (2001), pp. 23–33. DOI: 10.1109/54.970421.

[72] S. Stattelmann, O. Bringmann and W. Rosenstiel. 'Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems'. In: *Proc. of the conference on Design, Automation and Test (DATE'11)*. IEEE, Mar. 2011, pp. 1–6. DOI: 10.1109/date.2011.5763044.

[73] Stefan Stattelmann, Oliver Bringmann and Wolfgang Rosenstiel. 'Fast and Accurate Source-level Simulation of Software Timing Considering Complex Code Optimizations'. In: *Proceedings of the 48th Design Automation Conference*. DAC '11. New York, NY, USA: ACM Press, 2011, pp. 486–491. ISBN: 978-1-4503-0636-2. DOI: 10.1145/2024724.2024838.

[74] Stavros Tripakis. 'Compositionality in the Science of System Design'. In: *Proc. IEEE* 104.5 (May 2016), pp. 960–972. ISSN: 0018-9219. DOI: 10.1109/JPROC.2015.2510366.

[75] S. van der Walt, S. C. Colbert and G. Varoquaux. 'The NumPy Array: A Structure for Efficient Numerical Computation'. In: *Comput. Sci. Eng.* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[76] L. Van Ertvelde and L. Eeckhout. 'Benchmark synthesis for architecture and compiler exploration'. In: *IEEE International Symposium on Workload Characterization (IISWC)*. Dec. 2010, pp. 1–11. DOI: 10.1109/IISWC.2010.5650208.

[77] Steve Vestal. 'Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance'. In: *28th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE, Dec. 2007, pp. 239–243. ISBN: 978-0-7695-3062-8. DOI: 10.1109/RTSS.2007.47.

[78] Michael Waskom et al. *seaborn: v0.7.1*. June 2016. URL: https://doi.org/10.5281/zenodo.54844.

[79] Reinhold P. Weicker. 'Dhrystone: A Synthetic Systems Programming Benchmark'. In: *Commun. ACM* 27.10 (Oct. 1984), pp. 1013–1030. ISSN: 0001-0782. DOI: 10.1145/358274.358283.

[80] Reinhard Wilhelm et al. 'The worst-case execution-time problem—overview of methods and survey of tools'. In: *ACM Trans Embed. Comput Syst* 7.3 (Apr. 2008), pp. 1–53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389.

[81] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo and Lui Sha. 'Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality'. In: IEEE, July 2012, pp. 299–308. ISBN: 978-1-4673-2032-0. DOI: 10.1109/ECRTS.2012.32.

[82] *Zynq-7000 SoC Data Sheet: Overview (DS190)*. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

AL	Application Layer	32
AMP	Asymmetric Multi-Processing	17
API	Application Program Interface	16
ASIL	Automotive Safety Integrity Level	37
BRAM	Block-RAM	67
CISC	Complex Instruction Set Computer	30
COTS	Commercial Off-the-Shelf	49
CPS	Cyber-Physical Systems	2
DAL	Design Assurance Level	37
DMA	direct memory access	20
DUT	Design under Test	26
EDA	Electronic Design Automation	12
EDF	Earliest Deadline First	23
FMEA	Failure Mode and Effect Analysis	3
FPGA	Field-Programmable Gate Array	16
FTTS	Flexible Time-Triggered Scheduling	47
HAL	Hardware Abstraction Layer	24
HCFSM	Hierarchical Concurrent Finite State Machine	13
HDL	Hardware Description Language	19
IMA	Integrated Modular Avionics	26
ISA	Instruction-Set Architecture	98
ISS	Instruction-Set Simulator	24
KDE	Kernel Density Estimation	81
MMIO	Memory-Mapped I/O	20
MMU	Memory-Management Unit	68
MPSoC	Multi-Processor System-on-a-Chip	1
OSSS	Oldenburg System Synthesis Subset	11
PDF	Probability Density Function	81
QoS	Quality of Service	4
RMSE	root-mean-square error	110
RTL	Register-Transfer Level	20
RTOS	Real-Time Operating System	16
SIL	Safety Integrity Level	25

SLDL System-Level Design Language 18
SMP Symmetric Multi-Processing 17
SoC System-on-a-Chip 35
TDMA Time-Division Multiple Access 37
TLM Transaction-Level Modelling 20
vCPU virtual CPU 69
VHDL Very High Speed Integrated Circuit Hardware Description Language 33
VRL Virtual Resource Layer 55
VTA Virtual Target Architecture 32
WCET Worst-Case Execution Time 17
WCRT Worst-Case Response Time 44
PL Programmable Logic 67
PS Processing System 67

DECLARATION/ERKLÄRUNG

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis an der Carl von Ossietzky Universität Oldenburg und den DFG-Richtlinien festgelegt sind, befolgt habe. Des Weiteren habe ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste in Anspruch genommen.

Oldenburg, 20. August 2018

Philipp Ittershagen