Information Engineering Department
Institute of Communication, Information and Perception Technologies
University of Pisa, Scuola Superiore Sant'Anna

Code Generation for multi-core embedded systems with mix-critical applications

Master Degree in Embedded Computing Systems

Supervisors:
Prof. Marco Di Natale
Prof. Giorgio C. Buttazzo

External Supervisor:
Dr. Stylianos Basagiannis

Author:
Pasquale Antonante

University of Pisa

May 2017

Ai miei genitori, Aniello e Gianfranca, al loro amore e alla loro pazienza ...

Acknowledgements

This work would not have been possible without the advice and support of many. Firstly, I want to thank my supervisor, Marco Di Natale, for having helped me to get into the domain of Model-Based Design and for having always been available to answer my questions.

I would like to thank my colleagues from my internship at United Technologies Research Centre, you supported me greatly and were always willing to help me. I would particularly like to thank my supervisor Phil Harris for giving me the opportunity to work at UTRC. Stylianos Basagiannis, I want to thank you for all of the opportunities I was given to conduct my work. Thanks to Juan Valverde-Alcala, without your patience and suggestions this thesis would be in a far earlier state.

Lastly, I would like to thank all people expressed their support through their love, friendship and encouragement.

In questi ultimi due anni ho preso parte a tante attività, incontrando tante persone che mi hanno aiutato a ogni passo. Grazie ad Alex, Michele, Enrica, Jasmin, Martina, Angela, Chiara, Alessandra, Antonio, Michele, Elena, Antonello e Luca. Grazie per tutti i bei momenti passsati insieme.

Grazie a te Sara, averti avuto affianco in questi ultimi due anni per me è stato fonte di grande forza. Grazie per aver supportato, e spesso sopportato, la mia voglia di fare. Niente sarebbe stato uguale senza di te.

Grazie a tutti i parenti che mi hanno supportato, in particolare a zio Francesco, senza i tuoi insegnamenti non sarei chi sono oggi.

Grazie a voi papà e mamma. Siamo stati lontani ma il vostro aiuto non è mai venuto meno. Mi avete dato sempre tutto quello di cui avevo bisogno e di questo ve ne sono grato. Il futuro ci ripagherà dei nostri sacrifici.

Pasquale Antonante
Cork, April 2017

Abstract

In real-time and safety-critical systems, the move towards multi-cores is becoming unavoidable to satisfy the increasing processing requirements while maintaining a reasonable power consumption. A common trend in real-time safety-critical embedded systems is to integrate multiple applications on a single platform. Such systems are known as mixed-criticality systems as the applications are usually characterized by different criticality levels. However, multi-core systems are mainly designed to increase average performance, whereas embedded systems, and in particular mixed-criticality system, have additional requirements on safety, reliability and real-time behavior. Therefore, the shift to multi-cores raises several challenges. These architectures are challenging for safety-critical applications because they are in general not predictable. The difficulties increase when the multi-core hosts several applications and in particular mixed-critical applications.

Mixed-criticality embedded systems are gaining considerable interest, but there is a lack of model-based tools for their development. This thesis proposes a model-based approach to handle the design complexity with the support of optimization techniques and code generation methods.

Table of contents

List of figures

Chapter 1

Introduction

1.1 Embedded Systems

Most results in information processing and computer science usually apply to the domain of general computing (Personal-Computers and desktop applications included). However, according to several forecasts, the future of information and communication technologies (ICT) is characterized by terms such as *ubiquitous computing*, *pervasive computing*, *ambient intelligence*, and the *post-PC era*. These terms reflect the fact that computing (and communication) will be everywhere, the information available anytime and anywhere. The technology leading this future is the embedded systems technology.

Embedded systems, which are part of the broader area of Cyber-physical systems (CPS), are special-purpose computer systems designed to control or support the operation of a larger technical system, see [Ber12] for a conceptual map. Unlike the general-purpose computer, they only perform a few specific and more or less complex pre-defined tasks. The typical use cases of CPSs are medical devices, aerospace, autonomous systems (like robots, autonomous cars or Unmanned Aerial Vehicles - UAVs), process, factory and environmental control, and intelligent buildings. CPS interact with the physical world and must operate dependably, safely, securely, efficiently, and in real-time.

In a simpler case, software consists of a single program running in a loop, starting at power-on, and responding to certain internal or external events. In more complex cases (robotics or aerospace) operating systems are employed providing features like multitasking, synchronization, resource management, among others.

There are almost no areas of modern technology in which we could do without embedded systems. Rajkumar et al. [RLSS10] described CPS as *the next computing revolution*. CPSs are starting to pervade areas such as wearable electronics and domotic applications. As they

are becoming ubiquitous, we do not notice them anymore. Contemporary cars, for example, contain around 60 embedded computers[1]. The driver is not aware of them but uses their functionality.

By definition, embedded systems operate in real-time, which means that their temporal behavior is equally important as their functional behavior. The verification of functional properties by formal methods is today a reality, applied to hardwrae and software designs. However, the verification of temporal behavior (especially for hybrid systems, in which a continuous-time dynamic needs to be accounted for) is much more difficult and typically provides limited results. In this case, the verification is usually based on testing and the quality and coverage of these tests depends mostly on the requirements of the certification process (when applicable) or simply the experience and intuition of the developers.

1.2 Mixed-Criticality

As a consequence of the evolution of hardware systems, powerful and cheap computing platforms (especially multicore) are now available. These platform could support the execution of several functions, possibly with different criticality levels in an integrated fashion, with better flexibility, lower costs and power consumption. Criticality can include all forms of dependability (availability, integrity, etc.) [Lap92], but it usually refers to functional safety, i.e., the absence of catastrophic consequences on the user and the environment. This means that multiple functionalities (tasks) with different safety-critical levels, such as flight-critical and mission-critical tasks, could be integrated on a single, shared hardware device. as navigation or braking. However, the platform integration must be performed with the guarantee that the functions at different criticality levels are free from any type of mutual interference.

A more formal definition of criticality can be obtained with reference to the safety standards (see [EN16] for a more detailed discussion) that define the design and development processes for safety-critical embedded systems (hardware and software). There are a variety of domain specific safety standards, such ISO 26262 for road vehicles. This work is focused on aerospace where is usual to refer to DO-178C [oR11] for avionic software and ARINC 653 [4] for avionics real-time operating systems.

DO-178C (aka EUROCAE-ED-12B) was drafted by a co-operation of the European Organization for Civil Aviation Equipment (EUROCAE) and its US counterpart. The

[1] according to a 2014 report from the Alliance of Automobile Manufacturers

Level	Failure Condition	Failure Rate
A	Catastrophic	10^{-9}/h
B	Hazardous	10^{-7}/h
C	Major	10^{-5}/h
D	Minor	10^{-3}/h
E	No Effect	N/A

Table 1.1 Failure rate per DO-178C criticality level

standard considers the entire software life-cycle and provides a basis for avionic systems certification. It defines five levels of criticality, from A (components whose failure would cause a catastrophic failure of the aircraft) to E (components whose failure have no effect on the aircraft or pilot workload) as in table 1.1. This is the primary document by which the certification authorities approve all commercial software-based aerospace systems.

ARINC-653 is a software specification for space and time partitioning in safety-critical avionics real-time operating systems. It allows the hosting of multiple applications of different software insurance levels (SIL) on the same hardware in the context of an Integrated Modular Avionics (IMA) architecture. Each software component is inside a partition and has its memory space and dedicated time slot. The current work includes the enhancement of ARINC-653 for multi-core processor architectures.

All these documents are used by the certification authorities that must establish system safety. In this process, developers need to convince official entities that all relevant hazards have been identified and dealt with. In general, mixed criticality approaches are hardly practical in today's processes. The reason is that demonstrating functional and time isolation for most multi-core platforms is quite challenging give the complex set of interactions occurring at the hardware level (at the level of the memory hierarchy or the intercore connects).

All standards define different levels of concerns, in the aerospace field they are called Safety Integrity Level (SIL) in IEC 61508 and ARINC 653, or Design Assurance Level (DAL) in DO-178C. The levels indicate the severity and frequency of a function failure and assign requirements to failure probability, architectures, and design processes to each of the levels. They also regulate the combination of functions with different levels; so, they provide a basis for Mixed-Criticality systems design.

Because larger systems, such as vehicles or aircrafts, include a few safety-relevant applications and many non-critical ones, such as air conditioning or infotainment, mixed criticality

is a well-known problem for both research and industrial actors. Safety standards strongly regulate mixed-critical design and integration. The generic standard IEC 61508 requires that *sufficient independence* is demonstrated between functions of different criticalities. Designing the full system as being high-critical is clearly impractical. This approach is too costly. The application of the strict safety-critical development rules to non-critical functions, which often include third party (e.g., infotainment) subsystems, is far too costly, and demonstrating *sufficient independence* is the only viable option.

1.2.1 Robust Partitioning

The concept of *Robust partitioning* is defined differently by different standards, without an officially agreed or common definition [JFG+12]. Rushby [Rus00] defines the *Gold Standard for Partitioning* as *"A robustly partitioned system ensures a fault containment level equivalent to its functionally equivalent federated system."*. Federated architecture is the traditional design for avionic architecture where each application is implemented in self-contained units. Wilding et al.[WHG99] define the *Alternative Gold Standard for Partitioning* as *"The behavior and performance of software in one partition must be unaffected by the software in other partitions"*, which is a stronger property and a sufficient condition to establish robust partitioning. In any case, robust partitioning consists of the following the concepts:

- *Fault Containment*. Functions should be separated in such a way that no failure in one application can cause another function to fail. Low criticality tasks should not affect high criticality tasks.
- *Space Partitioning*. No function may access the memory space of other functions (unless explicitly configured).
- *Temporal Partitioning*. A function's access to a set of hardware resources during a period of time is guaranteed and cannot be affected by other functions..

ARINC-653 contains its interpretation of robust partitioning: *"The objective of Robust Partitioning is to provide the same level of functional isolation as a federated implementation."*. This space partitioning concept can be implemented on multi-core systems with the help of the Real-Time Operating Systems.

1.2.2 Operating systems for Mixed-Criticality applications

The core concept of demonstrating sufficient independence among different function can be approached using Kernels and schedulers that guarantee resource management to provide

independence in the functional and time domain; separation kernels are the most notable example.

A separation kernel is an operating system-level resource manager that enforces *spatial and temporal separation* among functionalities or partitions that are managed by it. The concept was first proposed by John Rushby in 1981 [Rus81], *"the task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow from one machine to another along known external communication lines. One of the properties we must prove of a separation kernel, therefore, is that there are no channels for information flow between regimes other than those explicitly provided."*. In other words, a single board platform that is indistinguishable by a federated system.

A typical structure of a separation kernel is depicted in figure 1.1. A partition is a logic

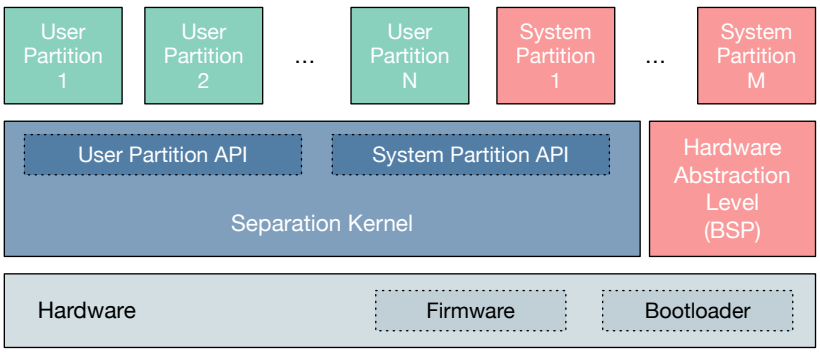


Fig. 1.1 Typical Separation Kernel Architecture

unit maintained by the separation kernel, and each of them is separated from the other. For each partition, the separation kernel provides resources such as physical memory space, I/O memory space, CPUs, and so on (spatial separation). Moreover, separation kernels are typically implemented using time-triggered schedulers and assigning each partition a dedicated time slot in a cycle to provide time separation. Usually two type of partitions are supported: *User Partitions* and *System Partitions*. The partitions are identified configured by the system designer, which is a trusted person. The content of a user partition does not need to be approved by the designer and can be arbitrary, even malicious [EM15], whereas a system partition contains applications and data supplied and approved by the designer. All partitions use the API (Application Program Interface) provided by the separation kernel to interact with it. Since partition are spatially isolated they cannot communicate directly with each other. They can only interact through the kernel API, and communicate under the supervision of the kernel. This communication occurs via objects that are statically configured by the designer. The separation kernel also includes the *Hardware Abstraction Level*, which includes

in part or completely the *Board Support Package* (BSP). The BSP contains a set of drivers for the hardware components, providing an abstraction of the underlying hardware, an onboard support package can be exchanged without changing the content of any partition.

Separation kernels are gaining importance thanks to the increasing adoption of multi-core embedded systems.

1.3 Multi-core embedded systems

The traditional approach to provide an increased processing power to software (including embedded) applications is to increase the CPU clock frequency: to increase the instruction level parallelism through instruction pipelines; to increase the cache size and number of cache levels and so on. With today's technology, this approach is no longer sustainable. Increasing the CPU frequency causes excessive power consumption and thermal dissipation loss and raises more problems for the chip designbecause of the need for smaller sized features. Parallelization has become a key solution for this problem. For this reason multi-core platforms have the potential to meet modern computing requirements by offering greater computational capabilities and advantages in size, weight, and power (SWaP).

However, in a multicore system different cores share hardware resources such as caches and central memory, which were developed focusing on maximizing the average performance, but when placed in the safety-critical context can introduce challenges to predictability.

Safety-critical multi-core CPS are still not fully embraced by the industry for safety critical applications. For example, aerospace systems are subject to costly and time-consuming certification processes, which require a predictable behavior under fault-free and certain hazardous conditions, hard to prove in multi-core platforms. Despite these problems (and the certification challenges), the industry is moving towards a higher exploitation of commercial-off-the-shelf (COTS) devices to reduce development costs[XJ12], as well as exploiting the low SWaP characteristics of multi-core.

The introduction of COTS multi-core processors is by motivated several aspects:

- Provide a long-term answer to the increasing demand of processing power.
  - Increased performance: better exploitation of the thread parallelism
  - Increased integration: Less equipment to perform the same functionality or same equipment to host more functionality

  - Reduce environmental footprint: Fewer embedded equipment, less power consumption, less dissipation compared to the single core equivalent

- Anticipate mass market obsolescence of single-core processors.
- Be able to "simplify" the overall system architecture (for example a partitioned architecture can avoid Ethernet communication).

The barrier to the adoption of the multi-core technology is its complexity of the certification. For the certification process it is important to ensure the *Execution Integrity* of its software components. That means it will be correctly executed in a nominal situation, and the system state will be predictable in non-nominal situations (internal faults). Moreover, it must be possible to perform a *WCET analysis* (Worst Case Execution Time) of the embedded software. Timing information is tightly coupled with both the software and hardware architecture since they introduce *interferences* between parallel applications that are reflected in timing delays. In COTS this analysis become even more difficult because of the lack of documentation on the system design.

1.3.1 Hardware interference channels

Applications running on different cores of a multi-core processor are not executing independently from each other. Even if there is no explicit data or control flow between these applications, a coupling exists at platform level since they are implicitly sharing resources. A platform feature which may cause interference between independent applications is called a hardware interference channel. The analysis of hardware interference channels requires a deep understanding of the platform architecture including the CPU internals.

In this work, we consider only commercial multi-core platforms. Therefore we assume that the platform implements the Unified Memory Model, which means that all cores share the same physical address space. Figure 1.2 depict a typical such architecture.

1.3.1.1 Caches

While two different cores can execute independently as long as they are not using shared resources, caches may introduce cross-CPU interference trough *Cache Coherency* and *Cache Sharing*. The L1 cache is typically divided into data and instruction cache while all other levels store data as well as instructions. Most multi-core processors have a dedicated L1 data and instruction cache for each core while other levels might be shared or not depending on the architecture. Shared caches are an essential cause of interference in a multi-core processor.

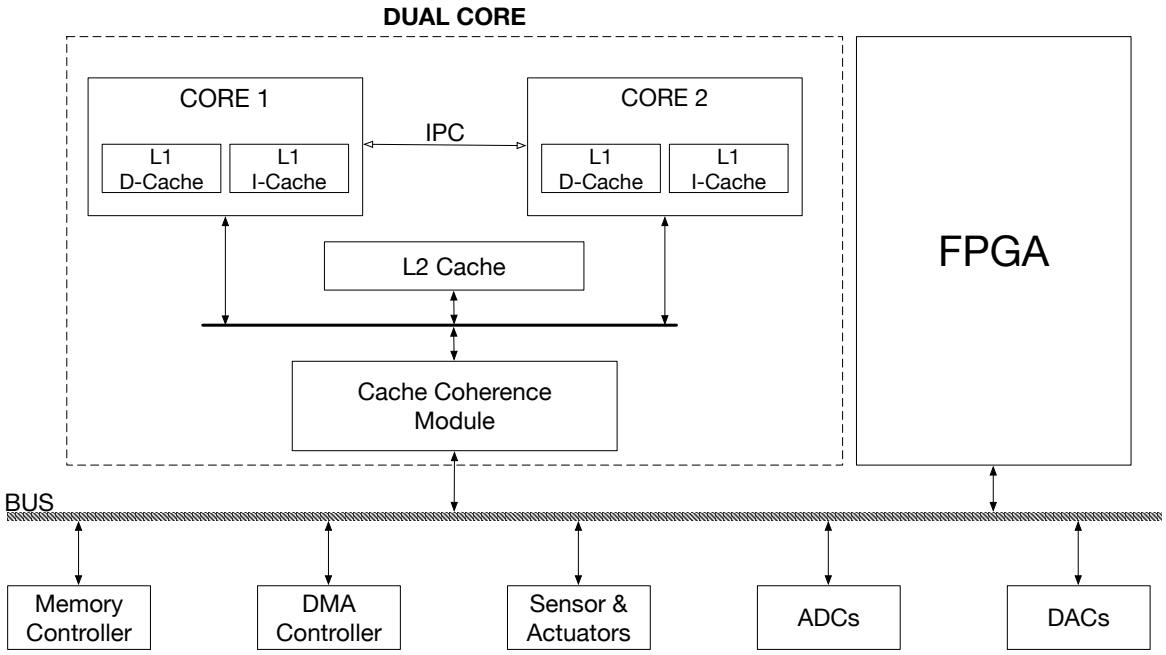


Fig. 1.2 Unified Memory Model Architecture

If data used by tasks are small enough to fit inside the private cache of the processor no performance loss occurs. If it is not the case, data are replaced according to some *cache replacement algorithm* on demand of the tasks executing on the cores. Therefore, tasks may experience long delays due to the recurring contentions when accessing the memory.

Another important aspect related to the use of caches is the consistency of local caches connected to a shared resource. Cache coherency is crucial in multi-core systems because: if one of the local caches of a core contains a reference to a physical resource, and the cached value is more recent than the value stored in the physical resource itself then any read access from any core must provide the value cached. In multi-core processors, cache coherency is solved by utilizing the core communication bus. So this mechanism determines how the memory system transfers data between processors, caches, and memory.

1.3.1.2 Interconnect

Multi-core processors are often part of a System on Chip (SoC) where the cores are packed together with peripherals such as external memory, serial I/O and Ethernet. To handle all requests to the shared peripherals, an *interconnect* is implemented to arbitrate the requests. It is the key point where all the accesses are performed and arbitrated. Indeed, the interconnect has been built to sustain a higher bandwidth to serve all cores efficiently. Usually, a significant

source of performance degradation is the concurrent access to shared Bus and shared I/O devices such as the graphical device, the GPIO (General Purpose Input Output) or the network interface. If a device can handle one request at a time, it may block the second request for hundreds of microseconds, or even worst for milliseconds. Moreover, shared devices can also rise Interrupts. On multi-core platforms, a hardware interrupt is typically routed to one core. If multiple devices are attached to one interrupt line and the devices are not served by the same core, the core which receives the interrupt must pass this interrupt to the other core(s). All these factors worsen the determinism of the systems and WCET analysis. Indeed, the execution time of software on one core depends on software executed on the other cores because of potential inter-core conflicts. The internal workings of the interconnect and how it prioritizes the requests are often part of the manufacturer's intellectual property, and they heavily impact on the amount and the duration of interferences. It may be difficult to determine an upper bound on their impact whatever the concurrent software even with full information on the design.

Characterizing the behavior of the interconnect in every possible situation in multi-core COTS is technically difficult. To overcome this problem the *Interconnect Usage Domain* can be defined as a set of constraints restricting the accesses to the interconnect. The usage domains give the possibility to treat the interconnect as black-box. The objective is to reach an acceptable characterization of the interconnect behavior to enable further analyses even with poor documentation on the behavior.

How these interference channels impact of the concurrent application also depends on the software architecture.

1.3.2 Parallelism basic concepts

There are several types of parallelism used in multi-core programming, we can fist distinguish between *Task Level Parallelism*, *Data Level Parallelism*, and *Instruction Level Parallelism* [Pat11]. Instruction level parallelism refer to overlapping the execution of instructions to improve performance. There are two largely separable approaches to exploiting this parallelism approach: 1. an approach that relies on hardware to discover and exploit the parallelism dynamically, and 2. an approach that relies on software technology to find parallelism statically at compile time. Data level parallelism focuses on distributing the data across different cores, which operate on the data in parallel. It is commonly applied on regular data structures like arrays and matrices by working on each element in parallel. Finally, Task level parallelism focuses on distributing tasks (functional units) across cores.

While Instruction and Data Level Parallelism are highly exploitable in FPGA and GPU, in this work we focus on Task Level Parallelism.

At system level, Task Level Parallelism can be partitioned into two other classification: *Asymmetric Multiprocessing* (AMP) and *Symmetric Multiprocessing* (SMP).

1.3.2.1 Asymmetric Multiprocessing

In this approach each core runs its own single-core aware application (a partition) as in figure 1.3. Scheduling inside a partition is sequential. The advantages of using this programming approach are:

- applications do not need to be multi-core aware; this simplifies the design.
- reduced need to mutual exclusion.
- interferences are mainly caused by shared caches, memory, I/O buses and concurrent access to shared devices.

The disadvantages are:

- all applications must be certified to the highest assurance level since they have full access to the processor;
- it can be hard to identify different single-core aware, independent applications; this can limit the use of cores to few of them;
- synchronization among applications running on different cores is more complex.

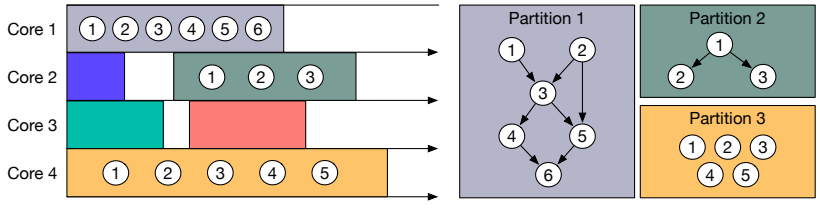


Fig. 1.3 Example of Asymmetric Multiprocessing

1.3.2.2 Symmetric Multiprocessing

In this approach each application has access of all cores. Threads inside the partition run concurrently and the Operating System typically controls cores and platform resources as shown in figure 1.4. The advantages of this approach are:

- More flexibility is allowed and better load balancing can be achieved.
- There is only one system component responsible for the partitioning.
- Different safety-levels are allowed in the system (mixed-critical systems).
- Application can still be completely isolated by the other, e.g. by disabling concurrent execution.
- Inter-process conflicts does not impact time and space partitioning as they occur inside the same partition.

The disadvantages come from the additional system-level software components:

- System level components responsible for the partitioning are complex and must be certified with the highest level of insurance.
- Synchronization effort can be higher.
- Due to the shared system software layer, an implicit coupling of unrelated threads cannot be completely avoided.

A careful design can limit the impact of these drawbacks.

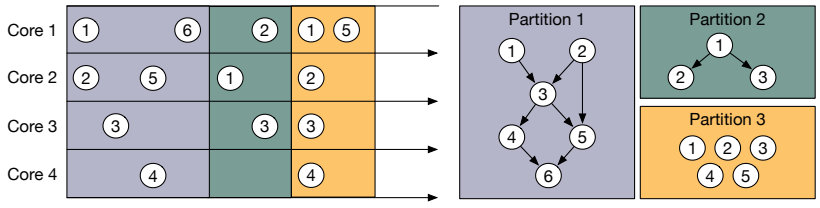


Fig. 1.4 Example of Symmetric Multiprocessing

1.3.2.3 Selected approach

The asymmetrical approach presents some difficulties in the demonstration of robust partitioning. It is interesting for very specialized applications, for example where, in a dual-core platform, one core is completely dedicated to I/O processing and the other core runs the application software. On the other hand, the symmetric approach needs to be considered to use at best the platform. All its drawbacks are manageable if the systems runs under control of the (trusted) operating systems and the design take into account all the possible issues. Moreover we assume that future embedded systems have to host, besides the critical applications, an increasing number of applications with high performance requirements but lower criticality.

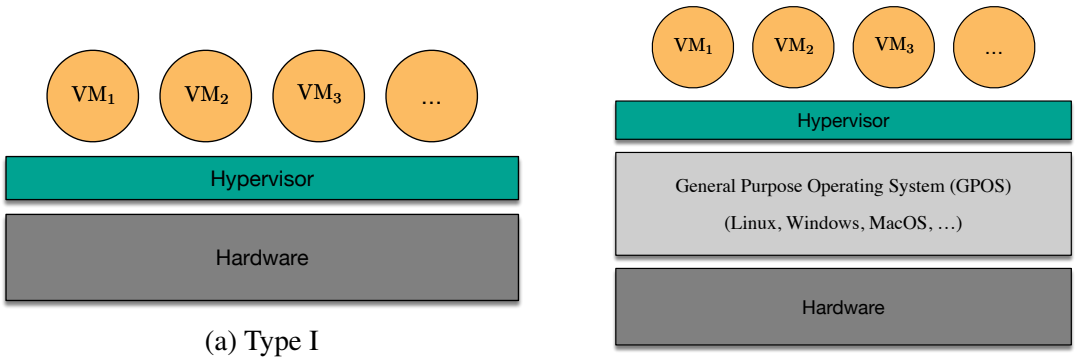
Implementing a mechanism to isolate different application inside a multi-core platform is crucial. A virtualization layer hosting several virtual machines can provide this service.

1.4 Virtualization in Embedded Systems

The main concept for the design of a mixed-critical system is, first and foremost, the demonstration of sufficient independence among software components. System virtualization, which is the abstraction and management of system resources, facilitates the integration of mixed-criticality systems [TCA13]. This approach results in independent virtual machines that are fully contained in an execution environment that can not affect the remaining system.

1.4.1 Overview

Platform virtualization refers to the creation of *Virtual Machines* (VMs), also called guest OS, running on the physical machine and managed by a *hypervisor*. Virtualization technology enables concurrent execution of multiple VMs on the same hardware (single or multi-core) processor. Virtualization technology has been widely applied in the enterprise and cloud computing field, however, in recent years, it has been increasingly deployed in the embedded systems domain. Virtualization for embedded systems must address real-time behavior, safety, and security such that it offers protection against external attacks and unintended interactions between the critical and non-critical components of the system. The hypervisor provides an isolation mechanism that can encapsulate an entire OS and applications into a Virtual Machine (VM).



(a) Type I

(b) Type II

Fig. 1.5 Hypervisor Types

1.4.2 Hypervisor types

In 1974 Gerald J. Popek and Robert P. Goldberg [PG74] classified hypervisors in two categories:

- *Level II hypervisor* as a software layer that runs on top of a General Purpose Operating System (GPOS) [Kle13] (fig. 1.5b). It takes advantage of the underlying Operating System services and hardware abstraction to enable the creation of virtual machines. However, the security of type II hypervisors is as robust as the host GPOS. Therefore, the hypervisor can be subverted by one of the security gaps in the host GPOS, thereby corrupting the entire system. Additionally, the host OS layer increases system complexity and overall code size, which is a major factor for resource-constrained embedded systems. As a result, type II hypervisors are not suited for most embedded systems.
- *Level I hypervisor* as a software layer that runs directly on the hardware platform (bare-metal) (fig. 1.5a. This approach avoids the complexity and inefficiency of GPOS, and can achieve a higher level of isolation for safety and security critical applications [Kle13].

The selected operating system, PikeOS, is a Level I hypervisor. It provides safe and security services through virtualization.

1.4.3 Virtualization Approaches

Virtualizing an operating systems requires placing a virtualization layer under the operating system to create and manage the virtual machines. As clarified below, virtualization is provided mainly in two ways [AG09]:

- *Full/Native Virtualization*. With this technique, guest operating systems are unmodified and unaware of the virtualization environment. Each virtual machine is provided with all services of the physical system (e.g. virtual BIOS, virtual devices, and virtual memory). Full virtualization usually employs binary translation techniques to trap-and-emulate non-virtualizable and sensitive system instructions or hardware assistance [Hor07]. However, the computational complexity of this technique results in an unacceptable performance level for embedded systems [Kle13].

- *Para-virtualization*. Unlike full virtualization, in para-virtualization, guest operating systems are modified to improve the performance of the hypervisor. These modifications are applied specifically to the guest OS kernel to replace non-virtualizable instructions and critical kernel operations with hypercalls that can request services directly from the hypervisor. These services represent system calls that are part of the OS kernel, and they execute with the highest privilege level in the system. Consequently, the hypervisor is the only software component to be executed in privileged mode. Para-virtualization overcomes the issues of full virtualization, and it is the only viable solution for embedded platforms that do not provide any hardware virtualization support[Kle13].

1.4.4 Microkernel-Based Hypervisor

In order to increase the robustness of the hypervisor, its size should be as small as possible. Microkernel-based hypervisors represent a thin software layer that runs as bare-metal in the highest privileged mode. It can provide strong isolation among the guest operating systems. This approach implements virtualization as a service on top of the trusted microkernel wich is the near-minimum amount of software that implements the needed mechanisms to implement an operating system. Therefore, each separate instance is as robust as the guest environment itself. Since the code size of the hypervisor is small, it is easier to verify and validate. Authors from Lockheed Martin [GWF10] presented a study towards the application of a Microkernel-Based Hypervisor architecture to enable virtualization for a representative set of avionics applications requiring multiple OS environments and a mixed-critical application.

1.4.5 Available Solutions

Many hypervisor solutions are available as either open-source or commercial products. For example, the Xen hypervisor has recently been ported to Xilinx Zynq Multi-Processor System-on-Chip (MPSoC) devices [xen]. Xen Zynq Distribution is released under the GNU

General Purpose License 2 (GPL2) but it is not designed for mixed-critical applications. Several commercial RTOS products that comply with the ARINC 653 standard [GZ12] are available, e.g. LynuxWorks LynxOS-178 [Lyn], Green Hills INTEGRITY-178B [INT], Wind River VxWorks 653 [VxW], Real- Time Systems GmbH Hypervisor [RTG], Tenasys eVM for Windows [eVM], National Instruments Real-Time Hyper Hypervisor [NIH], Open Synergy COQOS [COQ], Enea Hypervisor [Ene] etc.

Even though the offer is quite big, in this thesis we focus on PikeOS [Pikb] from SysGO AG. It is a microkernel-based Type one hypervisor certified for the most common standards (e.g. Do-178C, ARINC-653, ISO 26262 etc.) and has been designed for functional safety and Security requirements which makes it a suitable choice for our applications.

1.5 Model Based design

Applications are evolving to cover more and more complex functionalities. The increase in complexity is leading to an increase in the required throughput, and it is becoming a challenge for software developers. Model-based Design (MDB) appears as an excellent solution to cope with this complexity increase.

Model-Based Design is a model-centric approach to system development that enables system-level simulation, automatic code generation, and continuous test and verification. Rather than relying on physical prototypes (that can be very expensive) and textual specifications, Model-Based Design uses a model throughout development. The model includes every component relevant to system behavior—algorithms, control logic, physical components, and intellectual property (IP). MDB is being adopted in all areas of engineering; moreover, certification has matured considerably in the last decade attracting considerable interest from companies; recent studies [BMOS10] have shown that the application of model-based certification and formal verification can be a practical and cost-effective solution against certification requirements.

Examples of available commercial tools are Simulink®[Sim], SCADE Suite®[Sca], LabVIEW®[Lab] and SystemModeler®[Mod]. Open source and research tools include Scicos [Sci] and Ptolemy[EJL$^+$03]. In this work we use Simulink which is the *de-facto standard* and provides mature tools for design, simulation, and code generation.

The model includes every component relevant to system behavior-algorithms, control logic, physical components and intellectual property (IP). This make MBD the perfect candidate for a complete work-flow that goes from the system development, verification, validation and

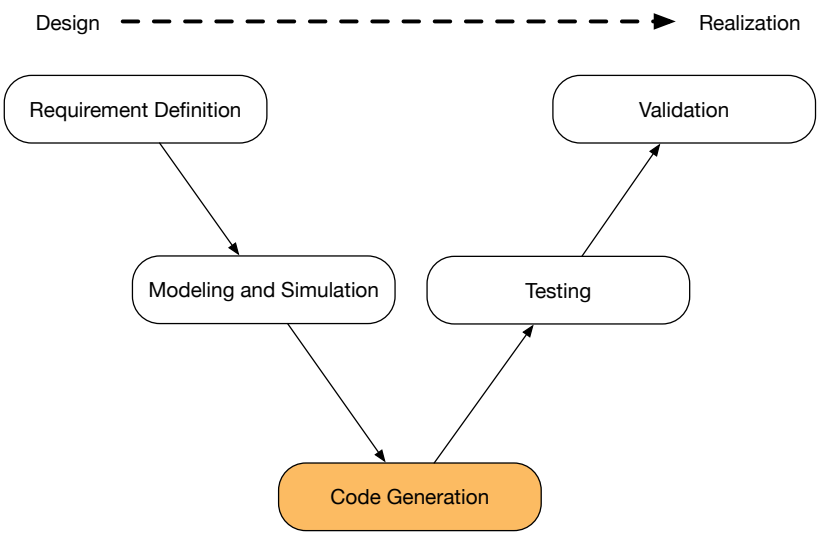


Fig. 1.6 Model-Based Design Flow Chart

deployment. A typical design flow in MDB is shown in figure 1.6. The traditional embedded system development process follows the standard *V-shaped* lifecycle. V-cycle splits the product development process into a design and an integration phase. The Code Generation step is the turning point of the process. This work focus on this step.

1.6 EMC^2

This work fits inside the European EMC^2 - "Embedded Multi-Core Systems for Mixed Criticality applications in dynamic and changeable real-time environments" project [emc]. The objective of the project is to foster changes through an innovative and sustainable service-oriented architecture approach for mixed-criticality applications; the project bundles the power of 98 partners from 19 European Countries and 100 millions of euro in budget.

Within the EMC^2 project the objective was to demonstrate the possibility of using a Model-Based Design approach to assist the design and the implementation of mix-critical applications running in multi-core platforms. For that purpose, we selected a typical aerospace use case: motor drive control. This is a widely known application that is used for example in the control of the actuators for the primary and secondary flight control. The platform selected to implement this use case was the Xilinx ZedBoard™[zed] based on the Xilinx Zynq®-7000 All Programmable System-on-Chip (SoC). The board is equipped with dual-core ARM®Cortex-A9 processors and an Artix-7 FPGA which adds more complexity to the design and more flexibility. For the motor control we used the *FMCMOTCON2* [FMC]

evaluation kit from Analog Devices (figure 1.7 which provide a complete motor drive system including a stepper motor, a control board, a low voltage drive board and a Dynamometer drive system.

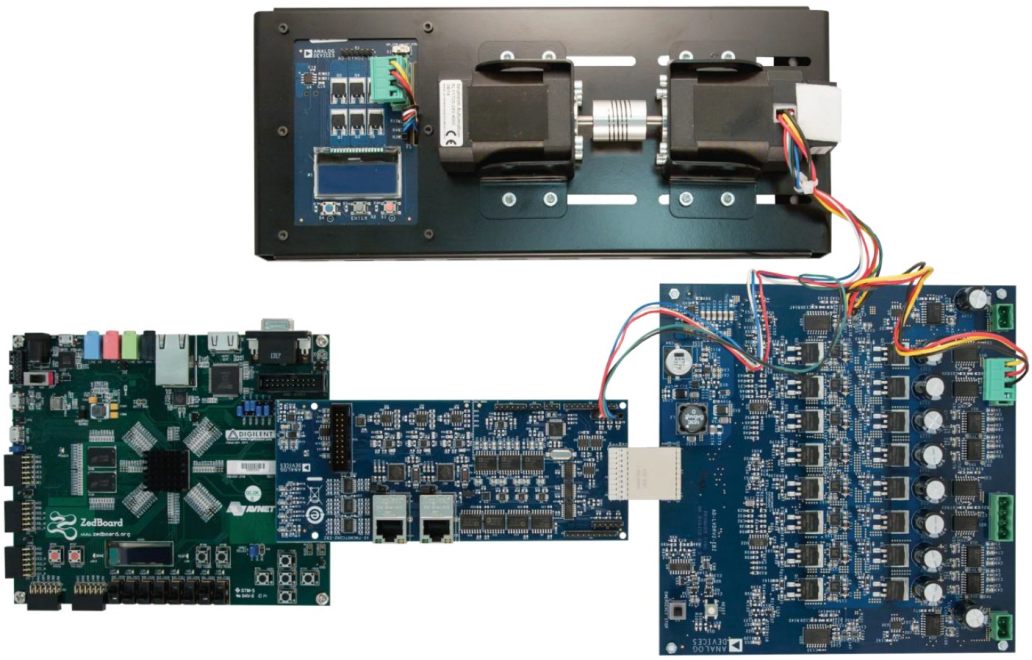


Fig. 1.7 Analog Devices FMCMOTCON2 Evaluation Kit

Chapter 2

Frameworks and tools

2.1 PikeOS

A trusted Operating System, capable of providing isolation among partitions is crucial for mixed-criticality systems. PikeOS [Pikb] is a commercial operating system from SYSGO AG, designed in early 2002 to address the needs of a kernel targeting safety and security applications. It uses a separation micro-kernel architecture and provides a hypervisor model at its core. Moreover, PikeOS hypervisor has been certified for all relevant certification standards including DO-178C, IEC 61508, EN 50128, EN 62304, and ISO 26262 coming from the fields of Aerospace/Defense, Automotive, Transportation and Industry/IoT.

The micro-kernel architecture allows the OS to be used in resource constrained devices like embedded systems. It supports single and multi-core processor architectures with both models: AMP (Asymmetric Multi-Processing) and SMP (Symmetric Multi-Processing).

PikeOS includes *CODEO*, an Eclipse-based IDE that provides several functionalities such as guided configuration, compilers, assemblers, remote debugging, application deployment, target monitoring, and timing analyses.

2.1.1 Hypervisor general overview

The PikeOS Hypervisor offers a performance optimized para-virtualization, meaning that the OS presents a software interface to virtual machines that are aware of the virtualization level.

The PikeOS Microkernel consists of a generic part (CPU architecture dependent) called *Architecture Support Package* (ASP) and a platform dependent part called *Platform Support Package* (PSP): together they form the *Board Support Package* (figure 2.1). The PikeOS

Microkernel (with the ASP) is the only component that runs with supervisor privileges. It provides:

- Hardware abstraction
- Spatial and temporal separation among partition
- Communication primitives

The *PikeOS System Software* (PSSW) component is the first user space application launched by the PikeOS Microkernel, it performs some initialization and, at run-time, acts as a server providing services like communications, file system access, health monitoring and partition/process management to the applications.

2.1.2 Personalities

In PikeOS guest operating systems, runtime environments or API are called *personalities*. They run on top of the hypervisor in a non-privileged mode as shown in fig 2.1. Personalities included are:

- AUTOSAR
- Android
- Embedded Linux
- Native PikeOS

Moreover, regarding the runtime environments:

- Ada
- AFDX (Avionics Full-Duplex Switched Ethernet)
- ARINC-653 (Avionics Application Standard Software Interface)
- Certified POSIX
- RTEMS (Real-Time Executive for Multiprocessor Systems)
- Real Time JAVA

Other personalities are supported through SYSGO partners.

These guests are contained inside a Virtual Machine with their memory space, resources and application set. Applications hosted on one virtual machine run completely independently of those in the others.

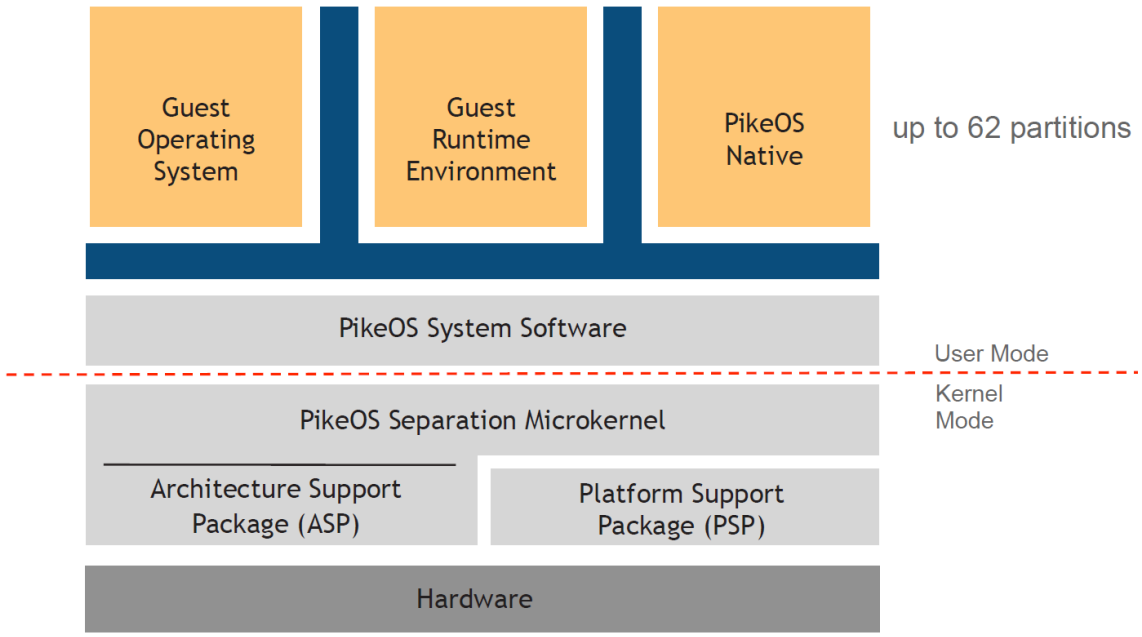


Fig. 2.1 PikeOS System Architecture

2.1.3 Resource Partitions

Resource partitions are one of the basic security mechanisms to support multiple virtual machines on top of PikeOS. They can be thought of as containers within which applications execute. Partitions define the system resources that their applications can use and provide protection domains between different applications. For this reason, PikeOS partitions are also referred to as *resource partitions* (fig.2.2). An application running in one partition is completely unaware of applications in other partitions, and cannot access resources to which it does not have explicit access permission. A partition can be stopped, restarted or reloaded with different applications without affecting other partitions. PikeOS supports up to 63 resource partitions. All resource partitions are created by the kernel at boot time.

Resource partitioning is enforced by using the MMU to control access to the available resources. To access a resource, each hardware device is somehow represented by a physical address. Thanks to this, resource partitioning can be realized by forcing the MMU to map a certain memory area into a partitions virtual memory space and allow or deny read/write access to this memory space. The configuration of the MMU is done statically at system startup in the PikeOS microkernel and is not modifiable at run-time, also ensuring security. In summary, the separation makes sure that errors occurring in one partition cannot propagate to another partition.

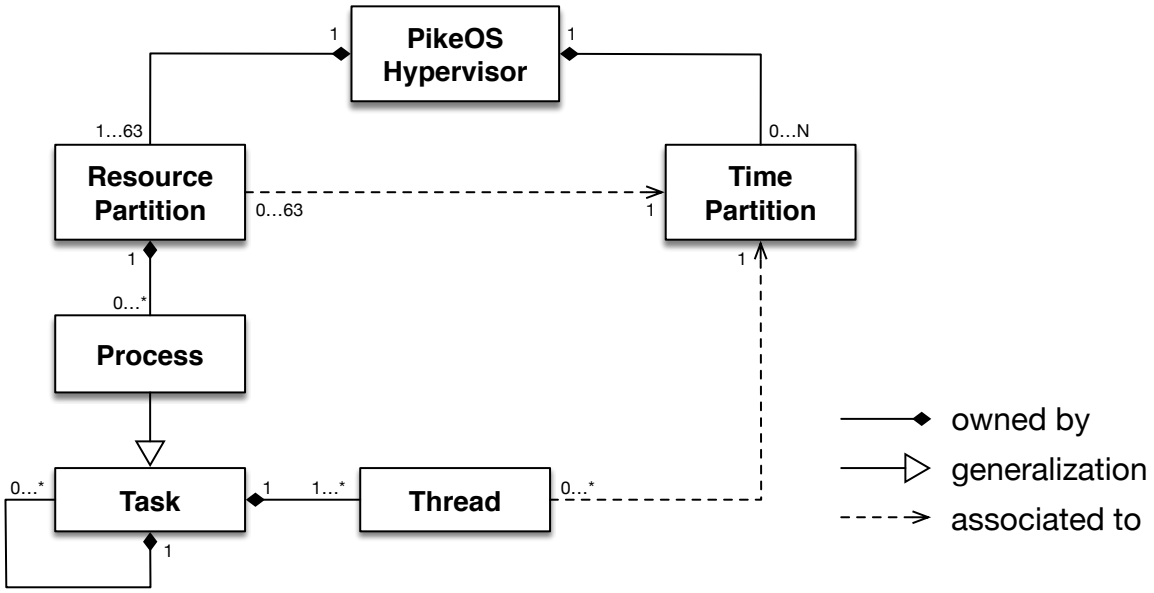


Fig. 2.2 PikeOS Execution Entities

2.1.4 Processes and Tasks

Inside a resource partition we can find different *processes* that, from the kernel point of view, are PikeOS *tasks* (see fig.2.2). Each process has its access rights to resources, including CPUs. The set of processors on which task threads can be scheduled, can be specified in the CPU mask attribute.

Moreover, each PikeOS task has a set of abilities associated with it, which the kernel uses to verify that the task's threads have sufficient permissions to use certain kernel services. An ability is said to be enabled if the task is allowed to use the corresponding kernel service, and it is said to be disabled if the task is denied use of that service. Abilities are a task property, so all threads within a task share the same set of abilities.

2.1.5 Threads

Threads are the schedulable entities of a task. Each thread has it execution priority and, again, the affinity mask that express on which processors the thread can be scheduled. A thread may assume one of the following states (fig. 2.3):

- *Inactive*. Threads in the INACTIVE state are never scheduled and are not valid targets for communications. When a task is activated, all threads are in the INACTIVE state. Also, a thread delete can put the target thread in this state.

- *Current*. A thread which owns the CPU is in the state. This thread (the current thread) can either execute user-level code or execute code inside the PikeOS kernel.
- *Waiting*. A thread is in this state if is waiting for a communication, an event, a resource.
- *Ready*. A is tread in this state if it is ready to continue its execution. Therefore is eligible to become the current thread.
- *Stopped*. A thread in this state is never eligible to become the current thread.

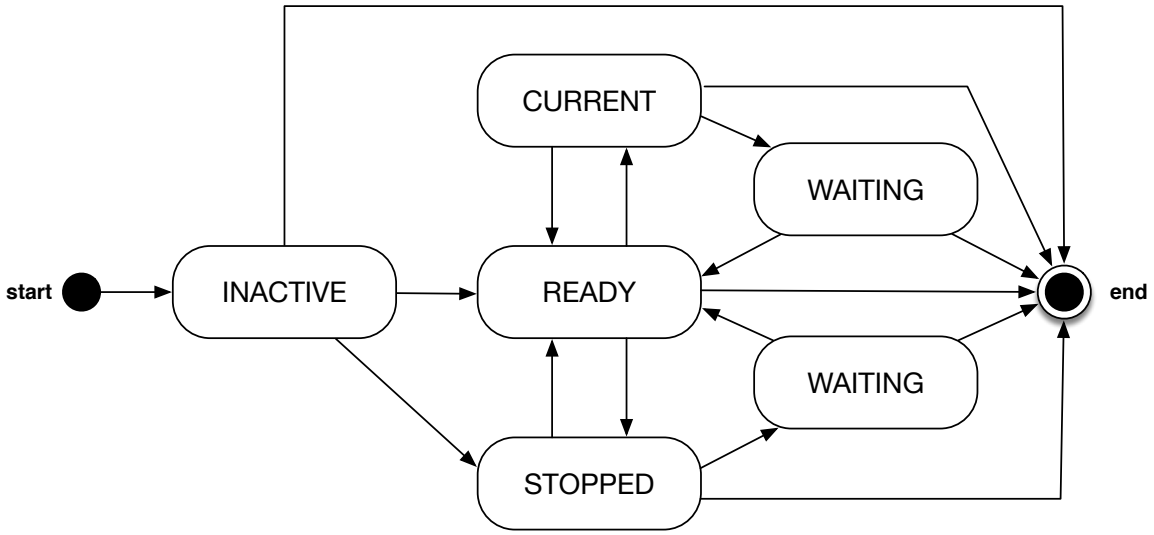


Fig. 2.3 PikeOS Thread States

A thread is put into the STOPPED state using p4_thread_stop().

2.1.6 Time Partitions

The PikeOS partition scheduler uses a combination of priority-based and time-driven scheduling.

In contrast to the ARINC 653 standard, PikeOS scheduling uses a n-to-one assignment of resource partitions to time-partitions. This means that one or more resource partitions can be assigned to one time-partition. In this work, time partitioning is used to allocate a given amount of CPU time to each resource partition (even though PikeOS allow a many-to-1 mapping, our approach is fully ARINC 653 compliant). In this case, there is a *one-to-one relationship between time and resource partition*s. This type of configuration is illustrated in figure 2.4. Moreover, a PikeOS partition may host more than one task (Process), but in this work, each task is assigned to one and only one resource partition.

Each Time Partition consists of one or more *Time Windows* representing time intervals allocated to the Time Partition. Each Time Window can be marked as the start of a new period, allowing a static definition of preemption between partitions.

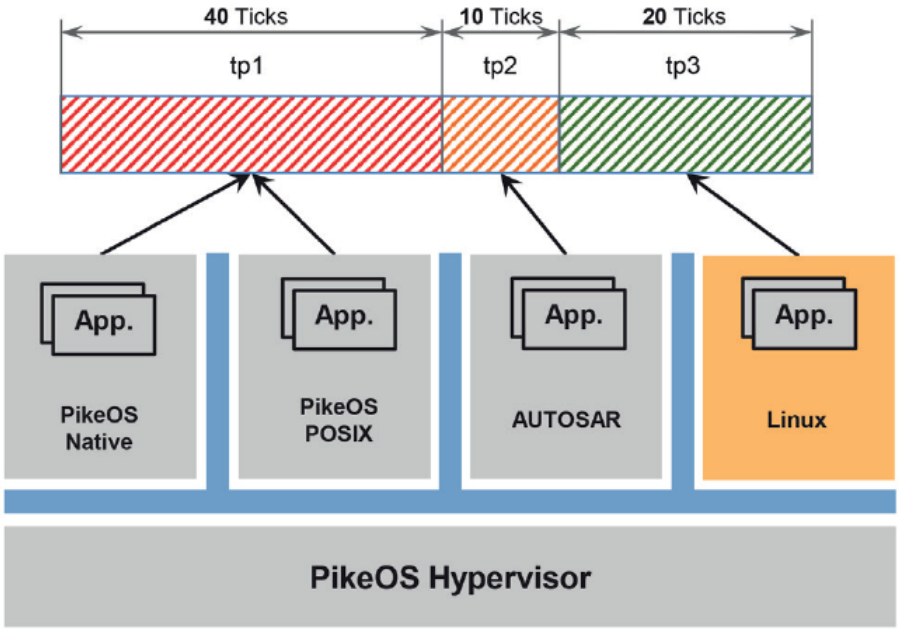


Fig. 2.4 PikeOS Time Partitioning

It is worth mentioning that, in contrast to the ARINC 653 standard, there is one special partition that is active at all times. This time partition is referred as the *background partition*[1], TP_0, whereas the currently active time-switched partition is called the *foreground partition*, TP_i with $i = 1,...,N$.

2.1.7 PikeOS Scheduler

Each time partition has its own priority, in addition to that, threads also have a priority attribute. Whenever both the foreground and background partitions have active threads, the thread to be executed is selected according to its priority. Figure 2.5 shows the principle of operation: each time partition is represented as a priority-sorted list of FIFO ready queues. Each of them delivers its highest priority ready thread and of these, the highest priority one from either TP_0 or the currently selected TP_i gets chosen for dispatch.

[1]Background time-partition in parallel to the active time-partitions is an invention by SYSGO and under active patents.

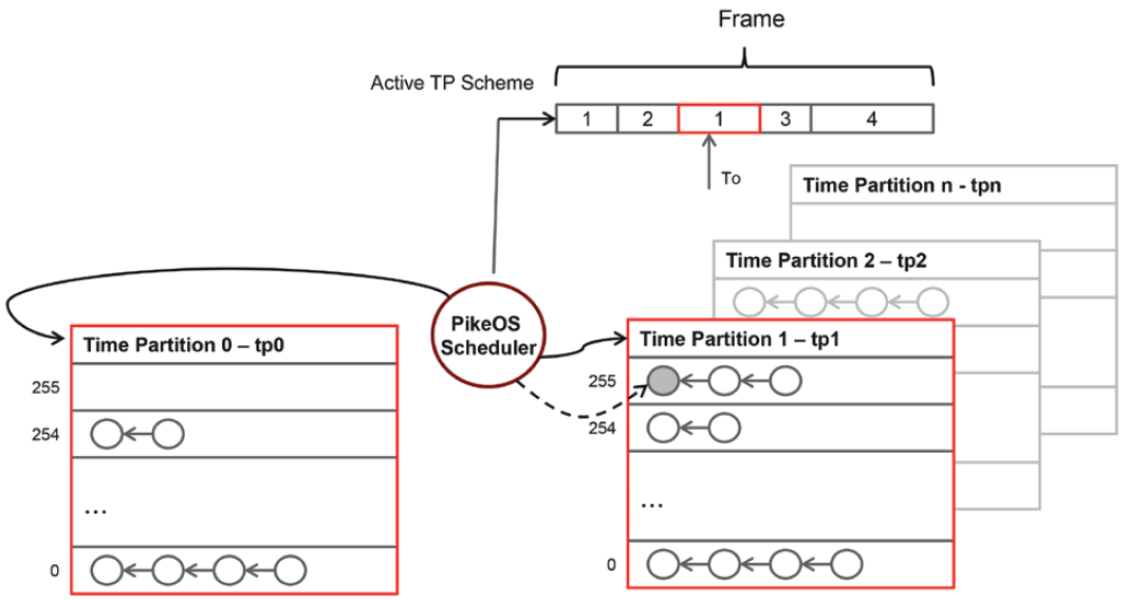


Fig. 2.5 PikeOS Time Partition Scheduler

Partitions periodically receive guaranteed time slices. But, whenever one partition completes its job prior to having consumed its entire time slice, the unused time automatically falls back to the next lower priority thread in the background time partition. Thus, by running explicitly non-real-time applications with lower priority threads in the background time partition (TP_i), non real-time applications can receive spare computational resources that were assigned to, but not consumed by real-time application in the foreground partition (TP_i). This simple approach allows to improve the throughput of safety-critical systems. Figure 2.6).

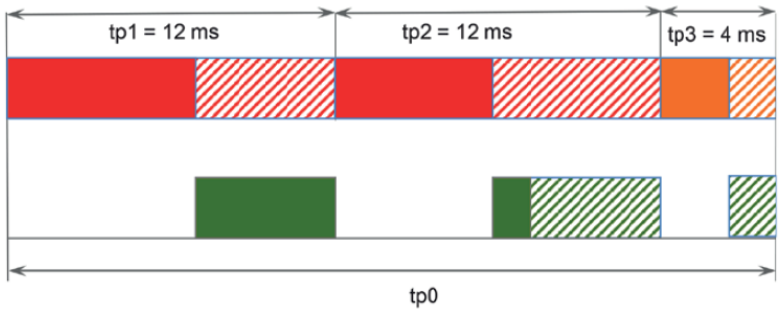


Fig. 2.6 PikeOS Scheduling with low priority background application

In a somewhat opposite scenario, it is possible to define threads in the background domain, which can override time partitioning by assigning them a priority above those of the foreground domain threads (fig. 2.7).

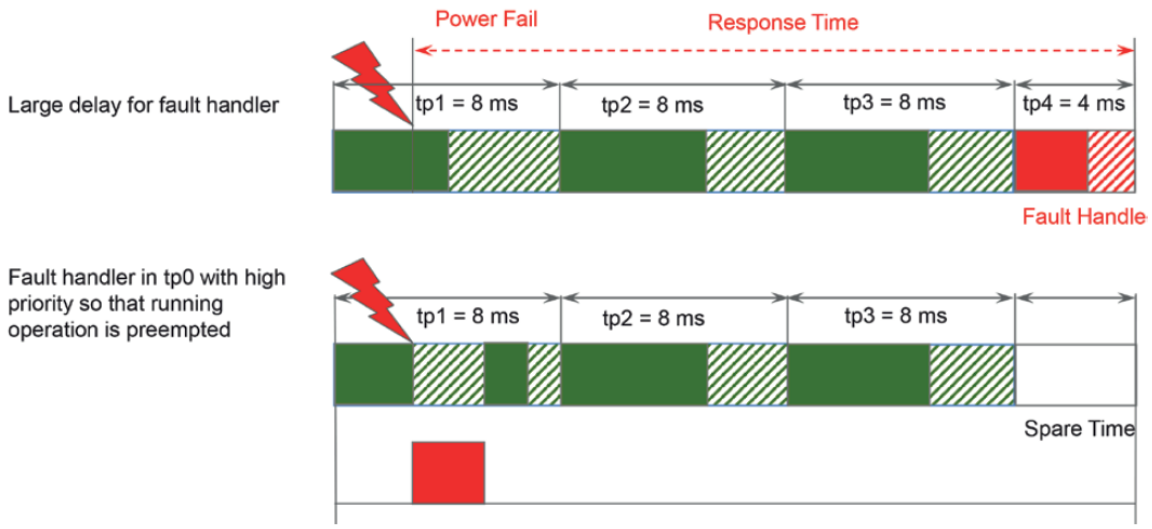


Fig. 2.7 PikeOS Scheduling with with high priority background application

The upper time-diagram in Figure 2.7 has a fault handler, which can be executed in a dedicated time-partition TP_4. In case of a failure the response-time of the fault-handler can be high if the error happens right after the fault handler time-partition has ended. By assigning the fault handler to TP_0, the handler is starter immediately, as it has the highest priority among all active threads (lower time-diagram of Figure 2.7). Clearly, such high priority threads must be considered as trusted code from the point of view of the threads that they can preempt.

PikeOS support the possibility to change, at runtime, the Time Partition schedule. Each schedule is expressed (statically) inside the configuration file, and it is called *scheme*. Schemes allow the system designer to adjust the time partitioning to handle exceptional situations rather than having to design a single time partitioning scheme which will suit all possible scenarios. The following XML code fragment shows a configuration with two schemes named "SCHED_BOOT" (which is the one loaded at startup) and "SCHED_POWERFAIL".

```
<ScheduleTable>
  <ScheduleScheme Name="SCHED_BOOT">
    <WindowTable>
      <Window Identifier="0" TimePartitionID="1"
        Start="0" Duration="3" Flags="" />
      <Window Identifier="1" TimePartitionID="2"
        Start="3" Duration="3" Flags="" />
      <Window Identifier="3" TimePartitionID="3"
        Start="9" Duration="3" Flags="" />
    </WindowTable>
```

```
11    </ScheduleScheme>
12    <ScheduleScheme Name="SCHED_POWERFAIL">
13      <WindowTable>
14        <Window Identifier="0" TimePartitionID="1"
15          Start="0" Duration="1" Flags="" />
16      </WindowTable>
17    </ScheduleScheme>
18  </ScheduleTable>
```

Where the duration is expressed in *Ticks*. Each Time Windows can be marked as a new period starting point just adding *VM_SCF_PERIOD* its flags field.

The PikeOS hypervisor provides means to invalidate instruction caches and TBLs and to flush the data cache between time partition switches (fig. 2.8). This ensures that caches and TLBs are in a defined state when a partition starts its execution. The cache/TLB flush and invalidate operation takes place during the time partition switch, so it will introduce a delay for the partition activation and thus cause a jitter. A possible approach to know the duration [Pika] of this delay is to define a small (but big enough) time partition window, which is allocated to an unused time partition ID and to insert this before the time critical application. This eliminates the jitter of the time critical application.

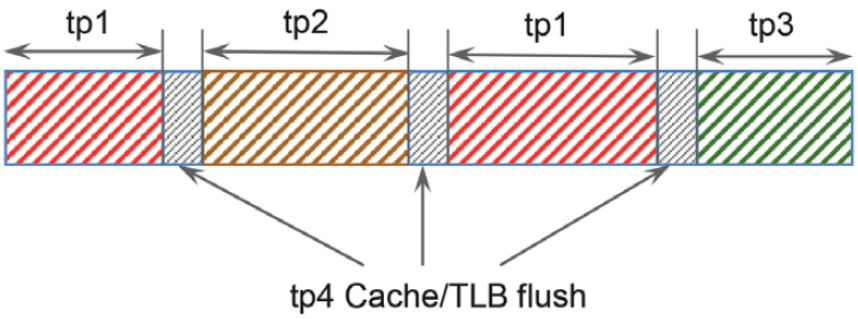


Fig. 2.8 PikeOS Cache and TLB Flushing

2.1.8 Timeouts

The PikeOS kernel provides a logical timebase for system time and timeouts. The system time value represents the elapsed time since startup. The resolution of the system time is platform- and configuration-dependent.

The PikeOS kernel provides different types of timeouts, in this work only *Normal Timeouts* are used. They are timeouts that specifies a point in time with respect to the system

time. Each timeout can be absolute or relative: an absolute timeout specifies the timeout as a system time value, while a relative timeout specifies the timeout as a difference relative to the current system time. Valid values are:

- `P4_TIMEOUT_NULL`
- `P4_TIMEOUT_INFINITE`
- numeric values

Expired timeouts are evaluated with a minimum frequency corresponding to the period of the system tick. Timeout expiration is checked before every rescheduling decision. A rescheduling is always evaluated when a system tick is processed by the kernel. However, the effective expiration depends on whether a time window of the caller's time partition is active or not. If the caller's time partition is active, the timeout expires immediately. If no window of the caller's time partition is active, the timeout does not expire when this time point is reached. The caller remains blocked, and the effective timeout expiration occurs when the caller's time partition is next activated. In the first case, the thread may be scheduled immediately if it has the highest priority in the system. In the second case, the service call behaves as if the timeout did not expire.

2.1.9 Communication primitives

PikeOS provides message based communication through statically configured communication channels. A communication channel is a link between two ports, a *source port* and a *destination port*. The data flow is from the source port to the destination port – the view point is thus from the channel between the ports. The source and destination ports may be located within the same partition, within two different partitions, or within one partition and a port provider. In this work, we use communication ports mainly as an inter-partition communication mechanism.

The PikeOS kernel provides two different types of port that allows different partitions to communicate, these are:

- *Sampling Ports*. They use a single message buffer that is atomically updated by a write operation; they also maintain the age of a message measured from the time of reception, which can be compared to the port's refresh rate (this is used to derive a message validity);
- *Queuing Ports*. Operate as a FIFO buffer with a maximum message size.

Communication via ports guarantees that either an entire message is written to or read from a port or no data is transferred at all. Queuing ports provide a blocking API with a message queue and with an optional timeout. Messages written to a queuing port are buffered within the PikeOS framework, i.e., they are copied twice: once on the write and once on the read. This ensures that a message that has been successfully written to a queuing port and cannot be corrupted by any user application, particularly not the sending one when it changes the send buffer. It is not an error for a port to be unconnected to another port. Instead, the port will behave as if the other side is completely unresponsive.

Every channel must have exactly two endpoints: one source and one destination port. A queuing port can only belong to one channel. A destination sampling port can only belong to one channel, while a source sampling port can be referenced in an arbitrary number of channels. This allows a configuration where one source sampling port is connected to many sampling destination ports (1-to-n multicast). Summarizing, the main characteristics of a communication port (either Sampling or Queuing Port) are the following:

- Message Oriented (one message per transaction).
- Channel establish port-to-port communication.
- Queuing ports support blocking with timeouts, sampling ports provide the last valid data with validity (freshness) check.
- Suitable for medium size data (data are copied twice).

2.1.10 User Space Synchronization

For synchronization between threads, PikeOS supports all the most common mechanisms including: *Mutexes*, *Condition Variables*, *Barriers*, *Semaphores* and *Spinlocks*. All of these synchronization objects allow the calling thread to acquire and release objects and do not consume any kernel memory resources. Because we are focused on non-preemptive and highly deterministic behavior, the spinlock mechanism is the one that fits better.

A spinlock causes the calling thread trying to acquire it to simply wait in a loop (*spin*) while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of this mechanism leads to a busy wait. Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are very efficient if threads are likely to be blocked for only short periods and nonpreemptive behavior is accepted. In PikeOS, three API call are available for spinlocks handling:

- `void p4_spin_init(P4_spin_t *lock)`: Initialize a spinlock to unlocked state.
- `void p4_spin_lock(P4_spin_t *lock)`: Acquire spin lock, spin in the contended case.
- `void p4_spin_unlock(P4_spin_t *lock)`: Release spin lock.

2.1.11 Integration project

The configuration of a PikeOS system is mainly done in the integration project, i.e., configuration of all partitions, communication, permissions, etc., is done by the system integrator. The top-level tool for configuration is the Project Configurator. This tool is available as a command like tool as `pikeos-projectconfigurator`, and as an Eclipse plugin as part of the graphical CODEO tool.

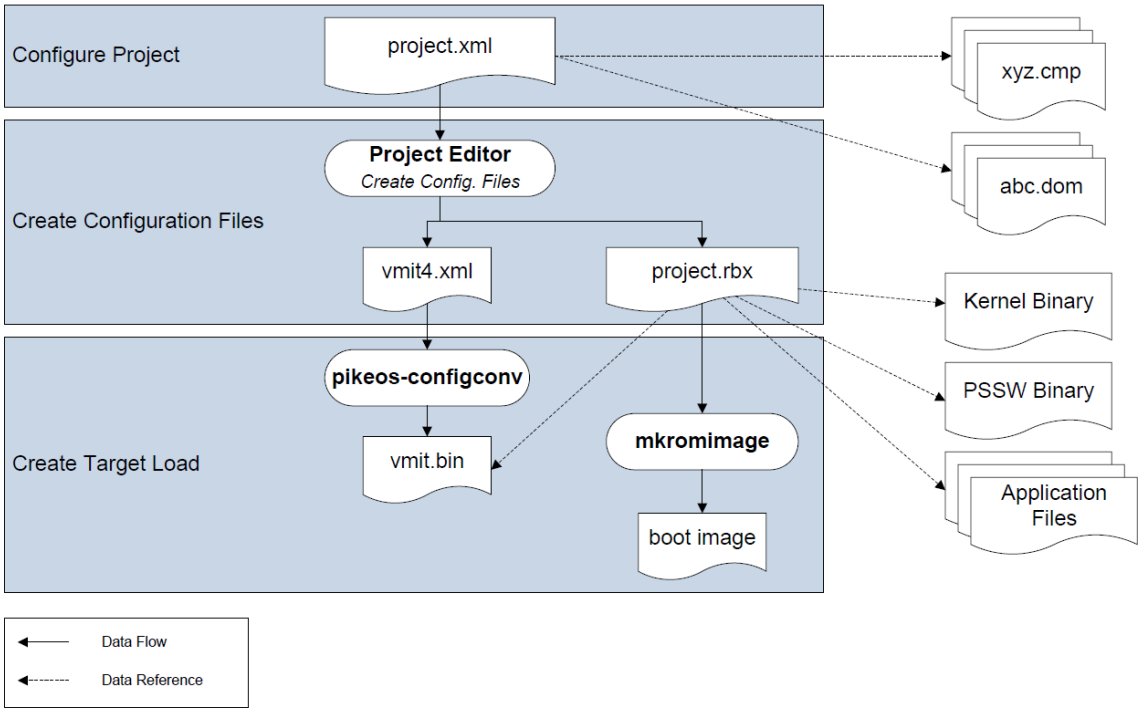


Fig. 2.9 PikeOS Project Configuration Overview

All the PikeOS relevant information are stored inside the `project.xml` file. However, for lower level tools, the meta information inside this if file are not fully required; for that reason, the PikeOS Project Configurator converts the `project.xml` file into the lower level configuration files used by the PikeOS system:

- *ROM Boot Extended Configuration (RBX)* contains a description of the kernel, the memory region setup, the ROM file system and the file system;
- *Virtual Machine Initialization Table (VMIT)* contains all of the configuration settings of the PikeOS system and is used by the kernel to set up time and resource partitions as well as communication channels.

Both RBX and VMIT files are stored in XML format, but are converted to binary format before being loaded into the system.

2.2 Simulink and Embedded Coder

Simulink®[Sim], developed by *The MathWorks Inc.*, is a graphical programming environment for modeling, simulating and analyzing multidomain dynamic systems such as signal processing, control and communication applications. It supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Thanks to its features, it is the *de-facto* standard in model-based design.

In order to generate C/C++ code from Simulink models, we can use *Simulink Coder*®, formerly *Real-Time Workshop*®. It supports code generation from both Simulink diagrams and Stateflow charts. Moreover; with Embedded Coder®, we can generate C/C++ code optimized to be used on embedded processors, on-target rapid prototyping boards, and microprocessors. It also provides traceability reports, code interface documentation, and automated software verification to assistance support standards such as DO-178 and ISO 26262 during development. It also supports external mode to connect the Simulink block diagram to the relative application that runs the model on the target hardware. The block diagram becomes a user interface to the real-time application, and by changing parameters in the Simulink blocks, the parameters in the real-time application also changes.

2.2.1 Synchronous-Reactive Model of Computation

Simulink implements a Synchronous-Reactive (SR) model of computation (MoC), and is used for the representation of systems in which components evolve synchronously, a fundamental concept for the modeling of concurrent systems [Pto14].

Synchronous-Reactive models are networks of Mealy-type blocks, eventually clustered into subsystems and can be continuous, discrete or triggered [CMDN15]. Continuous blocks process continuous-time signals and produce as output other continuous-signal functions

according to the block description, typically a set of differential equations. Discrete-time blocks are activated at periodic-time instants and process input signals producing outputs signals and state updates. Finally, triggered blocks are only executed on the occurrence of a given event (a signal transition or a function call).

Since we are interested in automatic code generation let us focus on discrete-time blocks which are (eventually) activated at each tick-time t. Let us denote a generic block i as B_i, the input vector at tick-time t as $\vec{u}_i(t)$ and similarly the output vector as $\vec{y}_i(t)$. Each block maintains a state vector $\vec{s}_i(t)$, and the behavior of B_i is represented by its *output update* function

$$\vec{y}_i(t) = f(\vec{u}_i(t), \vec{s}_i(t)) \tag{2.1}$$

and a *state update* function

$$\vec{s}_i(t+1) = g(\vec{u}_i(t), \vec{s}_i(t)) \tag{2.2}$$

Often, the two update functions are considered as one:

$$(\vec{y}_i(t), \vec{s}_i(t+1)) = h(\vec{u}_i(t), \vec{s}_i(t)) \tag{2.3}$$

A fundamental part of the model executable semantics are the rules dictating the evaluation order of the blocks. A block has a *direct feedthrough* relationship between ports when the output port is controlled directly by the value of any input port signal. Any block with direct feedthrough relationship cannot execute until the block(s) driving its input has (have) executed. The set of topological dependencies implied by the direct feedthrough relationships defines a partial order of execution of the blocks. When the simulation starts, Simulink computes a total order of block execution compatible with the partial one.

2.2.2 Real-Time Workshop Code Generation Process

The Real-Time Workshop (RTW) simplifies the process of building application programs. One of its features is automatic program building, which provides a standard means to create programs for real-time applications in a variety of host environments. The functioning of RTW is fully customizable. The overall model build process is depicted in figure 2.10.

The initial stage of the code generation process is to analyze the source model; the resulting description file, `model.rtw` contains a *compiled* representation of the model including a hierarchical structure of records describing systems, blocks, and their connections. This intermediate model description feeds the *Target Language Compiler*® (TLC) that interprets it and guide the C/C++ code generation. It is also possible to write a *Template Make File*

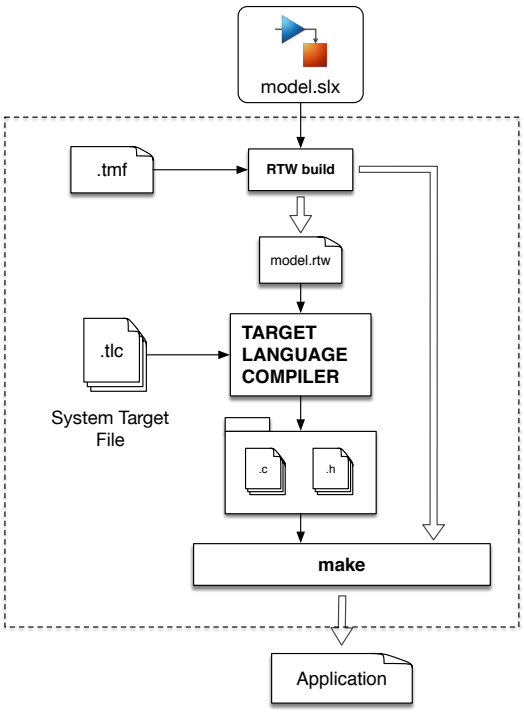


Fig. 2.10 Real-Time Workshop Model Building Process

(TMF) that define how to generate a Makefile for the actual compilation of the generated source files.

The RTW file is a language-independent format, basically stored as an ASCII file. For example, consider the model depicted in figure 2.11. An excerpt of the generated `model.rtw` is in figure 2.12.

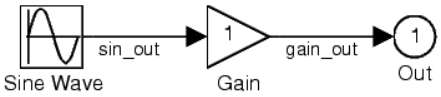


Fig. 2.11 Model Example

2.2.2.1 Code Architecture

When a model or a block is generated, the RTW follows a general structure for the generated code. Blocks have inputs, outputs, parameters, states, and other general properties. All Block inputs and outputs are written into a block I/O structure (`rtXX`) and instantiated as global variables. Figure 2.13 shows the general block data mappings. This structure applies

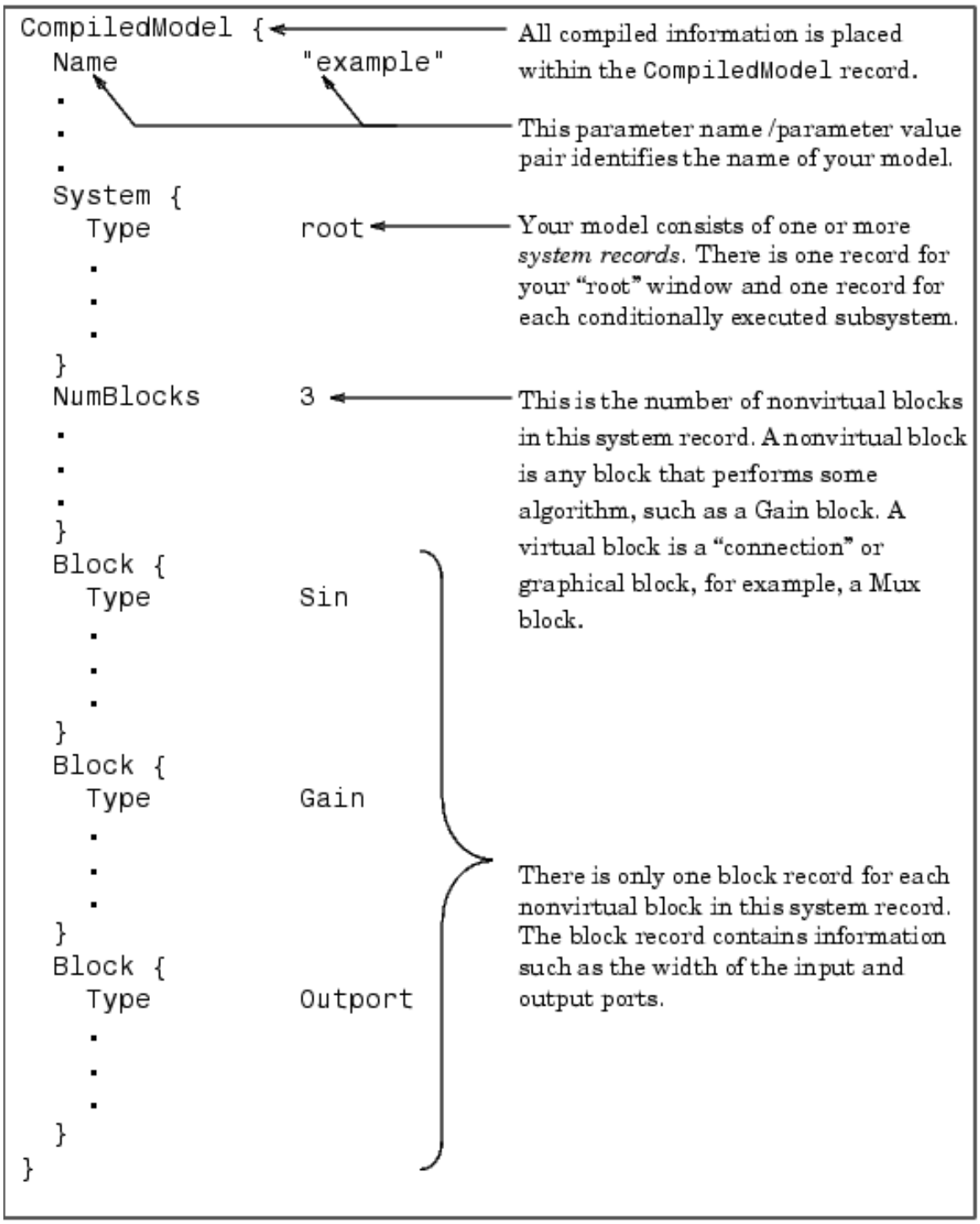


Fig. 2.12 RTW File excerpt

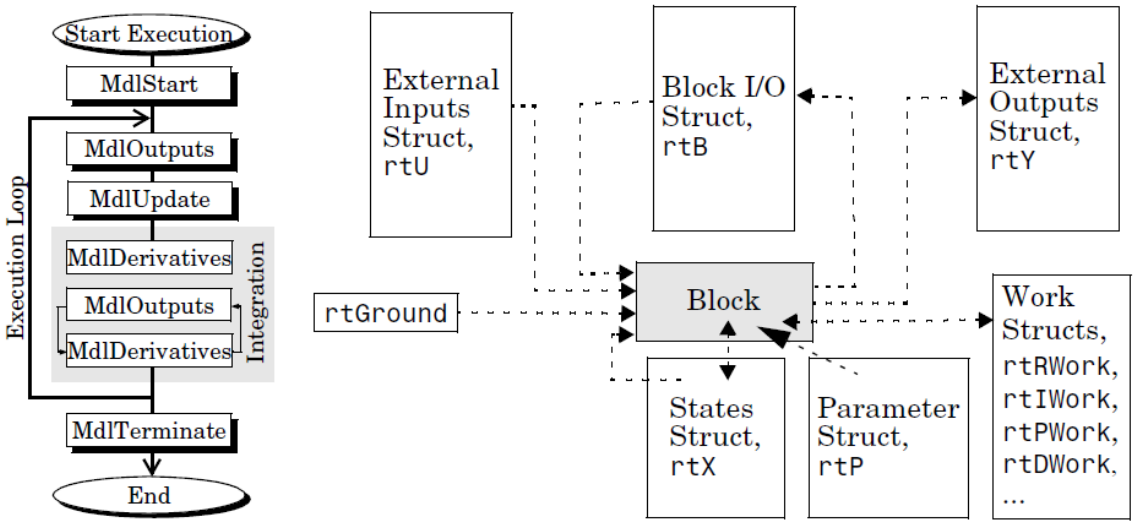


Fig. 2.13 TLC Block code structure

also to the generated model. Out of the generation process, three *Entry Point Functions* are available:

- *model_initialize*. Model initialization entry point.
- *model_step*. Step routine entry point.
- *model_terminate*. Termination entry point.

The structures and the function are stored in the source code. Embedded Coder creates a build folder in the working folder to store the following generated source code:

- *model.c*. Contains entry point functions implementing the model algorithm.
- *model_private.h*. Contains local macros and local data that are required by the model and subsystems. This file is included in the model.c file as a `#include` statement.
- *model.h*. Declares model data structures,a public interface to the model entry points and data structures. This file is included in the `model.c` file.
- *model_types.h*. Provides forward declarations for the data structure and the parameters data structure.
- *rtwtypes.h*. Defines data types, structures, and macros required by Embedded Coder generated code.

- *Optional files*. Sometimes additional source or header files are required by the generated code. These files are placed in the build folder.

2.2.3 Concurrent Workflow

Simulink provides a way to address the challenge of designing systems for concurrent execution through its *Concurrent Workflow*. It uses the process of partitioning, mapping, and profiling to define the structure of the parallel application. Partitioning enables to designate regions of the model as tasks. Mapping enables to assign partitions (tasks) to processing elements such as FPGAs or CPUs. Then Profiling simulates the deployment of the application under typical computational loads.

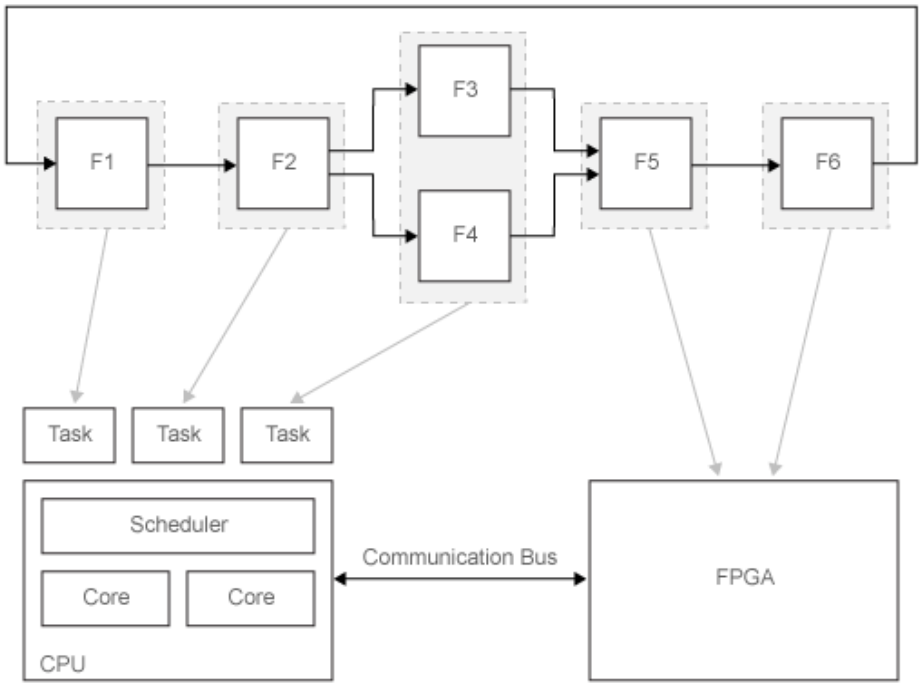
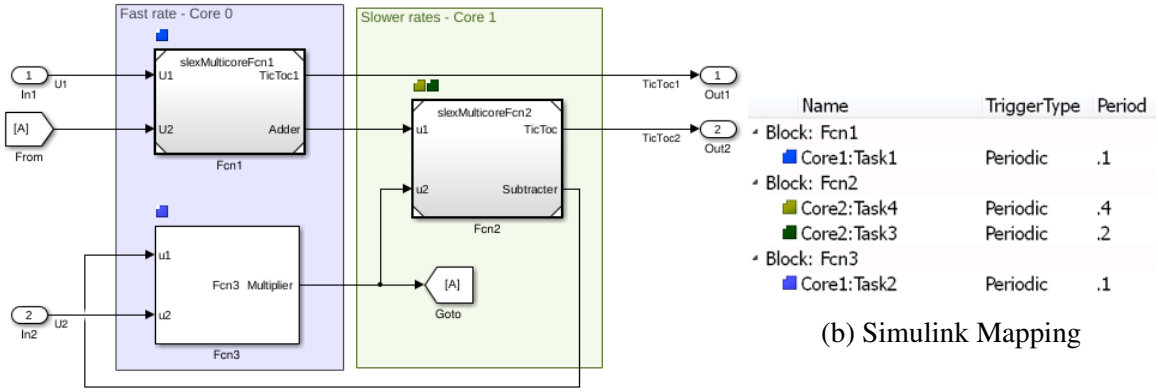


Fig. 2.14 Simulink Multi-Core Partitioning

Partitioning is the first step for defining the application's structure. There are two ways to partition the model for running on individual processing nodes. The automated way of creating tasks and mapping them to processing nodes is called *implicit partitioning*. Simulink partitions the model based on the sample times of blocks. Each sample time in the model corresponds to a partition, and all blocks activated with the same rate or sample time belong to the same partition. Then these partitions are mapped to tasks and their priority is assigned according to their rate. Therefore, implicit partitioning assumes the hardware architecture



(a) Simulink Explicit Partitioning

Name	TriggerType	Period
Block: Fcn1		
Core1:Task1	Periodic	.1
Block: Fcn2		
Core2:Task4	Periodic	.4
Core2:Task3	Periodic	.2
Block: Fcn3		
Core1:Task2	Periodic	.1

(b) Simulink Mapping

Fig. 2.15 Simulink Concurrent Workflow (by *The Mathworks*)

consists of a single multi-core CPU and that the Operating System scheduler handles all the execution according to task priorities.

Another way to specify partitions is to use *explicit partitioning*. In explicit partitioning, partitions are created in the root-level model by using *referenced models* (fig. 2.15). For example, if the model is made by a data acquisition and a controller, a possible partitioned model puts these components in two difference referenced models at the model root-level. Each sample time in a referenced model or a system block is a partition. Then using the *Model Configurator*, in the Concurrent Execution dialog box, each task can be mapped to one processing node (by setting its affinity mask).

This tool is not yet mature, and complex to customize, for this reason, it does not suit well to mixed-critical applications. The aim of both implicit and explicit partitioning is the extraction of tasks by grouping blocks instead of finding a way to isolate functionality (robust partitioning). The automatic partitioning and mapping does not address the problem of safety and secure code execution. Instead, it focuses only on performance without any optimization. Even with explicit partitioning and mapping, the task priority assignment is rate-monotonic (tasks with shorter rate results in a lower priority) and does not take into account the Worst-Case Execution Time of tasks and the Resource usage. This information is critical for scheduling tasks while minimizing interferences and exploiting at best the task parallelism. Moreover, the whole process is strictly tied with the Simulink environment making difficult to integrate pieces of software developed by hand or automatically generated by other software.

To overcome these limitations, one possible approach is to customize the code generation process. Simulink provides a configuration tool for this: the *Target Language Compiler*. The following section briefly introduces it.

2.2.4 Target Language Compiler

The Target Language Compiler tool is part of the Real-Time Workshop. It enables the customization of the C/C++ code generated from any Simulink model or custom defined blocks. It has been designed for the purpose of converting the model description file, `model.rtw`, into target-specific code.

After reading the rtw file, the Target Language Compiler generates code based on *target files*, which specify the source code implementation of each block and the overall code framework for the entire model. TLC works like a text processor, it has a mark-up syntax similar to HTML, along with the power of scripting languages, plus the data handling power of MATLAB (TLC can invoke MATLAB functions). All Simulink blocks are automatically converted to code (except MATLAB function blocks and S-function blocks that invoke M-files).

In order to create a target-specific application, Real-Time Workshop also requires a template makefile that specifies the appropriate C compiler and its options for the build process. The template makefile is then transformed into a makefile (`model.mk`) by performing *token expansion* specific to a given model.

Any rtw file contain a series of (usually hierarchically nested) *records* in the form:

```
recordName {itemName itemValue}
```

Item names are alphabetic. Item values can be strings or numeric values (including scalars, vectors, and matrices). Curly braces set off the contents of each record, which may contain one or more items, delimited by space, tab, and/or return characters.

In an rtw file automatically generated from a Simulink model, the top-level (first) record's name is `CompiledModel`. Each block is represented by a subrecord within it, identified by the block name. The TLC, however, can parse any well-formed record file. For example, we can access:

```
%% In tlc file you can:
/* Print a comment in the output file */
%assign genDate = CompiledModel.GeneratedOn
The Model Name is: <%CompiledModel.Name>
Generated on: %<genDate>
```

The first line is a comment; all text on a line following the characters %% is treated as a comment (ignored, not interpreted or output). The text on the second line is printed in the file as it is.

The third line executes a TLC directive (because it starts with %), the directive `assign` creates a variable named `genDate` and assigns a string value to it. The `%assign` directive creates new and modifies existing variables. Its general syntax is:

```
%assign [::]variable = expression
```

The optional double colon prefix specifies that the variable being assigned to is a global variable. In its absence, a local variable in the current scope is created or modified.

The last two lines evaluate (expand) records. More precisely, the first line evaluates the field called `Name` in the scope of `CompiledModel`. The syntax `%<expr>` causes the expression `expr` (which can be a record, a variable, or a function) to be evaluated. This operation is sometimes referred to as an *eval*.

The previous statements do not print anything, because all outputs were directed to a null device (sometimes called the "bit bucket"). There is always one active output file, even if it is null. To specify, open, and close target files for code generation the following functions are available:

```
%openfile streamid ["filename"] [, mode]
%closefile streamid
%% Output operations ...
%selectfile streamid
```

The `%openfile` directive creates a file/buffer (in "w" mode), or opens an existing one (in "a" or "r" mode). The equal character is required for the specification of the output file name. Any number of streams can be open for writing, but only one can be active at one time. The directive can be used to switch between open files; they do not need to be closed until output operations to them actually end. When activated, `STDOUT` directs output to the MATLAB command window.

The `%with` and `%endwith` directive eases the burden of correctly coding TLC scripts and to clarify their context or scope. The syntax for their use is

```
%with RecordName
%% TLC stamements...
%endwith
```

they can also be nested; for example it can be very useful while using `foreach` loops to write

```
%with CompiledModel
%with DataTypes
%foreach dt=NumDataTypes
Data Type Index = %<dt>
Data Type Name: %<DataType[dt].SLName>
%endforeach
%endwith
%endwith
```

2.2.4.1 Counting Blocks and Subsystems

As an example, the following TLC code counts the subsystems and blocks in the model.

```
%% File: CountSubSysAndBlocks.tlc
%% Count blocks and subsystems of a model.rtw file
%%
%% NOTE: Change "<matlabroot>/" below to the
%% MATLAB main directory on your system
%addincludepath "<matlabroot>//rtw//c//tlc//lib" %%Enables TLC to find
    included files
%assign Accelerator = 0 %%Needed to avoid error in utillib
%include "utillib.tlc" %%Inserts one file in another, as in C (not used
    here)
%selectfile STDOUT
*** SYSTEMS AND BLOCKS IN RECORDFILE
%assign nbls = 0
%with CompiledModel
%foreach sysIdx = NumSystems
%with System[sysIdx]
%assign nbls = nbls + NumBlocks
*** %<NumBlocks> blocks in system %<sysIdx + 1>
%endwith
%endforeach
*** recordfile contains %<nbls> blocks in %<NumSystems> systems
%endwith
*** END LISTING
%% end CountSubSysAndBlocks.tlc
```

the script can be executed by writing in the command window

```
tlc -r model.rtw CountSubSysAndBlocks.tlc
```

obtaining, for example

```
*** SYSTEMS AND BLOCKS IN RECORDFILE
*** 8 blocks in system 1
*** 10 blocks in system 2
*** 4 blocks in system 3
*** 7 blocks in system 4
*** 17 blocks in system 5
*** recordfile contains 46 blocks in 5 systems
*** END LISTING
```

2.3 Developed Framework

The aim of the thesis is to develop a framework to run different applications with different criticality levels in multi-core platforms ensuring adequate safety requirements. While the development of a robust optimal partitioning and scheduling algorithm is still an open problem, we focused on developing a code generation framework to implement, on PikeOS, a suitable scheme.

For this reason, we first developed a complete but relatively simple partitioning and a scheduling algorithm that extracts information from a Simulink model, lets the designer perform some configuration options and then provides an optimization engine for the allocation of the threads. The framework enables a full process flow for the automatic code generation of a mixed critical application. Figure 2.16 shows the simplified design flow.

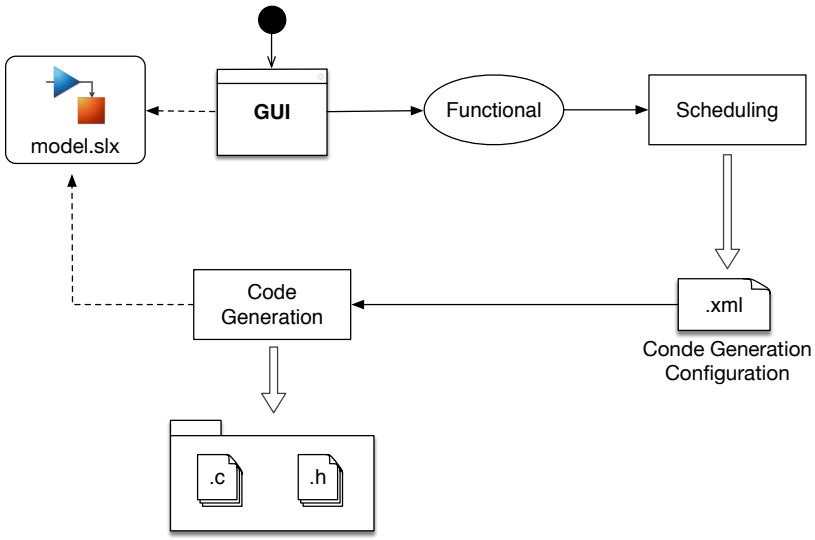


Fig. 2.16 Developed Framework overview

Once the models are developed and the simulation results are satisfactory, the system designer groups blocks in subsystems, which are considered as the unit of execution, such that there are no spare blocks among subsystems. Then a Matlab script uses the Simulink modeling API (programming interface) to parse the model structure and create a functional representation of the model to be utilized in the scheduling process (eventually exported as an XML). Finally, after the completion of the scheduling, the code generation process can start.

2.3.1 Model Functional Representation

A current challenge in the development of parallel applications is the achievement of a good scalability with the number of threads and processors. Often the scalability is heavily reduced by the precedence order of execution among threads (usually due to data dependencies). A possible approach to model the problem is through a *Directed Acyclic Graph* (DAG). The DAG is a functional representation of the model, it consists of vertices (threads) and edges (communications/precedences) among nodes, with each edge directed from one vertex to another. The fact that it is an acyclic graph ensures that there is no *algebraic loop* in the model. An algebraic loop occurs when a signal loop with only direct feedthrough blocks within it exist.

The designer can also add additional information to each node/thread such as

- Worst Case Execution Time (WCET) after performing some timing analysis on the subsystems[2].
- Criticality Level (A to E).
- Resource Usage (a piece of resource like Shared Memory, UART, Ethernet, etc. can be defined and assigned to one or more threads).

This information is used by the engine to properly schedule threads on cores.

The functional model is created by a Matlab script that uses the Simulink modeling API (programming interface) to parse the model structure and create the DAG. It is based on the work of Matteo Morelli [Mor15]. The Simulink model must comply with the restriction that:

1. The model consists of a collection of subsystems, with no spare blocks among them.
2. There are no continuous time blocks.
3. There are no multi-rate subsystems.

[2] the subsystem will become a task, so this is the execution time related to a task.

These restrictions are a result of the fact that a C implementation should be generated from the model.

2.3.2 Partitioning and Scheduling Optimization

The DAG represents a *precedence graph* among threads, so it imposes a partial order of execution. To automatically generate an implementation (code generation) we must compute a total order among threads. This gap is an opportunity to improve safety and predictability (determinism) through partitioning and scheduling.

In order to perform a real-time scheduling for the asks on multiprocessor platforms there exist two basic approaches: the *partitioned approach*, in which each task is statically assigned to a single processor and migration is not allowed, and the *global approach*, in which tasks can freely migrate and execute on any processor. Works from Gracioli[GFF13] and Melani[Mel] shows a comparison of global scheduling and partitioned scheduling. Even though the global scheduling has several advantages, this thesis is focused on partitioned scheduling because the global scheduling would lead to preemptions and migrations, which produce more overheads and less determinism. In particular the latter might cause certification issues and this thesis wants reduce this possibility.

In this thesis we propose a general framework for a partitioned scheduling that can be mapped on PikeOS execution entities. For this purpose we identified several phases:

1. Mapping execution unit to tasks.
2. Task partitioning.
3. Task scheduling.
4. Priority assignment.
5. Partition scheduling.

The scheduling process is implemented in Matlab as shown in figure 2.17.

2.3.3 Code Generation

An optimal solution for the generation of code for mixed-criticality systems is not available from the literature. Since robust partitioning is the common way to address mixed-criticality, the code generation framework should take as input the description of such partitioned system. Recalling the fact that the code needs to be mapped into PikeOS execution entities, it is

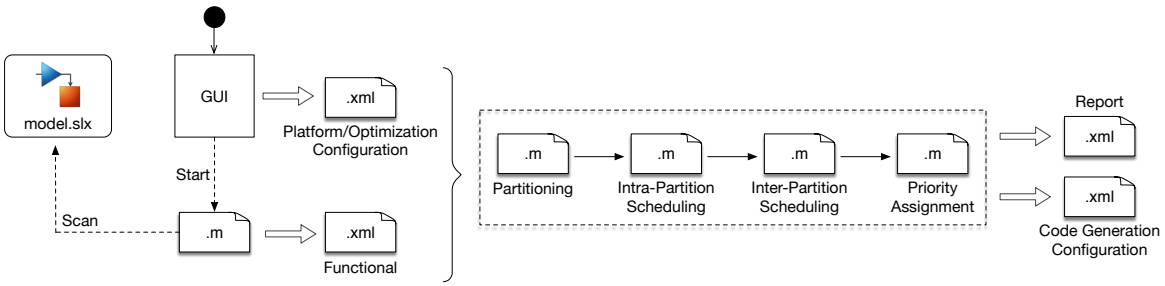


Fig. 2.17 Scheduling algorithm process

assumed that (with respect to PikeOS terminology) each Resource Partition is made by one Process with its threads and assigned to only one Time Partition. Obtaining the software architecture depicted in figure 2.18. The reason why this architecture is proposed is that

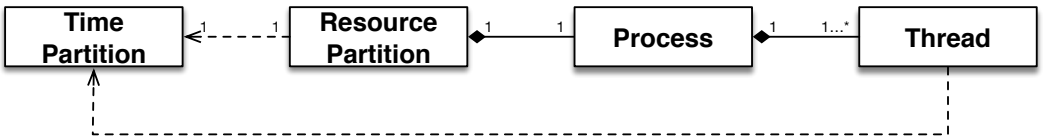


Fig. 2.18 Software Architecture in the Developed Framework

a different one would introduce further synchronization issues and less determinism. For example splitting a process in two separate processes, both assigned to the same resource partition would increase the possible interleaving among threads (basically all the possible states) and more synchronization calls for the communication. Moreover, since two processes (even if they are in the same Resource Partition) do not share the same virtual address space, more channels or additional shared memory regions must be defined. The same issues are caused if more than one Resource Partition is assigned to the same Time Partition.

The underlying idea is to create a heterogeneous code generator partially independent from the platform. Since in this work the model representation is the Simulink Model, the code generation process is divided in two steps:

1. Subsystems adaptation and generation: during this step Subsystem communications are eventually mapped to the right primitives (as defined in the code generation configuration file) and each Subsystem is generated with the Real Time Workshop/Embedded Coder as independent code.

2. System target code generation: here all the subsystems are glued together as threads, processes and partitions with target specific code.

It can be noticed that the first step is OS agnostic if the communication primitives are an abstraction of the Operating System ones. The resulting code generation process is depicted in figure 2.19.

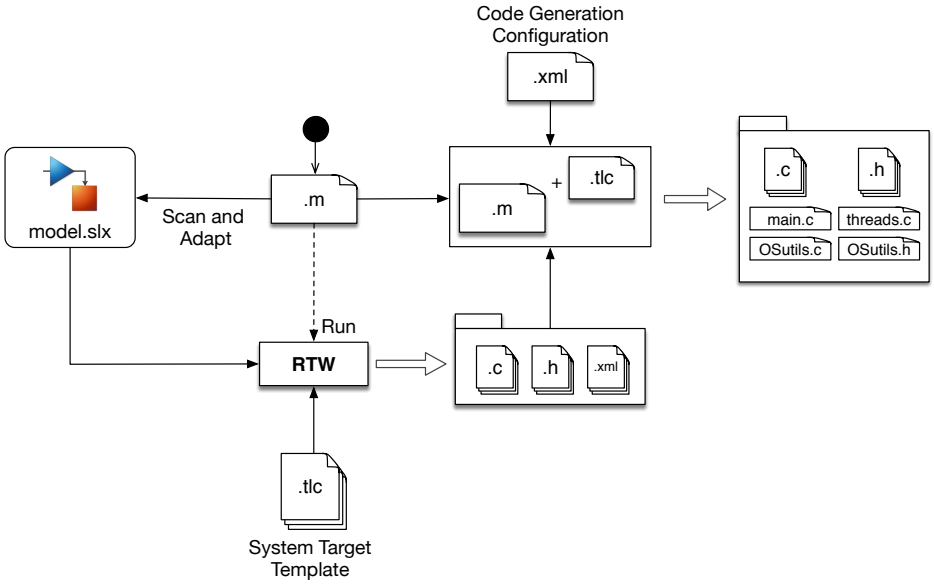


Fig. 2.19 Code generation process

Chapter 3

Scheduling

In this chapter a scheduling framework for mixed-critical applications with precedence constraint and communication costs is presented.

3.1 Introduction

It has been shown [Bar09] that the mixed-criticality schedulability problem (preemptive or non-preemptive) is strongly NP-hard even with only two criticality levels (High and Low). Nevertheless, different approaches had been proposed in the literature. The first research was presented in 2009 by Anderson et al.[ABB09] and extended in 2010 [MEA$^+$10]. The mechanism they presented is based on assigning to high-critical tasks higher Worst Case Execution Time and different scheduling policies (e.g. level A tasks are statistic assigned and cyclic released, level B with a Partitioned-EDF scheduler, level C and D with G-EDF). Kritikakou et al.[KRF$^+$14] provided a new scheduling for tasks distinguished in only two levels: HI-criticality and LO-criticality. Same as proposed by S.Baruah et al.[BBD$^+$12] in which they describe EDF-VD (for Earliest Deadline First with Virtual Deadlines) for mixed-criticality tasks (see [ZGDY14] for detailed analysis).

To properly treat the problem an formal abstraction of the real-time scheduling with mixed-critical tasks with precedence and periodicity is presented.

3.2 Problem Formulation

As stated in the previous chapter, tasks (or threads) to be scheduled are represented as an acyclic directed graph. Every node in the graph represents one task and the edges between

nodes, the communications. The node cost is the time required for the task to complete (WCET) and the edge cost is the communication cost.

More formally, the tasks are defined by $G = (\Gamma, E, C, T, K)$ where $\tau_i \in \Gamma$ represents a task and Γ the task-set. The set $E = \{e_{ij} : \forall \tau_i \rightarrow \tau_j\}$ represent the precedence constraints between τ_i and τ_j (meaning that τ_i must be completed before τ_j) with its communication cost expressed in time. The Worst-Case Execution Times are expressed in $C = \{c_i : \forall \tau_i \in \Gamma\}$. The periods (or rate) are $T = \{T_i : \forall \tau_i \in \Gamma\}$ and it is assumed that every T_i is a integer multiple of some base-period β_T. The criticality levels are $K = \{\chi : \forall \tau_i \in \Gamma\}$. Moreover, each task has its priority ρ_i which is assigned by the scheduling algorithm and a set of accessed resources $\mathbb{R}$ manually assigned by the system designer to each task.

The Direct Acyclic Graph made by all the partitions (which are the group of tasks) is denoted by $\mathbb{P} = (\Pi, H, L, R)$ and called *P-DAG*, where $\pi_i \in \Pi$ is a partition. The inter-partition communications are represented in H, $\lambda_i \in L$ and $\delta_i \in R$ are respectively the duration and the periodicity of the partition π_i. So we can define a map $\Psi : \Gamma \rightarrow \mathbb{P}$ as the partitioning algorithm. The subgraph of G made by all the tasks assigned to a given partition is called T-DAG.

The behavioral parameters for the task τ_i that the scheduling process must define are: the starting time s_i (*when* the task should execute), and the core μ_i on which it will execute (*where* should execute), also called *affinity mask*.

3.3 Assumptions

It is assumed that the COTS board is a connected network of processors with identical communication links (the Unified Memory Model shown in figure 1.2) and relatively small number of processors. This simplifies the mathematical formulation of the optimization problem and limits its computational complexity.

It is also assumed that the partitioning addresses the security and safety requirement. This mechanism relies on the Hypervisor, which is a trusted code (certified by the authorities) and it is the only one executing in the highest privileged mode. It ensures time and spatial isolation among partitions so the partitioning algorithm should map each task to one partition such that a fault in one partition does not affect another partition while considering the criticality as a decision variable. Moreover, interferences and inter-partition communications should be minimized.

Once partitions are determined, they need to be scheduled. The problem can be split in two parts: *intra-partition* scheduling and *inter-partition* scheduling. In the following sections is presented a detailed description of each phase.

3.4 Partitioning

Determine a way to measure safety is complex, hence, derive an optimization problem is not easy. In order to simplify the intra-partition scheduling and enforce determinism it is important that all the tasks inside a partition have the same period (or eventually integer multiple of the partition rate). To understand the rationale, assume that a task τ_i assigned to TP_1 needs to be activated at time $t_1 < t_{L_1}$ and $t_2 > t_{L_2}$ as shown in figure 3.1

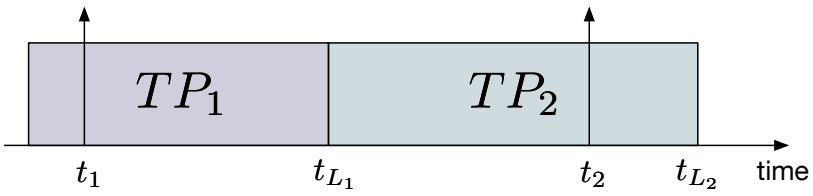


Fig. 3.1 Non rate-base partitioning

To allow this behavior, two approaches are possible:

1. Introduce preemption among time partition, losing determinism and the control of safety states.

2. Push the execution of τ_i o the new activation of TP_1. This approach leads to a lower level of determinism because τ_i now should interrupt any task assigned to TP_2 that is executing at time t_2. Moreover, the worst-case execution time of TP_2 will considerably change between different execution, leading to an over-estimating of it.

The solution adopted in this work does not pretend to be a solution for the partitioning problem. Instead, it would be an example. The algorithm simply groups all the tasks with the same rates in partitions and then split them according to some criticality threshold, creating smaller sub-partitions. A more complex partitioning algorithm is under development.

As stated before, the result of the partitioning is another DAG, called P-DAG, each of them with its T-DAG. Now the inter-partition and intra-partition scheduling can be introduced.

3.5 Tasks Allocation and Scheduling

3.5.1 Intra-Partition

In order to schedule partitions the execution time of the partition itself must be estimated. This optimization schedules a T-DAG on all the available $|M|$ processors. The solution space of this problem is spawned by all possible processor assignments combined with all possible task orderings that satisfy the partial order expressed by the T-DAG. The tasks are to be assigned in such a way as to minimize the total computation time required to execute that partition. This is also referred as reducing the makespan. The optimization problem that solves this problem presented below is based on the one proposed by S.Venugopalan and O.Sinnen [VS12].

For each task $\tau_i \in \Gamma$, let s_i the starting time, μ_i the core on which it will be executed and γ_i the cost of all outgoing communications. Let W the makespan and $|M|$ the number of available cores. Moreover, let $\delta^-(i)$ the set of tasks that need to be completed before task τ_i. Some tasks cannot execute in parallel with another due the shared resource they are going to use, $\mathscr{I}$ is matrix that represent *parallel incompatibilities*, the component $\mathscr{I}_{ij}$ is equal to one if τ_i and τ_j cannot execute in parallel, formally:

$$\mathscr{I}_{i,j} = \begin{cases} 1 & \text{if } \tau_i \text{ and } \tau_j \text{ share at least one resource} \\ 0 & \text{otherwise} \end{cases}$$

Let the variable x_{ik} is one if task τ_i is assigned to the processor μ_k, zero otherwise. In order to control the scheduling behavior, define the following set of binary variables:

$$\forall \tau_i, \tau_j \in \Gamma \quad \sigma_{ij} = \begin{cases} 1 & s_i + c_i \leq s_j \\ 0 & \text{otherwise} \end{cases}$$

$$\forall \tau_i, \tau_j \in \Gamma \quad \varepsilon_{ij} = \begin{cases} 1 & \mu_i < \mu_j \\ 0 & \text{otherwise} \end{cases}$$

The resulting MILP problem is

$$\min \quad W \tag{3.1}$$
$$\forall \tau_i \in \Gamma \quad s_i + c_i \leq W \tag{3.2}$$
$$\forall \tau_i \neq \tau_j \in \Gamma \quad s_j - s_i - c_i - \gamma_i - (\sigma_{ij} - 1)W_{\max} \geq 0 \tag{3.3}$$
$$\forall \tau_i \neq \tau_j \in \Gamma \quad \mu_j - \mu_i - 1 - (\varepsilon_{ij} - 1)M \geq 0 \tag{3.4}$$
$$\forall \tau_i \neq \tau_j \in \Gamma \quad \sigma_{ij} + \sigma_{ji} + \varepsilon_{ij} + \varepsilon_{ji} \geq 1 \tag{3.5}$$
$$\forall (i,j) : \mathscr{I}_{ij} = 1 \quad s_i + c_i + \gamma_i - s_j \leq W_{\max}(1 - \sigma_{ij}) \tag{3.6}$$
$$\forall (i,j) : \mathscr{I}_{ij} = 1 \quad s_j + c_j + \gamma_j - s_i \leq W_{\max}\sigma_{ij} \tag{3.7}$$
$$\forall \tau_i \neq \tau_j \in \Gamma \quad \sigma_{ij} + \sigma_{ji} \leq 1 \tag{3.8}$$
$$\forall \tau_i \neq \tau_j \in \Gamma \quad \varepsilon_{ij} + \varepsilon_{ji} \leq 1 \tag{3.9}$$
$$\forall \tau_j \in \Gamma : \tau_i \in \delta^-(j) \quad \sigma_{ij} = 1 \tag{3.10}$$
$$\forall \tau_i \in \Gamma \quad \sum_{k \in |M|} k x_{ik} = \mu_i \tag{3.11}$$
$$\forall \tau_i \in \Gamma \quad \sum_{k \in |M|} x_{ik} = 1 \tag{3.12}$$
$$0 \leq W \leq W_{\max} \tag{3.13}$$
$$\forall \tau_i \in \Gamma \quad s_i \geq 0 \tag{3.14}$$
$$\forall \tau_i \in \Gamma \quad \mu_i \in \{1, ..., |M|\} \tag{3.15}$$
$$\forall \tau_i \in \Gamma, k \in |M| \quad x_{ij} \in \{0,1\} \tag{3.16}$$
$$\forall \tau_i, \tau_j \in \Gamma \quad \sigma_{ij}, \varepsilon_{ij} \in \{0,1\} \tag{3.17}$$

Where $W_{\max}$ is an upper bound for the makespan W. It can be computed as all the tasks were executed on a single core (so it is the sum of computational cost and communication costs) or with some heuristics.

The formulation is a min-max problem: this is achieved by minimizing the makespan W while introducing the constraint (3.2). Constraint (3.3) impose the partial order on the tasks in terms of the σ variables. Constraint (3.4) impose the multi-core usage. Constraint (3.5) impose that at least one of the following is true: τ_i must finish before τ_j starts and/or $\mu_i < \mu_j$. Constraints (3.6) and (3.7) avoid that two tasks that share a common resource execute in parallel. By (3.8) and (3.9) a task cannot be before and after another task in both time and cores. Constraint (3.10) enforces the task precedences defined by the T-DAG. Constraints (3.11) link the assignment variable x with the core variables μ and finally (3.12) ensures that any given task runs only on one core.

The complexity in terms of constraint and variables, depends on $|G|$, $|E|$, $|M|$ and $|\mathscr{I}|$. Assuming that the number of processor $|M|$ and the number of shared resources $|\mathscr{I}|$ are small, then the MILP complexity is dominated by (3.10) which generates $O(|G||E|)$ constraints. In the worst case scenario $|E| = |G|(|G|-1)/2$, however, for task-sets representing real applications, we usually have $O(|E|) = O(|G|)$, hence the overall complexity is $O(|G|^2)$.

Once a T-DAG related to a partition is scheduled the makespan of the schedule is the Worst-Case Execution Time of the partition itself. Moreover, the variables s_i and μ_i for each task $\tau_i \in \Gamma$ are known so the priorities can be computed.

3.5.2 Inter-Partition

The inter-partition schedule is analogue to the problem of schedule the P-DAG on a single core. Indeed, each Resource Partition is assigned to a Time Partition that is as big the total Worst-Case-Execution-Time of the Resource Partition contained (this amount of time can be estimated after the intra-partition schedule). In PikeOS, Time partitions are scheduled according to a statically-assigned schedule scheme like they were on a single core.

The problem of scheduling tasks on a single core have received substantial attention and many algorithms are available in the literature, for a complete review see [But11] and [BEP+13].

When scheduling the P-DAG, the partial order expressed by it must be satisfied. Let introduce some concept (as in [BEP+13]). For seek of notation simplicity, let consider a partition like a task, so that the same notation as before can be used. In addition to the previous notation, let introduce the *arrival time* of a partition π_i as r_i which represent the moment in time in which a partition can start its execution, and the *due date* $\widetilde{d_i}$ as the moment in time in which the task must be completed. These parameters, together with the periodicity, represent the real-time requirement for a given partition.

3.5.2.1 Factorization

Considering all nodes in the P-DAG, it is common to find different periodicity. In the general problem formulation is assumed that the periodicity of each task is an integer multiple of a base-period β_T, so when they are grouped into a partition, the partition itself inherit the rate of the tasks it contains and the so does the property. If $T_i = k_i\beta_T$, the *Hyper-Period* or *Major*

Time Frame can be defined as

$$\Delta = \gcd(k_1, k_2, ...)\beta_T \quad i = \{1, ..., |\Pi|\} \tag{3.18}$$

Inside the hyper-period some partitions π_i should execute more than once, in general exactly k_i times. In order to generalize this behavior, a *factorized P-DAG* can be defined. Let denote it as $\widetilde{\mathbb{P}} = (\widetilde{\Pi}, \widetilde{H}, \widetilde{L}, \widetilde{R})$, it is a *finite repetitive precedence* of partition π_i by exactly k_i times, in a direct precedence relation. The factorization process is depicted in figure 3.2.

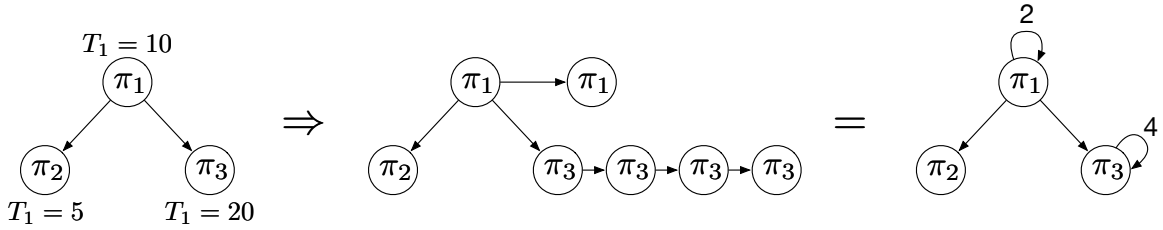


Fig. 3.2 Factorization example

3.5.2.2 Partitions precedence constraints

Given a schedule, it is called *normal* if, for any two partition $\pi_i, \pi_j \in \Gamma$, $s_i < s_j$ implies that $\widetilde{d_i} \leq \widetilde{d_j}$ or $r_j > s_i$. Release times and deadlines are called *consistent with the precedence relation* if $\pi_i \rightarrow \pi_j$ implies that $r_i + \delta t \leq r_j$ and $\widetilde{d_i} \leq \widetilde{d_j} - \delta t$, where δt represent a small amount of time (basically the scheduling decision tick-time). The following lemma proves that the precedence constraints are not of essential relevance if there is only one processor.

Lemma 3.5.1. *If the release times and deadlines are consistent with the precedence relation, then any normal one-processor schedule that satisfies the release times and deadlines must also obey the precedence relation.*

Proof. Consider a normal schedule, and suppose that $\pi_i \rightarrow \pi_j$ but $s_i > s_j$. By the consistency assumption we have $r_i < r_j$ and $\widetilde{d_i} < \widetilde{d_j}$. However, these, together with $r_j \leq s_j$, cause a violation of the assumption that the schedule is normal, a contradiction from which the result follows. □

This lemma ensures that release times and deadlines can be made consistent with the precedence relation if they are redefined by:

$$r_j^{'} = \max\left(\{r_j\} \cup \{r_i^{'} + \delta t : \pi_i \rightarrow \pi_j\}\right) \tag{3.19}$$

$$\widetilde{d}_j^{'} = \min\left(\{\widetilde{d_j}\} \cup \{\widetilde{d}_i^{'} - \delta t : \pi_j \rightarrow \pi_i\}\right) \tag{3.20}$$

These changes do not alter the feasibility of any schedule. Furthermore, from lemma 5.3.4 follows that a precedence relation is essentially irrelevant when scheduling on one processor.

3.5.2.3 Bratley algorithm

Scheduling partitions with precedence constraint (or adapted arrival times and due dates) is NP-hard in the strong sense, even for integer release times and deadlines [LKB77]. Only if all tasks have unit processing times, an optimization algorithm of polynomial time complexity is available. However, Bratley et al. [BFR71] proposed a branch-and-bound algorithm which solves this class of problems. Their algorithm is shortly described below.

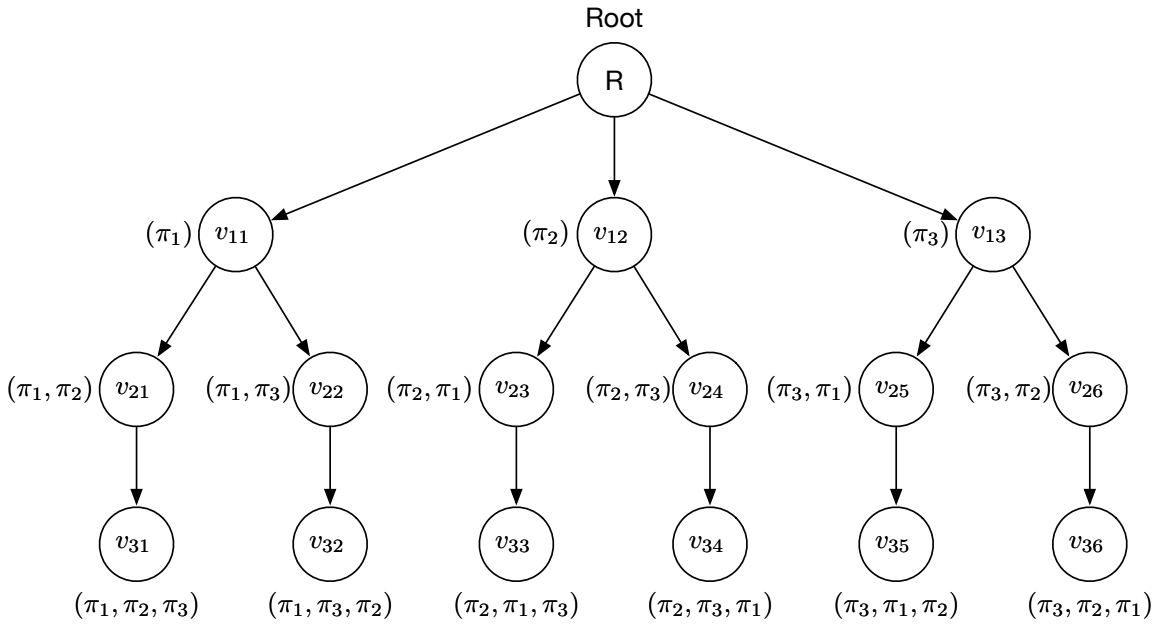


Fig. 3.3 Search tree example for Bratley et al. algorithm

All possible partition schedules are implicitly enumerated by a search tree as shown in Figure 3.3 (for three partitions). Each node v_{ij} of the tree represents the assignment of $i-1$ partitions and a new unscheduled one to be in the i-th position of the schedule scheme, with $i = \{1,...,N\}$, $N = |\widetilde{\mathbb{P}}|$. On each level i, there are $N-i+1$ new nodes generated from each node of the preceding level. Hence, all the $N!$ possible schedule will be enumerated. To each node is associated the completion time of the corresponding partial schedule.

The order in which the nodes of the tree are examined is based on a *backtracking search strategy*. Moreover, the algorithm uses two criteria to bound the solution space.

1. Exceeding deadlines. Consider a node v_{ij} of the tree where one of the i partitions exceed its due date, it will certainly exceed its deadline if other partitions are scheduled after it. Therefore, the node with all the sub-tree may be excluded from further consideration.

2. Problem decomposition. Consider a node v_{ij} where an unscheduled partition is assigned to i-th position of the schedule scheme. If the completion time of this partial schedule is less than or equal to the smallest release time among the yet unscheduled partitions, then the problem decomposes at level i, and there is need to backtrack beyond level i. This follows from the fact that the best schedule for the remaining $N-i$ partitions may not be started before the smallest of their release times.

After enumerating all the possible $N!$ the best schedule according to an objective function can be selected. A common objective function is the makespan minimization.

3.6 Priority assignment

Priority assignment is required to allow the operating system scheduler to execute tasks according to the optimal schedule.

Let assume that each thread has its affinity mask, meaning that it can execute only on the core specified by it and that the scheduler is priority-based FIFO queue. To enforce the non-preemptive behavior for the tasks inside a partition, threads on the same core must have *strictly monotonically decreasing* priorities. Here to derive a correct assignment algorithm, an assumption on the implementation is required. Priorities alone cannot ensure mutual exclusion on communications memory locations. These shared memory regions are accessed only by communicating thread and them can be placed:

- On the same core: so priorities can ensure that the inputs are fulfilled, indeed the task with lower criticality will not execute before higher priority task.

- On different core: so spinlocks can be used.

The use of spinlocks for inter-core synchronization is suggested because they avoid overhead from operating system process rescheduling or context switching. Moreover, spinlocks are efficient if tasks are likely to be blocked for only short periods, which is true to a certain degree that depends on the worst-case timing analysis.

A simple yet effective way to achieve this result is through a Linear Programming optimization problem:

$$\begin{cases} \min \sum \rho_i & \\ \rho_i - \rho_j \leq -1 & \begin{array}{l} \text{for each consecutive task } \tau_i, \tau_j \text{ on the same core} \\ \text{for each communication edge } e_{ij} \text{ between cores} \end{array} \\ \rho_{\min} \leq \rho_i \leq \rho_{\max} & \end{cases} \tag{3.21}$$

where ρ_i is the priority assigned to task $\tau_i \in \Gamma$. This class of problems can be solved in polynomial time [Kar84].

Usually an operating system can only handle a finite set of priority values, for this reason the variable ρ is bounded. However, if the schedule priority assignment does not use all the possible priority values, it is possible to create a gap below and above the partition to allow the execution of sporadic tasks. For example, this behavior can be easily implemented utilizing the background PikeOS partition TP_0. The result is depicted in figure 3.4.

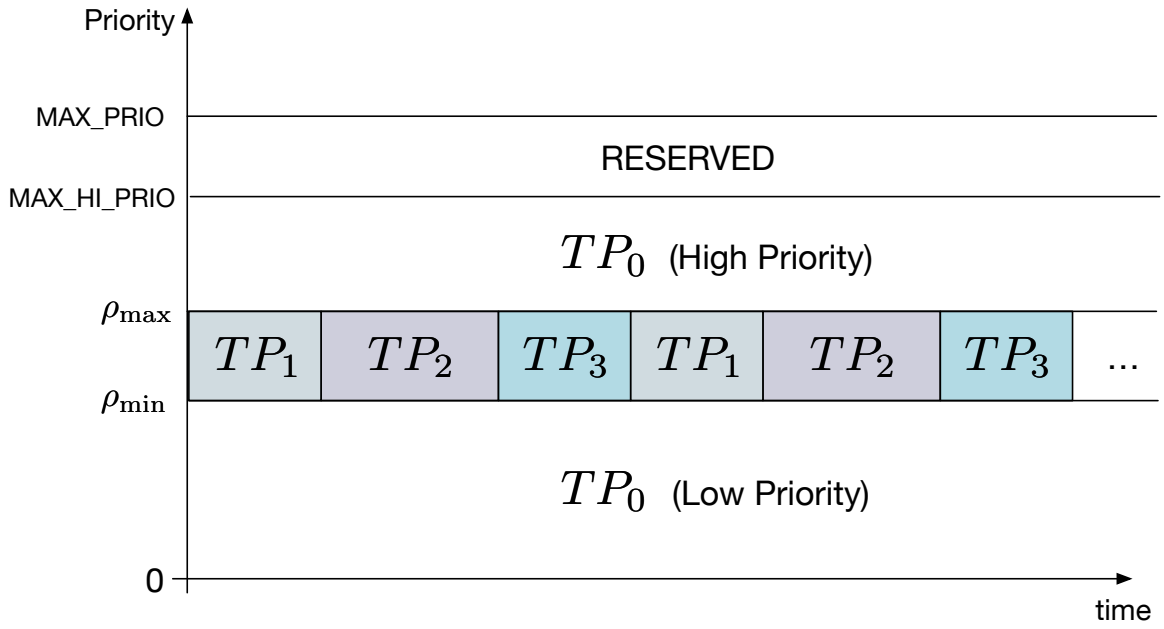


Fig. 3.4 Priority Assignment

Chapter 4

Code Generation

Software development for embedded systems is increasingly being done with the help of block diagram specifications, simulation tools, and automatic code generation. With the growing importance of multi-core processors in safety-critical systems, there is also an increasing need for certification-compliant and time-predictable implementation via code-generation. In this chapter the code generation framework for Simulink is presented.

4.1 Code-Generation Configuration

Before starting the code generation process, some configuration is required. In particular, the code generator needs to know about the software architecture, i.e.:

- *Partitions*. The number of Partitions and their properties such as the duration and the number of activations.
- *Threads*. The DAG enriched with the information coming from the scheduling (affinity mask, priority and partition).

The current implementation expects an XML file (which is automatically generated by the developed scheduling framework) containing the information as shown in the following example of the configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<Example filepath="C:\MATLAB\Models\Example.slx">
   <partitions number="2">
      <partition id="1" period="Inf" wcet="0.5">
         <activations/>
      </partition>
      <partition id="2" period="0.01" wcet="0.006">
```

```
8         <activations>
9            <activation>0</activation>
10        </activations>
11     </partition>
12  </partitions>
13  <nodes>
14     <node core="1" criticality="4" name="T1" partitionID="2" period
    ="0.01" priority="7" start="0.003" wcet="0.001">
15        <resources>
16           <resource id="3">AXI</resource>
17        </resources>
18        <feedthrough>
19           <relationship dst="1" src="1"/>
20        </feedthrough>
21     </node>
22     <node core="1" criticality="4" name="T2" partitionID="2" period
    ="0.01" priority="6" start="0.004" wcet="0.001">
23        <resources/>
24        <feedthrough>
25           <relationship dst="1" src="2"/>
26           <relationship dst="1" src="3"/>
27        </feedthrough>
28     </node>
29     <node core="1" criticality="4" name="T3" partitionID="2" period
    ="0.01" priority="5" start="0.005" wcet="0.001">
30        <resources/>
31        <feedthrough>
32           <relationship dst="1" src="1"/>
33           <relationship dst="1" src="2"/>
34        </feedthrough>
35     </node>
36     <node core="1" criticality="4" name="T4" partitionID="1" period="
    Inf" priority="5" start="0" wcet="0.5">
37        <resources/>
38        <feedthrough/>
39     </node>
40     <node core="1" criticality="4" name="T5" partitionID="2" period
    ="0.01" priority="8" start="0" wcet="0.003">
41        <resources>
42           <resource id="3">AXI</resource>
43        </resources>
44        <feedthrough/>
45     </node>
46  </nodes>
```

```
47    <edges>
48       <edge destination="T2" dstPort="3" size="192" source="T1" srcPort
      ="1"/>
49       <edge destination="T3" dstPort="2" size="192" source="T2" srcPort
      ="1"/>
50       <edge destination="T2" dstPort="2" size="192" source="T4" srcPort
      ="2"/>
51       <edge destination="T1" dstPort="1" size="192" source="T5" srcPort
      ="1"/>
52       <edge destination="T3" dstPort="1" size="192" source="T5" srcPort
      ="1"/>
53    </edges>
54    <platform>
55       <cpu>2</cpu>
56    </platform>
57 </Example>
```

This file is parsed by a Matlab script and represented as a hierarchical class structure in the workspace. The resulting object is used by the next steps.

4.2 Model Configuration for Code Generation

Once the configuration is loaded, a Matlab script adapt the model to implement the software architecture described in the configuration file. In particular, it has to address the communication issue.

All the threads inside a process share the same virtual memory address space, so a global variable containing an output (or input) of a thread is accessible from every thread. When the threads are scheduled inside two different partitions, this variable is no longer accessible. For solving this problem, two approaches are possible:

1. *Communication Primitives*. Use message oriented communication such as Sampling or Queuing Ports.

2. *Shared Memory Regions*. Define shared memory regions between the two partitions.

While shared memory is suitable for a large amount of data, usually a single inter-partition communication channel, which is the implementation of a Simulink Line, transfers a limited amount of bytes. This lead to the use of communication primitives. As explained in section 2.1.9, Queuing ports implement buffered communication while Sampling ports always contain the freshest value and its validity flag with respect to the expected refresh rate. This last capability is very useful to understand if a fault occurred to one of the predecessors of

the executing thread, and eventually implement some handler for a not up-to-date value. For this reason, Sampling Ports are used as Inter-Partition Communications.

Therefore, every subsystem's Outports and Inports that are at the edge of two different partitions must be replaced by the relative operation on the Sampling port. For this reason, a *Custom Block* for each operation has been developed.

4.2.1 Custom Blockset for Sampling Ports

Any Inport and the Outports (which are blocks) that implement an inter-partition communication need to be substituted with other blocks that, respectively, implement a Read and Write on the Sampling Port. The mechanism that Simulink provide for extending its built-in modeling functionality is the definition of *Custom Blocks*. Each custom block is composed of two files: an S-Function that describe the behavior of the block during simulation and a TLC file specifying the code that is going to be generated out of it.

An S-function is a description of a Simulink block written in MATLAB, C, C++, or Fortran. C, C++, and Fortran S-functions are compiled as *MEX files*, which are dynamically linked subroutines that the MATLAB interpreter can load and execute. Either a Level-2 MATLAB S-function or a C S-function (that then require a TLC file) support code generation.

In this work, a Level-2 C S-Function has been chosen. It must implement all the functions showed in the left side of figure 2.13 together whit a `mdlRTW` function that writes all the block parameters in the .rtw file generated from the model. How the Simulink Engine calls these function may vary, focusing on code generation, the callback sequence is outlined in picture 4.1 (see [SFu] for further details). The function `mdlInitializeSizes` is important because it sets the sizes of the various signals and vectors and so can inherit the port configuration information such as the size and the data type. Then through the function `mdlRTW` these parameters are written into the .rtw file to be used by the TLC file.

Each block has a target file that determines what code should be generated for that block [Blo]. Within each block target file, *block functions* specify the code to be output during the start function, output function, update function, and so on. After the Code generation block initialization (i.e. `BlockInstanceSetup` and `BlockTypeSetup`) different TLC function generate executable code, let focus on `Start` and `Outputs` which are used for implementing the ports.

- *Start*. The code inside this function executes once and only once, therefore, it is used to instantiate a global variable for the port descriptor and to initialize it (i.e. open the port).

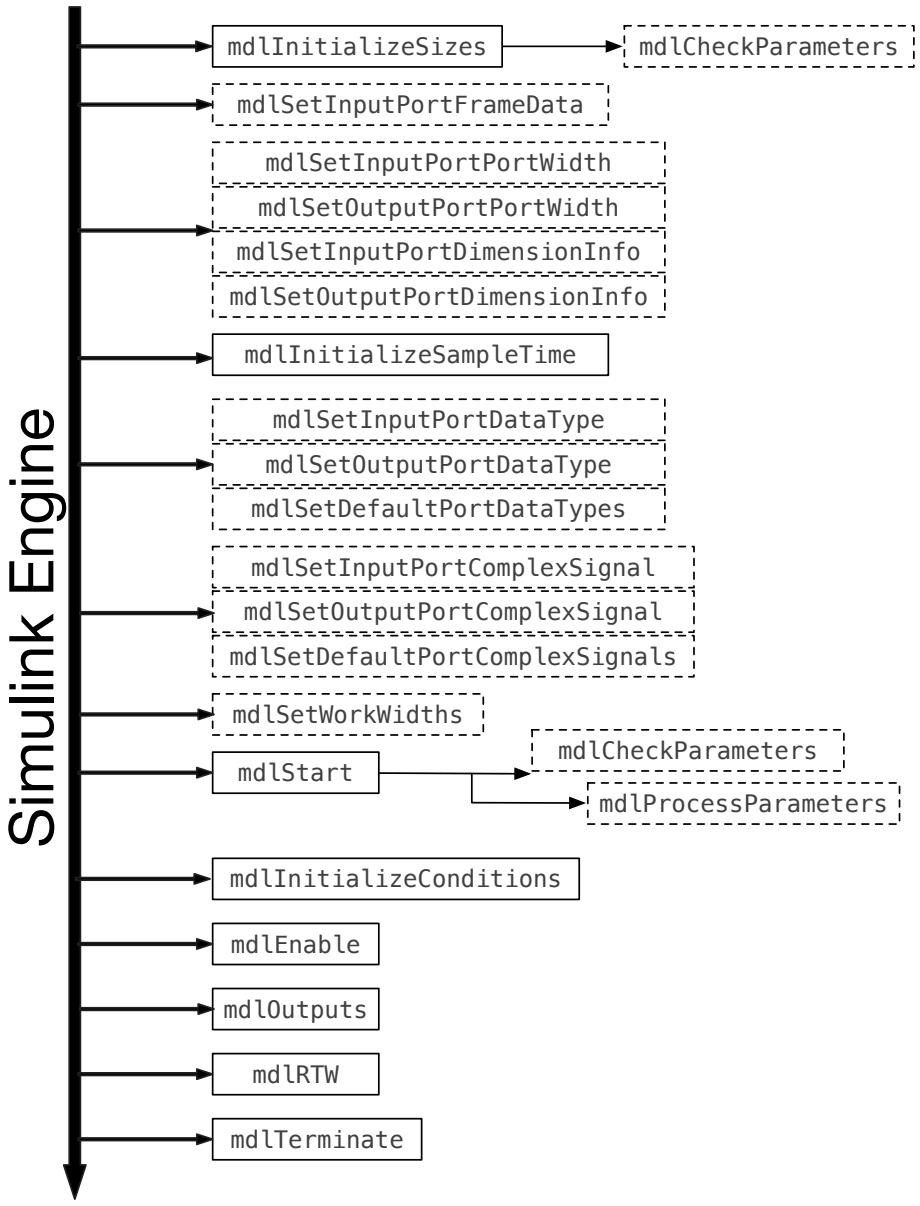


Fig. 4.1 Simulink Engine Callbacks Order in Case of Code Generation

- *Output*. The code inside this function is placed in model's Outputs function and generate the code required to access the port.

4.3 System Target File

Once the model has been adapted to the software architecture described in the configuration file, a Matlab script file scans the model start the code generation process. The script run the command `rtwbuild` on every subsystem, so RTW creates a folder in the current working directory and places all the generated files. In order to drive the code generation process a *custom System Target File* has been developed.

The custom System Target File generate C code as the default System Target File from Embedded Coder would do (see section 2.2.2.1). This is achieved by exploiting the built-in TLC script `codegenentry.tlc`. The generation process is incremental, so every time a new block is encountered, some code is added to the temporary data structure that later on will be combined to form the source files. At the end of the process, when all the information about the code is known, an additional XML file is generated. The file describes the interface of the generated code, including:

- *BuildDir*. The sub-folder of the current working directory where the source files are stored.
- *SrcFiles*. The list of all the source files.
- *Header and Body*. The header and the body of the subsystem, this are the main file implementing the subsystem code.
- *Inputs and Outputs*. The name of the structure containing the Inputs/Outputs for the subsystem code and every structure element (ports) with its name and dimension.
- *Functions*. The name of the entry point functions.

An example of a subsystem called `Ctrl1Err` is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<interface name="Ctrl1Err">
    <BuildDir>Ctrl1Err_p4_zynq</BuildDir>
    <SrcFiles>
        <item id="0">Ctrl1Err_h</item>
        <item id="1">Ctrl1Err_types_h</item>
        <item id="2">rtwtypes_h</item>
        <item id="3">Ctrl1Err_c</item>
```

```
        <item id="4">Ctrl1Err_private_h</item>
        <item id="5">Ctrl1Err_data_c</item>
    </SrcFiles>
    <header id="0">
        <item>Ctrl1Err_h</item>
    </header>
    <body id="3">
        <item>Ctrl1Err_c</item>
    </body>
    <Inputs structName="Ctrl1Err_U">
        <item dim="3">A</item>
        <item dim="3">B</item>
        <item dim="3">C</item>
        <item dim="9">C1</item>
    </Inputs>
    <Outputs structName="Ctrl1Err_Y">
        <item dim="3">RobTraj</item>
    </Outputs>
    <Functions>
        <Initialize>Ctrl1Err_initialize();</Initialize>
        <Step>Ctrl1Err_step();</Step>
        <Terminate>Ctrl1Err_terminate();</Terminate>
    </Functions>
</interface>
```

This file is required because every generation process ignores all the others, particularly, the last one, which should glue all the code to form PikeOS processes and partitions, does not know anything about the code interface of the subsystems. An alternative to the XML format might be an RTW file. The drawback of this approach is that now also the Partitions generation must be done through a TLC file and Matlab, whereas, with an XML file there is a free choice about the code generation tool. Moreover, via XML file is easier to implement a heterogeneous code generation tool.

Even though any other code generation tool can be used, the Partition generation is implemented used a TLC file for seamless integration with the current workflow.

4.4 Native Application Generation

For generating the code of each PikeOS Process (one for each Partition), another template is required. It must take the previously generated code and transform in threads than pack them in Processes. This task is performed by another TLC script.

The process is driven by a Matlab script that parses every XML file that has been generated by the custom System Target File enriching them with the information stored in the configuration file. The result of this operation is the previously created class structure with additional information, i.e.:

- Threads
  - Code interfaces
  - Communications
  - Affinity Mask
  - Priority
  - Partition Identifier
- Partitions

The top level class provides a public method to export all the threads inside a given partition as RTW file. This method is useful to feed the TLC script.

4.4.1 PikeOS Process Generation

Every PikeOS Partition is made by the Main thread and several child threads. The main thread is loaded by the PikeOS kernel, and it is responsible for initializing all the children threads. Therefore, to implement the proposed software architecture, the main thread perform the following operations:

1. Lower its priority to the lowest in its partition to let child threads execute their initialization code.
2. Creates all the child threads.
3. Increase its priority to the highest in its partition.
4. Enter in an infinite loop where resume all child threads and wait for the next period activation.

The C code implementing this behavior in PikeOS is the following:

```
1 int main(void){
2     P4_thr_t thread_num[TNUM];
3
4     /* Lower main's prio in order to let threads creation */
```

```
5   p4_thread_set_priority(P4_THREAD_MYSELF, 1);
6
7   /* For each child thread */
8   for(int i=0; i<INUM; i++)
9     initThread((P4_thr_t*)&thread_num[i], %<Priority>,
     P4_CPUMASK_CPU_TO_MASK(<AffinityMask>), "<Name>", <Prototype>);
10
11  /* Once thread are created (so they are in STOPPED STATE) main thread
     can escalate its priority */
12  rc = p4_thread_set_priority(P4_THREAD_MYSELF, TASK_HIGH_PRIO);
13
14    /* Main loop */
15  for(;;) {
16    /* RESUME THREADS*/
17    for (int i=0; i<INUM; i++)
18      resumeTask(i, thread_num);
19
20    /* WAIT FOR NEXT PERIOD */
21    p4_sleep(P4_TIMEOUT_TP_PERIOD | P4_TIMEOUT_INFINITE);
22    }
23 }
```

Where `<AffinityMask>`, `<Name>` and `<Prototype>` are variable that change for each thread, like in TLC script. The functions `initThread` and `resumeThread` are custom functions. The first one creates a PikeOS thread and sets its affinity mask, the second one resume a thread from the STOPPED state.

When the main lower its priority to the lowest one, the scheduler ensures that every child thread executes its initialization code than the main execute again and elevates its priority. During normal operation, the main is the highest priority thread in the partition and resumes all the threads, therefore the scheduler ensures that no one of the child thread executes before the main has resumed all threads, so the correct behavior is satisfied.

The child threads have a similar structure. Every thread:

1. Once created, execute the initialize function of the subsystems it refers to.
2. Move to the STOPPED state, waiting for the first activation.
3. Enters into an infinite loop where. Take all the require output from the predecessors, execute the step function of the subsystem it refers to and goes again in the STOPPED state, waiting for the next activation.

The C code implementing this behavior is the following:

```
static void thread(void){
  /* Subsystem Initialize entry function */
    subsystem_initialize();

  /* Wait for the first activatio */
  p4_thread_stop(P4_THREAD_MYSELF);

    /* Continous loop */
    for(;;){
    /* Get predecessor output */
    subsystem_getInput();

        /* Step */
        subsystem_step();

        /* Wait for next period */
    p4_thread_stop(P4_THREAD_MYSELF);
  }

    /* Terminate - Never Reached */
    subsystem_terminate();
}
```

As discussed before, all the threads inside a process share the same virtual memory space, so an output variable for a subsystems is a global variable that can be freely retrieved by one of it successor. This communication, referred as *inter-partition communication*, might require synchronization while accessing shared resources.

4.4.2 Intra-Process Communication

Assuming a non-preemptive, FIFO priority-based scheduler and a correct priority assignment, there is no need to use any synchronization mechanism when two threads on the same core try to access a shared variable. This because the scheduler ensures that the thread with the lower priority (the reader) does not execute before the completion of the higher priority thread (the writer). When the access occurs between threads scheduled on different cores there is no guarantee about the read-write operation. There might be cases in which a reader tries to get the input variable before its predecessor wrote the last value. In this case, a synchronization mechanism must be placed to avoid it.

4.4.2.1 Spinlocks

Several synchronization mechanisms solve the inter-core communication issue, the most commonly used locks are *mutexes* and *semaphores*.Both mechanisms eventually block the caller thread by moving it to the WAITING state, meaning that the next schedulable thread in the queue becomes the CURRENT thread and execute. Suppose that thread τ_1 perform a lock on a mutex and moves to the WAITING state, then a thread τ_2 from the READY queue is scheduled. Might happen that there is a precedence relation $\tau_1 \rightarrow \tau_2$ that force τ_2 to block, waiting for τ_1 to complete. This can happen again with the next scheduled thread and so on. Therefore, this might lead to a sequence of threads first check if all the predecessors completed their execution and eventually blocks.

A mechanism that avoids this drawback is the *spinlock* (already described in section 2.1.10). With this mechanism the thread performs a busy-wait (basically loops) on the lock until it is unlocked, so does not move to the WAITING state. Therefore, whenever the spinlock is released, the thread is sure about the availability of all the predecessor data. This mechanism, which is very cost-effective when the thread is expected to wait for a small amount of time, does not introduce the preemption issue.

Therefore, the TLC script adds a call to `p4_spin_lock` on every read and a `p4_spin_unlock` on every write on shared variables that needs to be accessed by threads scheduled on different cores.

4.5 Operating System Configuration Generation

As discussed in the section 2.1.11, the Integration Project provide a way to configure the generation of the PikeOS boot image, including all the Partitions. The Integration Project uses the file `project.xml` to store all the configuration information. The generation of the whole file is complex, so, for the seek of simplicity only some XML snippets are generated. This does not lead to a loss of functionality because it is possible to develop a tool that, once an Integration Project has been generated by CODEO or the command line, can scan the `project.xml` File and add programmatically the snippets.

The snippets that are required are the following:

- Partition and Ports
- Channels
- Schedule Scheme

With a procedure analogue to the generation of the partitions, the workspace structure containing all the information is exported in the RTW File format and used by a TLC to generate these snippets.

4.5.1 Partitions and Ports

First and foremost Resource Partitions and and their Processes musts be defined. This is made by adding some `ComponentInstance` elements as child of the `ConfigurationDomainTable` tag. An example of the definition of a Resource Partition is the following:

```
<!-- ... -->
<Group name="Partition A">
  <!-- Partition Configuration -->
  <ComponentInstance name="ServerPartition" ref="empty_partition_default
    ">
    <ParameterValue name="PARTNAME" value="PartitionA"/>
    <VmitConfigurationTable>
      <VmitConfiguration condition="true" isReference="true">
        <Partition Abilities="" CpuMask="0" Identifier="$(PARTID)"
    MaxChildTaskCount="20" MaxFDCount="20" MaxPrio="62"
    MultiPartitionHMTableID="0" Name="$(PARTNAME)" SchedChangeAction="
    VM_SCHED_CHANGE_IGNORE" StartupMode="VM_PART_MODE_COLD_START"
    TimePartitionID="1">
          <FileAccessTable>
            <ComponentReference ref="ProcessA"/>
          </FileAccessTable>
          <MemoryRequirementTable>
            <ComponentReference ref="ProcessA"/>
          </MemoryRequirementTable>
          <ProcessTable>
            <ComponentReference ref="ProcessA"/>
          </ProcessTable>
          <QueuingPortTable>
            <ComponentReference ref="ProcessA"/>
          </QueuingPortTable>
          <SamplingPortTable>
            <SamplingPort Name="sport_0_src" RefreshRate="100"
    MaxMessageSize="12"  Direction="VM_PORT_SOURCE">
            </SamplingPort>
            <ComponentReference ref="ProcessA"/>
          </SamplingPortTable>
          <HMTable>
            <DefaultSwitch>
```

```
28                <Default Action="P4_HM_PAC_IDLE" Code="0" Level="
    P4_HM_LEVEL_USER" Notify="0"/>
29              </DefaultSwitch>
30            </HMTable>
31          </Partition>
32        </VmitConfiguration>
33      </VmitConfigurationTable>
34    </ComponentInstance>
35    <!-- Process Configuration -->
36    <ComponentInstance name="ProcessA" ref="PikeOS Native Process">
37      <ParameterValue name="SUPPORT_DEBUG" value="true"/>
38      <ParameterValue name="DESTNAME" value="ProcessA"/>
39      <ParameterValue name="NUM_ARGS" value="0"/>
40      <ParameterValue name="FILE" value="CUSTOM_POOL/pikeos-native/object/
    processA"/>
41      <AssignedDependencyTable>
42        <AssignedDependency cmp="muxa" provideId="CHANNEL_01" dependId="
    DEBUG_PROV" />
43      </AssignedDependencyTable>
44    </ComponentInstance>
45  </Group>
46  <!-- ... -->
```

In this snippet all relevant information are defined, i.e. 1. the partition and its name (e.g. PartitionA), 2. the process belonging to this partition (e.g. ProcessA), 3. the ports and their identifier (e.g.`sport_0_src`). In this example both the Process and the Partition are grouped in a group called `Partition A` (which has no impact on the boot image creation) but it is displayed in the CODEO Project Configuration.

Later on in the `project.xml` File, the complete enumeration of the Resource Partitions is listed as child elements of `PartitionTable`, a configuration example is:

```
1 <PartitionTable>
2   <!-- ... -->
3   <ComponentReference ref="PartitionA"/>
4   <ComponentReference ref="PartitionB"/>
5   <!-- ... -->
6 </PartitionTable>
```

Here are defined two Resource Partitions, `PartitionA` and `PartitionB`.

4.5.2 Channels

Once Resource Partition and their Ports are declared, Channels must connect them allowing two Partition to communicate. For example a communication between partition π_2 and partition π_3 through ports `sport_0_dst` (end point of the communication) and `sport_0_src` (starting point of the communication) is the following:

```
<ConnetctionTable>
<!-- ... -->
  <PartitionChannelTable>
    <!-- ... -->
    <Channel PortType="VM_PORT_SAMPLING">
      <DestinationPortRef PortName="sport_0_dst" PartitionID="3">
            </DestinationPortRef>
      <SourcePortRef PortName="sport_0_src" PartitionID="2">
            </SourcePortRef>
    </Channel>
    <!-- ... -->
  </PartitionChannelTable
  <!-- ... -->
</ConnetctionTable
```

The `Channel` tag, with its children, is a child of the tag `PartitionChannelTable` in the `ConnetctionTable`.

4.5.3 Schedule Scheme

The schedule scheme is used to implement the Partition Schedule. As discussed in section 2.1.7, a Schedule Scheme is by Time Windows, each of them has an identifier, a starting time and a duration. The default scheme loaded by the kernel is called `SCHED_BOOT`. An example of the schedule snippet is the following.

```
<ScheduleTable>
  <ScheduleScheme Name="SCHED_BOOT">
      <WindowTable>
          <Window Identifier="1" Start="0" Duration="200"
              TimePartitionID="1" Flags="VM_SCF_PERIOD">
          </Window>
          <Window Identifier="2" Start="200" Duration="200"
              TimePartitionID="2" Flags="VM_SCF_PERIOD">
          </Window>
          <Window Identifier="0" Start="400" Duration="300"
              TimePartitionID="0">
          </Window>
```

```
13          </WindowTable>
14      </ScheduleScheme>
15  </ScheduleTable>
```

In this schedule the remaining time after the schedule of TP_1 and TP_2 and before the start of a new Major Time Frame (Hyper-period) is allocated to the TP_0. The flag `VM_SCF_PERIOD` mark that time window as the start of a new period. This flag is used by PikeOS kernel to notify the Main threads to wake up.

Chapter 5

Test and Validation

5.1 Example model

The example from the EMC2 project not relatively straightforward for showing the tool capabilities. Therefore, as test example model, let consider a quad-rotor flight control with a pan/tilt camera on it shown in figure 5.1. The pan and tilt camera is moved by two servo motors controlled by two PID that do not communicate with the flight control of the drone. The adopted drone control scheme is taken from [Cor11] with minor changes introduced to comply with our design restrictions. The original model in [Cor11] contains different functional loops, each of them has been included in a Simulink subsystem representing a task. The quad-rotor and the motors dynamics that has been used during the simulation have been substituted to enable code generation. The *Quadrotor* block contains the control for its motors and the interface to its sensors. The motors blocks are interface blocks to their encoders. The pan/tilt motor control has a sample time of 0.05 seconds, the drone flight control a sample time of 0.1 seconds and the quad-rotor block a sample time of 0.01 seconds.

This example is oriented to demonstrate the capabilities of the framework, so it does not focus on the performance of the controller. When the model is ready to be used, the framework GUI (fig. 5.2) can be started with the command `START`. Here the designer can configure all the additional parameters that are required for each task. In particular, he can add all the resources on the platform, set up the optimization engine and specify for each task:

1. The Worst-Case Execution Time
2. The Criticality level (A to E)

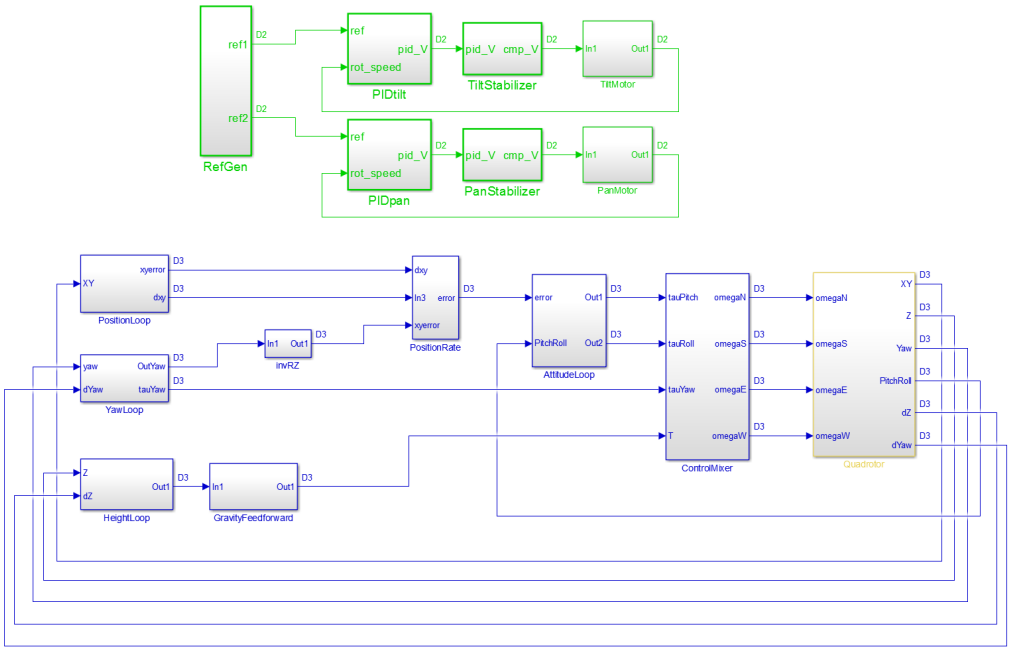


Fig. 5.1 Example Model

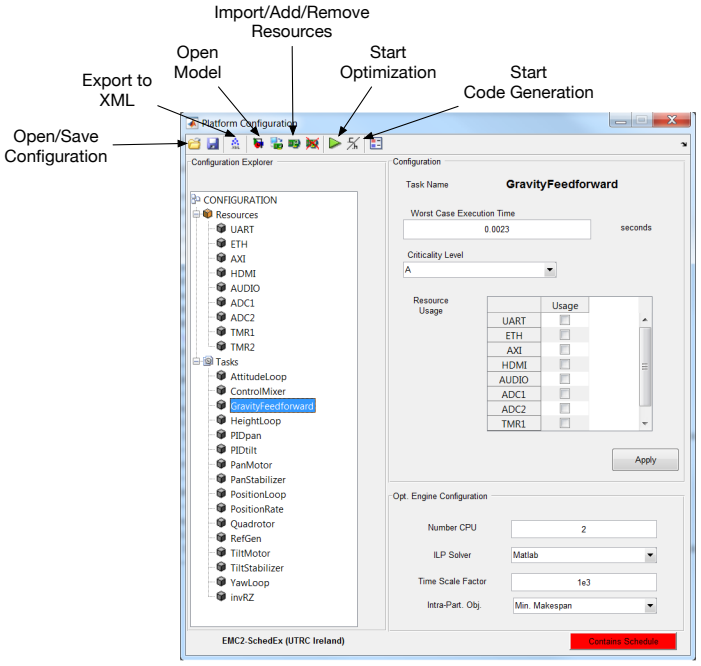


Fig. 5.2 Tool Graphic User Interface (GUI)

Task ID	Task Name	Period	WCET	Criticality	Resources
1	AttitudeLoop	0.1	$9.135 \cdot 10^{-4}$	A	–
2	ControlMixer	0.1	$9.8412 \cdot 10^{-4}$	A	–
3	GravityFeedforward	0.1	$8.7592 \cdot 10^{-4}$	A	AXI
4	HeightLoop	0.1	0.0012	A	AXI
5	PIDpan	0.05	$9.7148 \cdot 10^{-4}$	B	–
6	PIDtilt	0.05	$8.5084 \cdot 10^{-4}$	B	–
7	PanMotor	0.05	$5.4351 \cdot 10^{-4}$	B	AXI, ADC1
8	PanStabilizer	0.05	$5.6589 \cdot 10^{-4}$	B	–
9	PositionLoop	0.1	$9.9352 \cdot 10^{-4}$	A	UART
10	PositionRate	0.1	$5.9584 \cdot 10^{-4}$	A	–
11	Quadrotor	0.01	0.0033	A	AXI
12	RefGen	0.05	$4.8490 \cdot 10^{-4}$	B	UART
13	TiltMotor	0.05	$4.0925 \cdot 10^{-4}$	B	AXI, ADC2
14	TiltStabilizer	0.05	$9.0944 \cdot 10^{-4}$	B	–
15	YawLoop	0.1	$8.2928 \cdot 10^{-4}$	A	–
16	invRZ	0.1	$9.5648 \cdot 10^{-4}$	A	–

Table 5.1 Example Model Task-set

3. The Resources the task uses

In our example, the tasks are configured as shown in table 5.1. The WCETs are randomly selected to make the task-set schedulable, otherwise, the inter-partition optimization problem states that no feasible solution is available. The task-set will be allocated on a dual-core processor and scheduled minimizing the makespan.

5.1.1 Feedthrough relationships

To fully understand the resulting functional model it is important to highlight the feedthrough relationship among ports of each subsystem. In figure 5.3 are shown all the paths. This relationship is automatically extracted from the model by a Matlab script and used to build the functional model.

5.2 Functional Model Extraction

Once the configuration is complete (i.e. WCET, Resources and Criticality), the system designer can start the scheduling process. The first step is the DAG extraction, which is

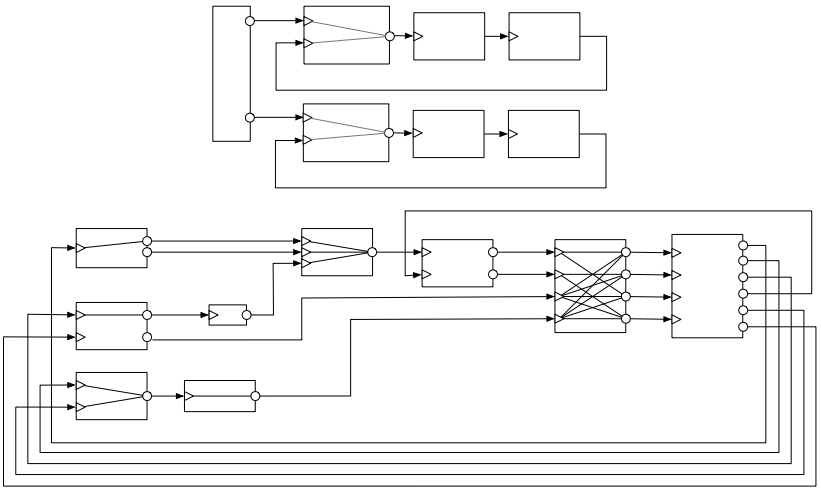


Fig. 5.3 Feedthrough Relationships

transparent to the designer. Indeed, it is extracted as soon as the GUI (or the model) is opened. The Direct-Acyclic Graph representing the task-set is shown in figure 5.4. Each

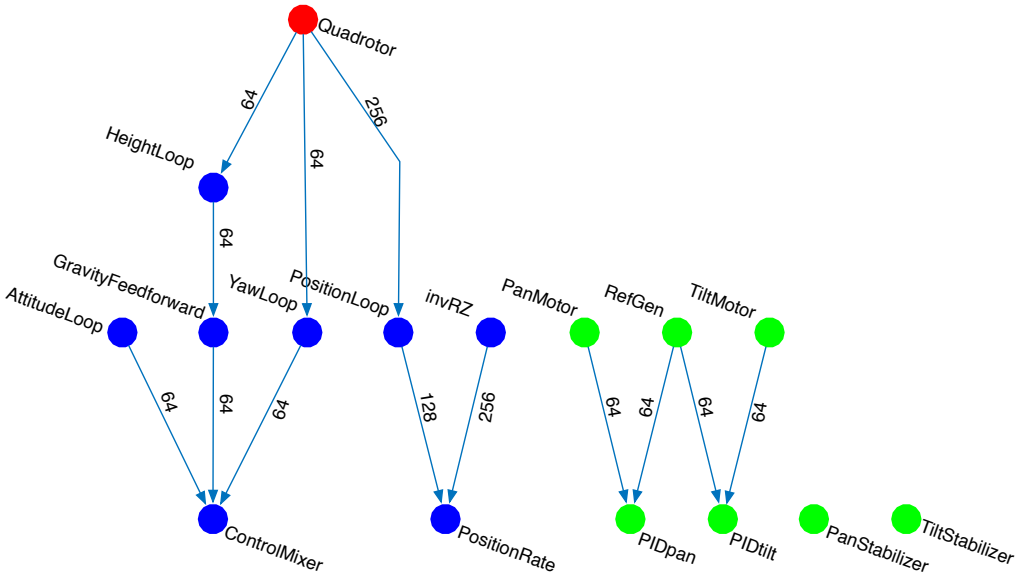


Fig. 5.4 Example Functional model - Task-set

color represent a different sample time, the edges the dependency among tasks. From the Simulink model, all the lines connecting two blocks where the input port of the destination block is not in a feedtrough relationship, has disappeared. This is the representation of the model used by the partitioning and scheduling algorithm.

Partition	Tasks	Period
P1	1,2,10,16,3,4,9,15	0.1
P2	5,6,7,8,12,13,14	0.05
P3	11	0.01

Table 5.2 Partitions

5.3 Scheduling

The DAG, enriched with the information manually configured by the designer is the main input for the partitioning and scheduling algorithm. As already explained in section 2.3, the scheduling process is made by three phases:

1. Partitioning
2. Intra-Partition Scheduling
3. Inter-Partition Scheduling

5.3.1 Partitioning

The partitioning algorithm implements some form of robust partitioning of the task-set graph, the result of this step is figure 5.5 and is called P-DAG. While grouping tasks in partitions, some edge in the task-set cut the boundary of the partition, this edge represents a dependency among two partitions. All the edges in the P-DAG represent the partial order of execution

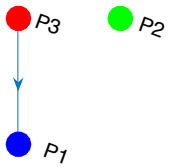


Fig. 5.5 Partition Graph, P-DAG

imposed by the flow preservation requirement. The how tasks are redistributed in partitions is shown in table 5.2.

Partition	Solution Time	Number of Constraints	Number of Variables (continuous)
P1	0.34 seconds	313	161(16)
P2	0.09 seconds	237	127(14)
P3	0.03 seconds	3	6(2)

Table 5.3 MILP formulations

5.3.2 Intra-Partition Scheduling

Each node of the P-DAG represent a partition so, each partition contains some tasks, the subgraph of the task-set graph related to a partition is called T-DAG. The allocation and scheduling of each T-DAG is called *Intra-Partition Scheduling*. For each T-DAG the MILP optimization problem is solved on the relative subgraph (table 5.3). Note that task three and four cannot execute in parallel in partition one because both use the AXI interface (resource). Similarly, task 7 and 13 cannot execute in parallel in partition 2. This constraint is encoded in the incompatibility matrix. The resulting schedule is depicted in figure 5.6.

Again, the arrows among tasks represent the dependency relationship.

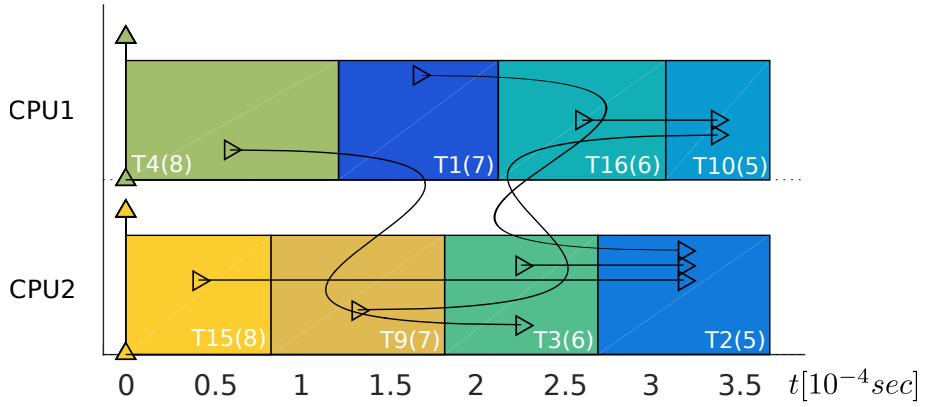
5.3.3 Priority Assignment

When a schedule is obtained another optimization problem assigns priorities to each task as described in section 3.6. In particular figure 3.4 shows that some priority value can be left unused for allowing some low priority task to be executed, in this example we use an offset of 5 for solving the optimization problem. The resulting assignment is shown in figure 5.6, tasks priorities are enclosed by parenthesis.

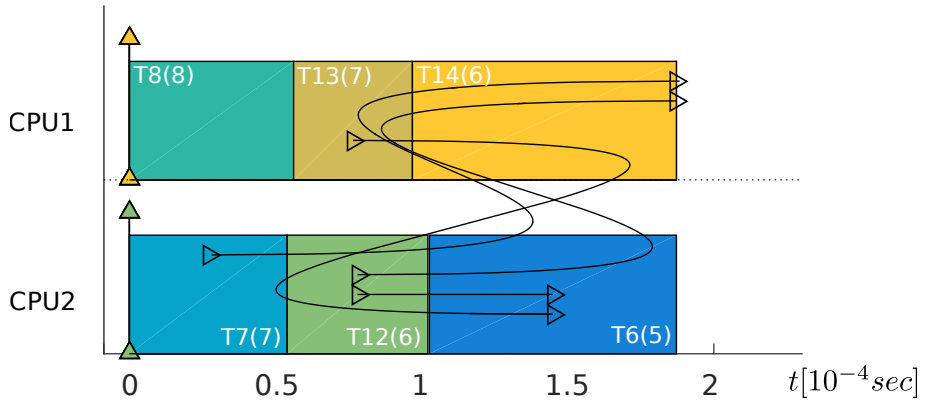
5.3.4 Inter-Partition Scheduling

The final step of the scheduling process is the inter-partition scheduling. As described in section 3.5.2, the Bratley algorithm is exploited to find the optimal solution. To include precedence constraints and periodicity, the P-DAG must be factorized. In the current example the Hyper-period is 0.1 seconds, so, inside this interval partition P_1 execute only one, partition P_2 execute twice and partition P_3 execute 10 times. The resulting factorized P-DAG is shown in figure 5.7.

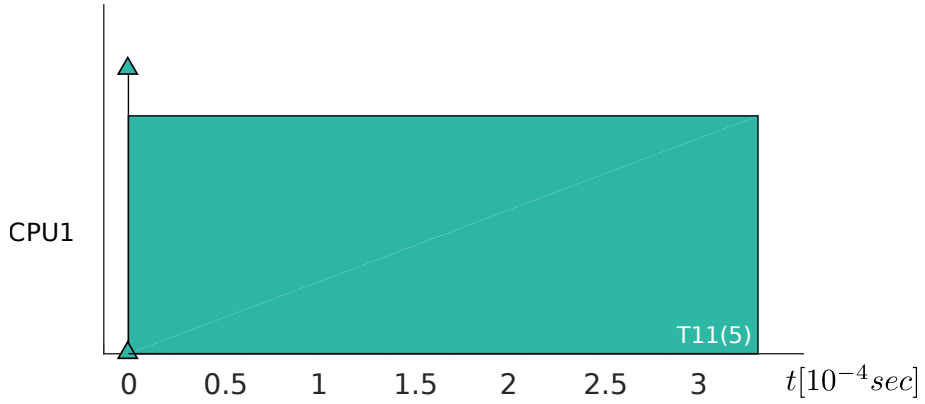
Thanks to Lemma , has been proven that if release times and deadlines are consistent with the precedence relation, then any normal one-processor schedule that satisfies the release



(a) Partition 1 Schedule



(b) Partition 2 Schedule



(c) Partition 3 Schedule

Fig. 5.6 Intra-Partition Schedule

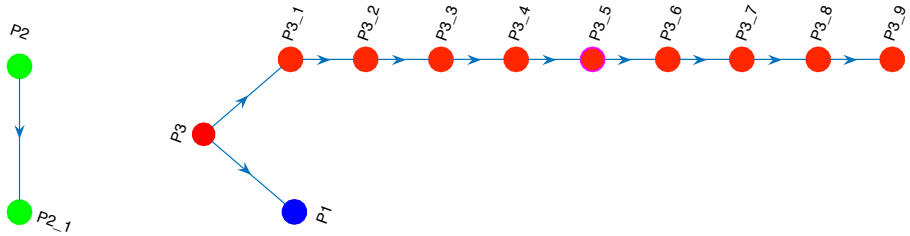


Fig. 5.7 Factorized Partition Graph

Partition ID	WCET	Release Time	Due Date	Activation
P_1	0.0037	0.0033	0.1	0.0052
P_2	0.0019	0	0.05	0
P_3	0.033	0	0.01	0.0019
P_2^1	0.0019	0.05	0.01	0.05
P_3^1	0.033	0.01	0.02	0.01
P_3^2	0.033	0.02	0.03	0.02
P_3^3	0.033	0.03	0.04	0.03
P_3^4	0.033	0.04	0.05	0.04
P_3^5	0.033	0.05	0.06	0.0519
P_3^6	0.033	0.06	0.07	0.06
P_3^7	0.033	0.07	0.08	0.07
P_3^8	0.033	0.08	0.09	0.08
P_3^9	0.033	0.09	0.11	0.09

Table 5.4 Partition-set

times and deadlines must also obey the precedence relation, so a release time and a due date can be assigned to each partition according to formula 3.19. They are summarized in table .

The Partition-set in table 5.3.4 (except the last column) feeds the Bratley algorithm that enumerate all the possible feasible solution and select the optimal one. Int inter-partition schedule is shown in figure 5.8. All the activation times for each partition is listed in table 5.3.4 (last column).

5.3.5 Theoretical Performance

In order to assert how well the model can be run on multi-core, various metrics need to be defined first. Using the right metric makes it possible to determine how one would benefit from parallelizing the model as well as how various actions like changing the model changed the performance. In this work only widely used performance metrics for parallel systems are

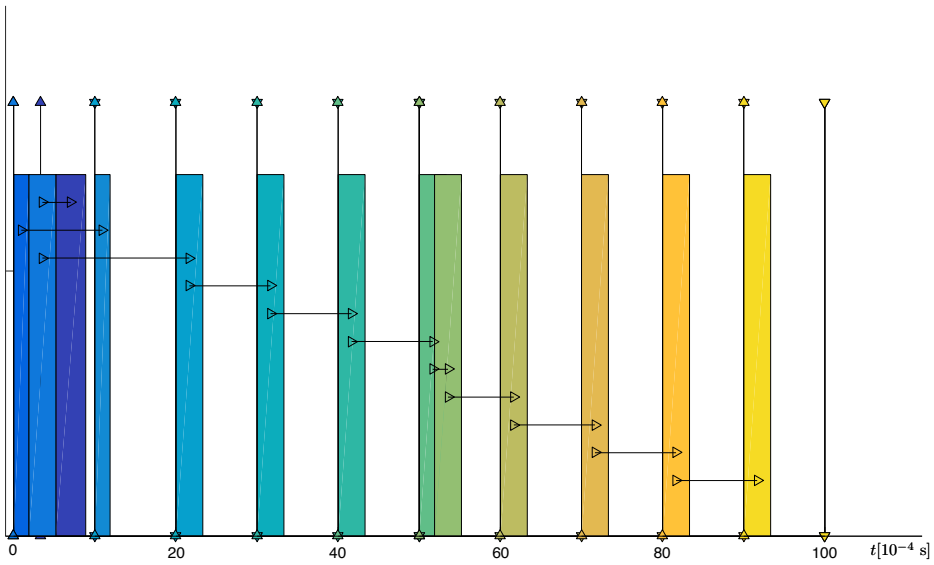


Fig. 5.8 Inter-Partition Schedule

used, since a system's performance is the main motivation behind switching to a multi-core architecture. These include the makespan (mentioned in the previous section), the *speed-up* and the *efficiency* [Gra03].

When used for parallel systems, the makespan can be divided into two metrics: *sequential makespan* and *parallelized makespan*. The sequential makespan is the execution time required by the task-set to execute on a single-core. Used together with the parallelized makespan, which is the execution time on multiple cores, various other metrics like the speed-up can be calculated.

Speed-up is defined as the ratio of the elapsed time when executing a program on a single processor (the sequential makespan W_1) to the execution time when p processors are available (the parallelized makespan W_p)[EZL89]. Throughout this work, it is defined as,

$$S(p) = \frac{W_1}{W_p} \tag{5.1}$$

With this metric it is possible to assert how the execution time of a system as improved by changing the number of cores. In theory, the speed-up can never exceed the number of processors p, the best case. This introduces the *efficiency*. It is defined as the average utilization of the p allocated processors [EZL89]. This value tells us how much average speedup is gained by investing in p-core, enabling us to declare how worthwhile a multi-core

Partition	Speed-up	Efficiency
P_1	1.9927	0.99
P_2	1.9927	0.99
P_3	1	0.5

Table 5.5 Performance

system is for a given system. Formally it is defined as:

$$E(p) = \frac{S(p)}{p} \tag{5.2}$$

Efficiency can only reach values between 0 and 1. Programs with linear speed-up have an efficiency of 1. It is worth mentioning that in practice, linear speed-up is not achievable in practice since adding more processors increases the communication time between processors as well as the time waiting for shared resources to unlock, which are not considered in this model.

For each partition both the speed-up and the efficiency can be computed, they are summarized in table 5.5. The MILP problem, since finds the optimal solution, does a good job in exploiting all the possible time in the partition on both available cores. There is no idle time in the intra-partition schedule (see fig. 5.6) so the speed-up is approximately 2 for partition one and partition two.

At this point all the information about the scheduling are known, so the code generation process can be started.

5.4 Code Generation

Once a feasible schedule is available, the system designer can start the code generation process pressing the button *Start Code Generation* in the framework GUI. This operation will save the current schedule to an XML file and call the code generation entry point. After parsing the file, the model is adapted to the communication model described in the exported configuration.

5.4.1 Adapted model

As described in section 4.1, before generating the C code, all the subsystem ports that are communication data to subsystem assigned to another partition needs to be substituted with the correct operation on a Sampling Port. This is automatically done by the script after parsing the configuration file. In figure 5.9 it is possible to see that all the lines connecting `Quadrotor` with all other blocks disappeared. As expected, all partitions are isolated one from each other, meaning that there are no lines between blocks. The next step generates the

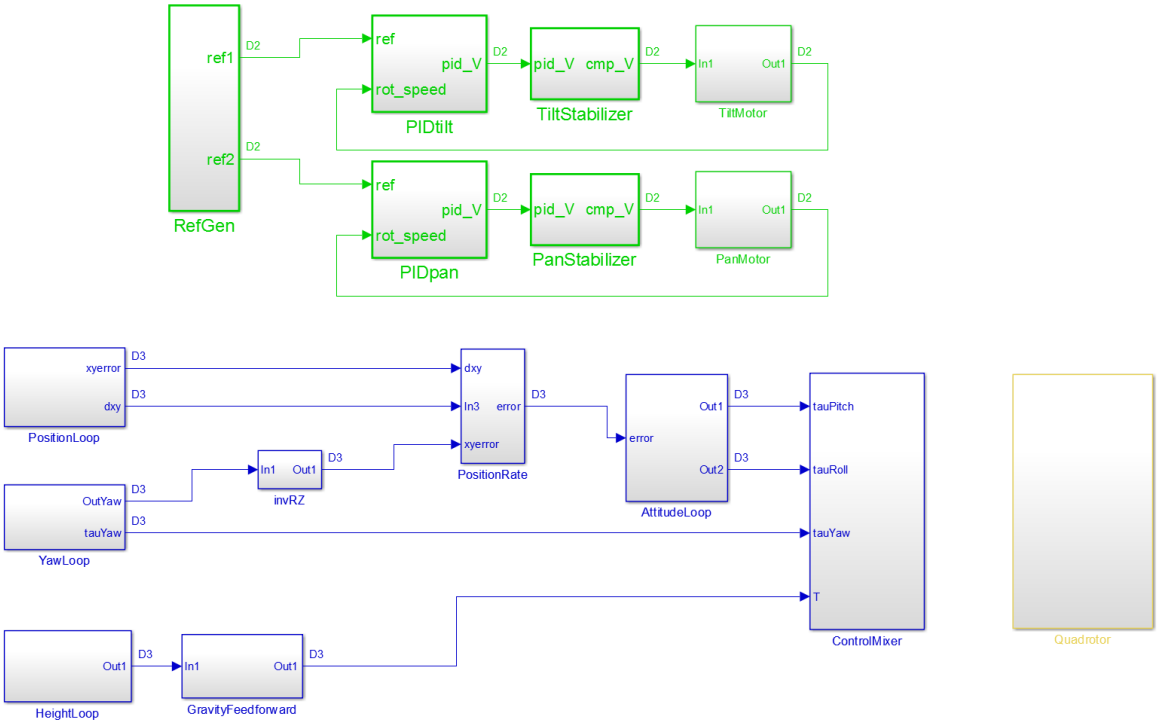


Fig. 5.9 Adapted Model

C code for each block.

5.4.2 Partition Main Threads

To generate the code of each block is used the System Target file described in section 4.3. It generates the C code for each single block as Embedded Coder would do, also it generates an XML file describing the structure of the code. When all the blocks have been generated, all these XML files are collected and parsed to correctly generate the code of each partition.

As described in section 4.4.1, each Native PikeOS process is made by the main thread and all the thread composing that partition. The code for this two entities is inside separate files for each partition, e.g. `P1_main.c`, `P2_main.c`, `P3_main.c`. As an example let consider the code of the main thread of partition one.

```
1  extern int main(void){
2    P4_uid_t me;
3    P4_e_t rc;
4    P4_thr_t thread_num[TNUM];
5
6    /* Spinlocks Initialization */
7    p4_spin_init(&AttitudeLoop1_ControlMixer1_spinlock);
8    p4_spin_init(&AttitudeLoop2_ControlMixer2_spinlock);
9    p4_spin_init(&HeightLoop1_GravityFeedforward1_spinlock);
10   p4_spin_init(&PositionLoop1_PositionRate1_spinlock);
11   p4_spin_init(&PositionLoop2_PositionRate2_spinlock);
12   p4_spin_init(&YawLoop1_invRZ1_spinlock);
13
14   /* Lower main's prio in order to let threads creation */
15   rc = p4_thread_set_priority(P4_THREAD_MYSELF, 1);
16   ASSERT(rc == P4_E_OK, "Main->p4_thread_set_priority() failed with rc
       == 0x%02x", rc);
17
18   /* Params: Tnumber, prio, affinitiMask, name, entryPt */
19   initThread((P4_thr_t*)&thread_num[0], 7, P4_CPUMASK_CPU_TO_MASK(0), "
       AttitudeLoop", AttitudeLoop_thread);
20   initThread((P4_thr_t*)&thread_num[1], 5, P4_CPUMASK_CPU_TO_MASK(1), "
       ControlMixer", ControlMixer_thread);
21   initThread((P4_thr_t*)&thread_num[2], 6, P4_CPUMASK_CPU_TO_MASK(1), "
       GravityFeedforward", GravityFeedforward_thread);
22   initThread((P4_thr_t*)&thread_num[3], 8, P4_CPUMASK_CPU_TO_MASK(0), "
       HeightLoop", HeightLoop_thread);
23   initThread((P4_thr_t*)&thread_num[4], 7, P4_CPUMASK_CPU_TO_MASK(1), "
       PositionLoop", PositionLoop_thread);
24   initThread((P4_thr_t*)&thread_num[5], 5, P4_CPUMASK_CPU_TO_MASK(0), "
       PositionRate", PositionRate_thread);
25   initThread((P4_thr_t*)&thread_num[6], 8, P4_CPUMASK_CPU_TO_MASK(1), "
       YawLoop", YawLoop_thread);
26   initThread((P4_thr_t*)&thread_num[7], 6, P4_CPUMASK_CPU_TO_MASK(0), "
       invRZ", invRZ_thread);
27
28   /* Once thread are created (so they are in STOPPED STATE) main thread
       can escalate its priority */
29   rc = p4_thread_set_priority(P4_THREAD_MYSELF, TASK_HIGH_PRIO);
```

```
30   ASSERT(rc == P4_E_OK, "Main->p4_thread_set_priority() failed with rc
       == 0x%02x", rc);
31
32   vm_cprintf("\nALL THREADS INITIALIZED\n");
33   vm_cprintf("
       ------------------------------------------------------------------------\n"
       );
34
35     /* Main loop */
36   for(;;) {
37       /* RESUME THREADS*/
38     for (int i=0; i<TNUM; i++)
39       resumeTask(i, thread_num);
40
41     /* WAIT FOR NEXT PERIOD */
42     p4_sleep(P4_TIMEOUT_TP_PERIOD | P4_TIMEOUT_INFINITE);
43   }
44 }
```

The threads it creates matches the ones in 5.2 for P_1 with the priorities in figure **??**. It also initializes some spinlocks, one for each inter-core communications, each of them can be graphically seen in figure 5.6 by edges between tasks scheduled on different cores.

After creating and letting each thread to execute its initialization code (lines 15-30) the main thread enters an infinite loop where each thread is resumed every time a new period starts.

As an example let consider the code of the thread about `AttitudeLoop` (thread 1), this is interesting because is used both data directly from a thread in the same process and data coming from the sampling port. The code of this thread is the following.

```
1  static void AttitudeLoop_thread(void){
2        P4_e_t rc;
3
4        AttitudeLoop_initialize();
5        vm_cprintf("AttitudeLoop initialized...\n");
6        p4_thread_stop(P4_THREAD_MYSELF);
7
8        /* Continous loop */
9    for(;;){
10     /* Input update */
11     AttitudeLoop_U.pitchrolldmd[0] = PositionRate_Y.error[0];
12     AttitudeLoop_U.pitchrolldmd[1] = PositionRate_Y.error[1];
13
```

```
14      /* Step */
15      AttitudeLoop_step();
16
17      /* Unlock Spinlocks */
18      P4_spin_unlock(&AttitudeLoop1_ControlMixer1_spinlock);
19      p4_spin_unlock(&AttitudeLoop2_ControlMixer2_spinlock);
20
21      /* Wait for next period */
22      p4_thread_stop(P4_THREAD_MYSELF);
23    }
24
25    /* Terminate - Never Reached */
26    AttitudeLoop_terminate();
27  }
```

On lines 11-12 it takes the data directly from the shared structure. The data from the sampling port are retrieved inside the `AttitudeLoop_step()` because the code for reading from the port is generated directly from RTW thanks to the Custom Block. Indeed, inside the file `AttitudeLoop.c`, the following code appears.

```
1 P4_e_t rc1 = vm_sport_read(&AttitudeLoop_sport1_d, &AttitudeLoop_B.
    PitchRoll[0],
2     4*sizeof(real_T), &read_size, &validity);
```

When the step function of the subsystem has been executed, the successor that is executing on another core can be notified. For this reason, two spinlocks are unlocked on lines 18-19 notifying the thread `ControlMixer` that the data are available. Finally, the thread goes in the STOPPED state on line 22, waiting for the next activation from the main thread.

5.4.3 Integration Project Snippets

Out of the code generation process, some XML snippets are generated to help the configuration of the Integration Project. For the sake of conciseness, they are nor reported here because they are quite long. For example, schedule scheme is 64 lines long since in every moment there is at least one partition active, this means that for each gap in the schedule of figure 5.8 the TP_0 is statically allocated for non-real time operation (e.g. muxa and other services).

Chapter 6

Conclusions

6.1 Contributions

Multi-core platforms have been introduced in many different settings, but so far they have not been utilized in the domain of safety-critical avionic real-time systems. A substantial amount of work has been put into the evaluation of hypervisors to address the security and safety for such applications.

In this thesis, the area of code generation for mixed-critical application in multi-core embedded systems is taken into account. For this purpose, a Model-Based design framework, supported by Simulink, has been developed. It is a proof-of-concept integrated tool that allows the designer to design and deploy hard real-time, mixed-critical applications with the assistance of optimization problems and code generation.

6.2 Future works

The planned future work has two separate tracks. One is related to the optimization framework and aims at extending it with support for more comprehensive, robust partitioning algorithm. The second track of future work relates to the code generation process.

6.2.1 Scheduling and allocation

A current limitation of this work is that the validity of the proposed design framework has been extensively tested in simulation, but no experimental validation was conducted. Extensive testing and robustness improvements are needed to improve the effectiveness of the partitioning and scheduling framework. Improvements on the partitioning algorithm are necessary to improve safety and security of the implemented code. One step ahead in

this direction might find some heuristics that estimates the impact of one step to the others. Another approach might be trying to encode each phase into a single MILP optimization problem. A possible drawback of the second method is that it can lead to problems which complexity makes them barely solvable for a small task-set.

A possible improvement for the scheduling algorithm might be to analyze more in depth how cache and memory interferences and delays can be minimized by the partitioning or the scheduling. An additional step might be added, in the current implementation there is a one-to-one map between subsystems and tasks, there can be cases in which this is not the optimal choice.

Another current limitation of the current implementation it the needs of the system designer to specify the Worst-Case Execution Time of each task.

Worst Case Execution Time analyses aim at determining an upper bound for a task execution time. Usually, the result of a WCET analysis is an upper approximation of the exact WCET which is nearly impossible to determine for real life Software. Simple architectures allow WCET determination using static analysis techniques using a model of execution. That means that the analyzed software is not executed but compiled and analyzed. On complex COTS processors architectures, it is not possible to determine an accurate model. Today, an alternative method is used. A worst case scenario is defined from an analysis performed on the Airborne Software. The execution time is measured under this scenario and is further corrected with parameters taking into account variability during operations. Timing analysis is tough in multi-core COTS due to the lack of information on the processor behavior. It may lead to a pessimistic estimation of those parameters.

There are a plenty of free and commercial tool for the timing analysis, some of them are directly integrated into the Simulink environment. These tools can be used to alleviate the workload of the system designer that can focus more on the architecture optimization. However, there is a lack of research on WCET estimation under faulty conditions on safety-critical, multi-core COTS platforms that require temporal partitioning. This research is needed to deploy multi-core platforms in the avionics domain safely and for certification authorities to accept and approve their usage.

Another path to explore is the area of formal verification instead of empirical measurement-based methods. Formal methods with temporal specifications can be used to prove that a given set of partitions, with their tasks, can never reach hazardous states. This proof would be another great help for certification authorities.

6.2.2 Code Generation

The biggest limitation of the current code generator is that it does not support non-periodic tasks. This is not an intrinsic limitation of the approach, instead, it has not been implemented yet. All the non-periodic tasks can be placed on PikeOS TP_0 thanks to the gaps left in the priority assignment. At the moment has not been implemented a semantics to mark those tasks.

An interesting improvements might be the support for the Hardware-in-the-Loop (HIL) Simulation. HIL simulation, which is quite common in the Aviation industry, is a type of real-time simulation, it can be used to test the controller design. In HIL simulations, the real controller responds to realistic stimuli coming from the virtual plant included in the model. HIL simulation can add great value to WCET estimation.

Another possible improvement might be improving the resource usage in the generated code. For example, if a block is sending data to another block via two different ports, and these are converted into sampling ports due to the partitioned architecture, it can be useful to merge the two data into the same port.

Moreover, the work-flow can be extended to other code generators, and eventually to hand-coded applications. The step in this direction is to model the system in *SysML* [sys] which is a general-purpose modeling language for engineering systems (defined as an extension of UML) and to use a more generic code generator, such as Acceleo [Acc], to generate the glue code.

References

[ABB09] James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg. Multicore operating-system support for mixed criticality. 2009.

[Acc] Acceleo. http://www.eclipse.org/acceleo/.

[AG09] François Armand and Michel Gien. A practical look at micro-kernels and virtual machine monitors. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–7. IEEE, 2009.

[4] Aeronautical Radio Inc (ARINC). ARINC 653, avionics application software standard interface.

[Bar09] Sanjoy Baruah. Mixed criticality schedulability analysis is highly intractable. 2009.

[BBD+12] Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *ECRTS*, 2012.

[BEP+13] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer Science & Business Media, 2013.

[Ber12] Cyber-physical systems conceptual map, 2012. http://cyberphysicalsystems.org/.

[BFR71] Paul Bratley, Michael Florian, and Pierre Robillard. Scheduling with earliest start and due date constraints. *Naval Research Logistics Quarterly*, 18(4):511–519, 1971.

[Blo] Block target file methods. https://www.mathworks.com/help/rtw/tlc/block-target-file-methods.html?searchHighlight=BlockInstanceSetup&s_tid=doc_srchtitle.

[BMOS10] Devesh Bhatt, Gabor Madl, David Oglesby, and Kirk Schloegel. Towards scalable verification of commercial avionics software. In *AIAA Infotech@ Aerospace 2010*, page 3452. 2010.

[But11] Giorgio Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[CMDN15] Fabio Cremona, Matteo Morelli, and Marco Di Natale. Tres: a modular representation of schedulers, tasks, and messages to control simulations in simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1940–1947. ACM, 2015.

[COQ] Opensynergy coqos. http://www.opensynergy.com/en/Products/COQOS.

[Cor11] Peter Corke. *Robotics, Vision and Control Fundamental Algorithms in MATLAB*. 1. ed. 2011, corr. 2. print. Berlin: Springer, 2011.

[EJL+03] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EM15] EURO-MILS. Secure european virtualisation for trustworthy applications in critical domains, common criteria protection profile, 2015.

[emc] Artemis emc2 european project. http://www.artemis-emc2.eu/.

[EN16] R. Ernst and M. Di Natale. Mixed criticality systems - a history of misconceptions? *IEEE Design Test*, 33(5):65–74, Oct 2016.

[Ene] Enea hypervisor. http://www.enea.com/software/products/hypervisor.

[eVM] Tenasys evm for windows. http://www.tenasys.com/products/evm.php.

[EZL89] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[FMC] Analog device ad-fmcmotcon2-ebz motor evaluation kit. http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/Eval-FMCMOTCON2.html.

[GFF13] Giovani Gracioli, Rodolfo Fröhlich, Antônio Augustoand Pellizzoni, and Sebastian Fischmeister. Implementation and evaluation of global and partitioned scheduling in a real-time os. *Real-Time Systems*, 49(6):669–714, 2013.

[Gra03] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[GWF10] Thomas Gaska, Brian Werner, and David Flagg. Applying virtualization to avionics systems—the integration challenges. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5–E. IEEE, 2010.

[GZ12] Zonghua Gu and Qingling Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. 2012.

[Hor07] Chris Horne. Understanding full virtualization, paravirtualization and hardware assist. *White paper, VMware Inc*, 2007.

[INT] Green hills integrity-178b. http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[JFG+12] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 7A4–1–7A4–9, Oct 2012.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[Kle13] David Kleidermacher. Chapter 7 - system virtualization in multicore systems. In Bryon Moyer, editor, *Real World Multicore Embedded Systems*, pages 227 – 267. Newnes, Oxford, 2013.

[KRF+14] Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems. In *RTNS*, 2014.

[Lab] Ni labview. http://www.ni.com/labview/.

[Lap92] Jean-Claude Laprie. Dependability: Basic concepts and terminology. In *Dependability: Basic Concepts and Terminology*, pages 3–245. Springer, 1992.

[LKB77] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, 1:343–362, 1977.

[Lyn] Lynxos-178 certified rtos. http://www.lynx.com/products/.

[MEA+10] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1864–1871, Washington, DC, USA, 2010. IEEE Computer Society.

[Mel] Global scheduling in multiprocessor real-time systems. http://cyberphysicalsystems.org/.

[Mod] Modeler. http://www.mathworks.com/products/simulink/.

[Mor15] Matteo Morelli. Simple simulink to ecore exporter. https://github.com/m-morelli/see_simple, 2015.

[NIH] Ni real-time hypervisor. http://www.ni.com/pdf/manuals/375174b.pdf.

[oR11] Special C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[Pat11] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[Pika] Pikeos safe real-time scheduling. adaptive time-partitioning scheduler for en 50128 certified multi-core platforms. SYSGO Whitepaper.

[Pikb] Sysgo gmbh. http://www.sysgo.com.

[Pto14] Claudius Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014.

[RLSS10] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, June 2010.

[RTG] Real-time systems gmbh. http://www.real-time-systems.com.

[Rus81] John M Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[Rus00] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000.

[Sca] Scade. http://www.esterel-technologies.com/products/scade-suite/.

[Sci] Scicos. http://www.scicos.org/.

[SFu] Simulink engine interaction with c s-functions. https://www.mathworks.com/help/simulink/sfg/how-the-simulink-engine-interacts-with-c-s-functions.html.

[Sim] Simulink. http://www.mathworks.com/products/simulink/.

[sys] Sysml. http://www.omgsysml.org/.

[TCA13] S. Trujillo, A. Crespo, and A. Alonso. Multipartes: Multicore virtualization for mixed-criticality systems. In *2013 Euromicro Conference on Digital System Design*, pages 260–265, Sept 2013.

[VS12] Sarad Venugopalan and Oliver Sinnen. *Optimal Linear Programming Solutions for Multiprocessor Scheduling with Communication Delays*, pages 129–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[VxW] Wind river hypervisor. http://www.windriver.com/products/hypervisor.

[WHG99] Matthew M Wilding, David S Hardin, and David A Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In *Dependable Computing for Critical Applications 7, 1999*, pages 287–300. IEEE, 1999.

[xen] Xen zynq hypervisor. http://www.wiki.xilinx.com/XEN+Hypervisor.

[XJ12] Marc Fumey Xavier Jean, Marc Gatti Guy Berthon. The use of multicore processors in airborne systems (mulcors), 2012. https://www.easa.europa.eu/document-library/research-projects/easa20116.

[zed] Xilinx zedboard all programmable system on chip. http://zedboard.org/product/zedboard.

[ZGDY14] Tianyu Zhang, Nan Guan, Qingxu Deng, and Wang Yi. On the analysis of edf-vd scheduled mixed-criticality real-time systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, 2014.