A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation

D4.1 Run-time Architecture

Version 1.0

Contract Number	871669
Project Website	www.ampere-euproject.eu
Contractual Deadline	30.06.2020 (postponed to 31.07.2020 due to the Covid-19 emergency)
Dissemination Level	PU
Nature	R
Author	Tommaso Cucinotta (SSSA), Alessandro Biondi (SSSA), Marco Pagani (SSSA)
Contributors	Alexander de Morais Amory (SSSA), Enkhtuvshin Janchivnyambuu (SYS), Björn Forsberg (ETHZ), Arne Hamann (BOS), Marco Merlini (THALIT), Viola Sorrentino (THALIT)
Reviewer	Claudio Scordino (EVI)
Keywords	Requirements, Software architecture, Run-time execution environment.

Documentation Information

Change Log

Version	Description Change
V0.1	Gathering of raw partners' contributions
V0.2	First draft after rearrangement and adaptation of contributions
V0.3	Integration of comments after review
V0.4	Integration of partners' refined contributions after review
V1.0	Document ready for submission

Table of Contents

Executive Summary

This document provides a preliminary high-level architecture description for the run-time environment to be realized in AMPERE, to support execution of the foreseen use-cases. To this purpose, the document starts summarizing the most relevant requirements from a high-level and application behavior perspective, which are subsequently mapped onto required features for the low-level environment that will support execution of the applications at run-time. This specification is analyzed to provide a first, preliminary, high-level architecture for the AMPERE run-time execution environment that will support applications throughout their deployment and execution.

1 Introduction

The AMPERE project will develop a full ecosystem supporting and easing the development of future high-performance and real-time embedded applications on heterogeneous architectures including multi-core, GPU and FPGA acceleration. AMPERE will develop: a set of off-line tools supporting a model-driven engineering (MDE) approach for the specification of non-functional requirements (NFRs), the definition and refinement of the architecture elements fulfilling them, and the implementation phase; and, an on-line software framework supporting the execution of the applications at run-time. The overall AMPERE vision and research plan have been successfully published [QRS20] and presented at the 23rd IEEE International Symposium on Real-Time Distributed Computing, held on-line due to the Covid emergency.

In the AMPERE workplan, WP4 focuses on the run-time software architecture that is needed to support the execution of parallel applications with possible GPU or FPGA acceleration. The architecture must fulfill the real-time constraints specified at the MDE level and ensure an energy-efficient execution on top of the available underlying heterogeneous hardware elements.

This document, D4.1, covers the first 6 months of activities carried out in the Task 4.1 about “Run-time requirement specification” of WP4. It includes the initial specification of the envisioned AMPERE run-time software architecture, focusing on how non-functional requirements (i.e., timing and performance constraints of applications, alongside with energy constraints for the overall platform) map onto features of the software architecture that are needed at run-time to support execution of application components.

To this end, we start from a brief summary of the relevant requirements from the AMPERE use-cases in Section 4, as gathered through interactions with Tasks 1.1, 2.1 and 3.1, from WP1, WP2 and WP3, respectively. Then, we summarize in Section 10 the formal languages that are planned to be used to capture these requirements, and proceed detailing the envisioned architecture elements in Section 15.

2 Requirements summary

This section summarizes key high-level requirements on the AMPERE eco-system, as arising from interactions with Task T1.1 about “System model requirement specification and use case definition” from WP1, Task 2.1 about “Model transformation requirements specification” from WP2 and Task 3.1 about “Multi-criteria optimisation requirements specification” from WP3. Only the most relevant requirements with a direct impact on the run-time architectural elements are summarized here, whereas additional details are available into AMPERE deliverables D1.1 [AD11], D2.1 [AD21] and D3.1 [AD31].

2.1 Requirements from the automotive use-case

Automotive electrical and electronic (E/E) architectures are currently undergoing a radical shift in the way they are designed, implemented and deployed. Especially, the computational power and communication bandwidth required for new functionalities, such as automated driving or connected vehicle functions

(e.g. path planning, object recognition, predictive cruise control), exceed the capabilities of compute nodes traditionally adopted in electronic control units (ECUs), mainly micro-controller SoCs; this is leading to a reorganization of automotive systems following the paradigm of so-called centralized E/E architectures that are based on a new class of computing nodes featuring more powerful micro-processors and accelerators such as graphic processing units (GPUs).

One consequence of these centralized E/E architectures is that heterogeneous applications will be co-existing on the same HW platform, heterogeneous not only in their model of computation (ranging from classical periodic control over event-based planning to stream-based perception applications) but also in their criticality, in terms of real-time and safety requirements. In comparison to the previous practice to integrate mono-functional ECUs on the network level, the burden of integration is shifted from the network to the ECU level and in this regard typically from the vehicle manufacturer to the supplier of the control unit.

As a result of these trends, established methods for performance modeling and analysis that are currently used in industry are no longer effective or adequate. The following requirements for the AMPERE ecosystem can be derived:

1. Model extensions for describing parallelism of software functions
    1.1. Support for automatic code generation including parallelism from the description
2. Model extensions for describing offloading of workloads to accelerators
    2.1. Support for automatic code generation including offloading of computations to hardware accelerators
3. Model extensions for describing publish-subscribe communication paradigms as typical in modern middleware systems
    3.1. Support for automatic code generation where communications among components are integrated with the target middleware interface
4. Description of end-to-end real-time requirements along cause-effect chains spanning general purpose microprocessors and special purpose accelerators (e.g., GPUs and FPGAs)
5. The runtime should contain mechanisms to monitor and regulate execution and memory bandwidth budgets to reduce contention effects
6. The software ecosystem should include development tools and run-time mechanisms enabling an energy-aware design, so to guarantee an energy-efficient execution of the applications on the target board
7. The software ecosystem should employ a hierarchical approach allowing for incremental and localizable changes, in contrast to a “global optimization” approach where local changes might potentially lead to changing everything on the global level
8. The software ecosystem should generate SW with “deterministic” input-output behavior
    8.1. Repeatability / reproducibility (for testing and for replay of recorded behavior in lab environment)
    8.2. Functional safety: deterministic execution is also needed for SW-lockstep on µP-based platforms which do not have lockstep cores
9. The runtime needs to be developed according to a safety standard (ISO26262)

2.2 Requirements from the railway use-case

Tramway systems are facing the challenge of autonomous urban transportation systems in a similar way as automotive industry is facing the autonomous car challenge. Interestingly, these two challenges converge in cities, as most of the time tram vehicles operate in a mixed traffic environment with cars.

Autonomy applied to urban transport imposes severe real-time, reliability, and cyber-security

requirements to guarantee a safety operation of the system. Moreover, it requires to make the trams smarter through digitalization, sensors data-fusion and artificial intelligence algorithms. As a result, the use of new complex and high-demanding parallel computing platforms are requested to be installed in tram vehicles.

In this context, the AMPERE project aims at supporting a use-case focused on an Obstacle Detection and Avoidance System (ODAS) system, with the following capabilities:

- Detect the obstacles in front of the tram;
- Avoid the collision with detected obstacles;
- Warn the driver of a potential collision.

The ODAS system is intended to detect and prevent the collision against obstacles along the line, taking advantage of the fusion between the selected radar, LiDAR and optical cameras. These sensors will be installed under the front mask of the train. Sensors output data are processed and merged together in the NVIDIA Jetson Xavier Developer Kit located in the HW prototype. It has a 512-core Volta GPU with Tensor Cores and an 8-core ARM 64-bit CPU. Supported by NVIDIA JetPack, DeepStream SDKs, CUDA and TensorRT software libraries, the kit provides the useful tools to AI-based video and image understanding, as well as multi-sensor processing. At the moment, the target operating system (OS) is a Linux Ubuntu 18.04. The main blocks of the ODAS architecture for data processing are illustrated in Figure 1.

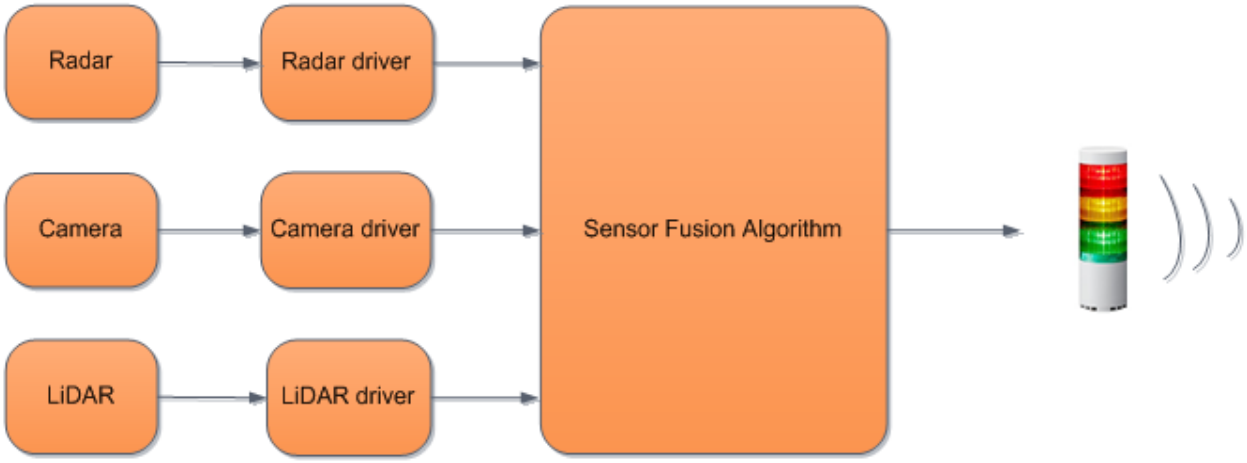

Figure 1: ODAS system architecture

The information output by the ODAS system will include:

1. Time stamp;
2. Obstacle ID;
3. Obstacle class: tram, car, bus, pedestrian, motorbike, bike;
4. Obstacle position;
5. Obstacle dimensions;
6. Spares.

If a potential collision will be detected, according to the outputs of the sensor fusion algorithm (SFA), the driver will be warned with a visual and sound alert.

The ODAS system, which will be installed on the train, is represented in Figure 2.

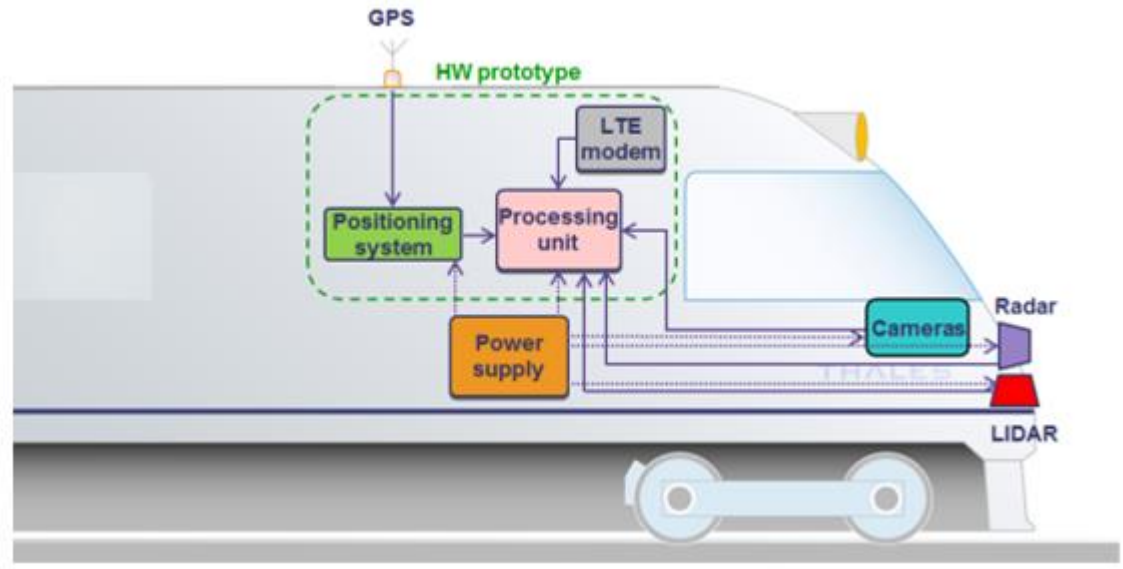

Figure 2: ODAS system on the train

Sensors have been carefully selected according to the performance and to the compliance to standards, in order to reduce the inaccuracies and the sources of error. The devices certifications are listed in Table 1.

Table 1: Certification of the selected ODAS sensors

Sensor	Certification
Camera	ECE R10 rev.04, EN 50121-4, IEC/EN/UL 60950-1, IEC 60068-2-1, IEC 60068-2-2, IEC 60068-2-14, IEC 60068-2-30, IEC 60068-2-78, IEC 60529 IP4X, IEC 62236-4, EN 55022 Class B, EN 55024, EN 61000-3-2, EN 61000-3-3, EN 61000-6-1, EN 61000-6-2, FCC Part 15 Subpart B Class B, ICES-003 Class B, VCCI Class B, C-tick AS/NZS CISPR 22 Class B, KCC KN22 Class B, KN24, EN 50581, IEC 60068-2-60, IEC 60529 IP66/IP67, IEC 60721-3-5 5M3 (vibration, shock), IEC/EN/UL 60950-22, NEMA 250 Type 4X
LiDAR	FDA, FCC, CE, RoHS, WEEE, ASTM G154, ETSI EN 300 019-2-5, IEC Class 5M3 , IEC-60079-15
Radar	EN 62311:2008, EN 60950-1:2006 + A11:2009 + A1:2010 + A12:2011 + AC:2011 + A2:2013, EN 301 489-1 V2.2.0, EN 301 489-51 V1.1.1, EN 301 091-1 V2.1.1, EN 303 396 V1.1.1, FCC and IC for USA and Canada compliant, NCC compliant for Taiwan

2.2.1 Requirements of the ODAS system

In this section, the non-functional requirements of the ODAS system are presented. The specifications of the railway use case are presented according to the operating environment.

HW requirements of the ODAS system

The ODAS system is characterized by a number of hardware, physical and electrical requirements, related to how the various sensors need to be mounted and oriented, how possible electromagnetic interferences should be considered, how to minimize the impact of vibrations during the drive, etc. These do not impact on the AMPERE software run-time architecture, so they are omitted for brevity.

The hardware requirements that may be relevant for the run-time architecture concern the maximum level of power consumption and the selected models of sensors, as detailed below.

PUID: **[SYS-ODAS-REQ-001]**

The OS shall support the following sensors for the ODAS:

- Axis F1005-E camera sensor unit + Axis F41 camera main unit;
- Quanergy M8 Plus LiDAR;
- Continental ARS408-21 Premium radar.

A viable OS, connecting to the mentioned sensors as detailed in **SYS-ODAS-REQ-002** below, seems Linux.

PUID: **[SYS-ODAS-REQ-002]**

ODAS sensors interfaces with the NVIDIA AGX Xavier board shall be:

Table 2 ODAS sensors interfaces

Sensor	Interface with AGX Xavier
Camera	Ethernet
LiDAR	Ethernet
Radar	USB

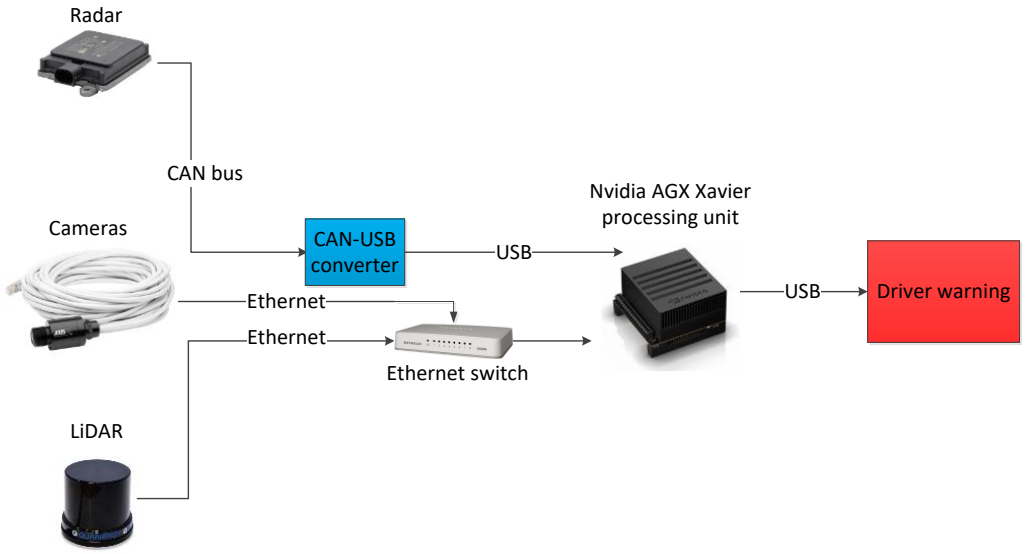

Figure 3 ODAS sensors interfaces

PUID: **[SYS-ODAS-REQ-003]**

The ODAS power consumption shall be lower than 200 W.

This constraint makes the ODAS system power consumption quite negligible compared to the 118 kW/h

consumed by each of the four motors of the Sirio train of the Florence tramway line.

SW requirements of the ODAS system

PUID: **[SYS-ODAS-REQ-004]**

The run-time (OS+middleware) for the ODAS shall include the following libraries:

- ROS;
- OpenCV;
- GStreamer;
- Qt;
- Python3;
- PyTorch;
- CUDA;
- DeepStream;
- Spdlog.

PUID: **[SYS-ODAS-REQ-005]**

The software ecosystem shall guarantee the support of fault tolerant architectures like 2oo2 (2-out-of-2) or 2003.

PUID: **[SYS-ODAS-REQ-006]**

The run-time shall allow the contemporary execution of safe and non-safe (SIL 0) applications over the same OS/safety layer software architecture.

PUID: **[SYS-ODAS-REQ-007]**

The run-time shall guarantee application replica determinism through synchronization mechanisms (SW-lockstep or similar) and message/data exchange methods.

PUID: **[SYS-ODAS-REQ-008]**

The run-time shall support POSIX queue mechanisms for voted and non-voted message passing paradigms.

PUID: **[SYS-ODAS-REQ-009]**

The run-time shall include middleware implementing safe voting across output produced, or messages sent, by application replicas from their source tasks, before applying the outputs or sending the messages to the receiving task(s).

PUID: **[SYS-ODAS-REQ-010]**

The run-time shall support proper shutdown of replicated applications in case of failure.

PUID: **[SYS-ODAS-REQ-011]**

The run-time shall support proper task application recovery after failure fix.

Timing requirements of the ODAS system

PUID: **[SYS-ODAS-REQ-012]**

Radar output data shall be sent every 72 ms, which is the sensor cycle time.

PUID: **[SYS-ODAS-REQ-013]**

Camera frame rate shall be 25/30 fps (50/60 Hz), recording a 1080p video with the WDR mode.

PUID: **[SYS-ODAS-REQ-014]**

The trained algorithms of the convolutional neural network (CNN) shall guarantee a pixel density of at least 25 pix/m, which is the requirement on the resolution of the camera model to detect objects.

PUID: **[SYS-ODAS-REQ-015]**

A packet of 50 LiDAR firing data divided between 8 layers shall be received and processed every $4.81 \cdot 10^{-4}$ Hz, considering a rotational speed in cycles per second of 10 Hz for the inner engine.

PUID: **[SYS-ODAS-REQ-016]**

The ODAS system shall detect obstacles along the tramway track in order to warn the driver about potential collisions in less than 2.5 s, which is the typical driver reaction time.

3 Modeling language

Section 4 summarized key non-functional requirements of the AMPERE use-cases that are typical of modern cyber-physical systems. In these systems, high-performance computing (HPC) requirements impose the use of parallelizable software components that are deployed on heterogeneous hardware with GPU and FPGA acceleration, while at the same time respecting requirements coming from the traditional world of embedded safety-critical control systems (like deterministic execution, precise control of the interference among software components co-located on the same hardware, and reliability).

In the MDE approach envisioned throughout the AMPERE project, these requirements have to be properly captured through the use of domain-specific modeling languages (DSMLs), and carried along throughout the whole software life-cycle. This spans across requirements specification to architecture and components definition, up to the implementation stage, possibly resorting to the help of automatic techniques for model transformation and code-generation from the models as typical in MDE approaches. In the final testing/verification stages, the actual performance of the system has to be verified against the NFR specifications.

As to the choice of the modeling language for requirements specifications, formalisms that are particularly relevant for the AMPERE use-case domains, that the AMPERE partners are actively using or particularly interested into, include:

1. AUTOSAR
2. AMALTHEA
3. CAPELLA
4. SIMULINK

In the following, we provide a brief description of these languages, with a particular emphasis on their capability to capture real-time requirements for complex, distributed and parallel applications. Here, only a brief summary of the main characteristics of these languages is included, while a detailed technical analysis is performed in the AMPERE Deliverable D2.1 [AD21].

3.1 AUTOSAR

AUTOSAR[1], a result from the former ITEA project EAST-EEA[2], defines a methodology for component-based development of automotive software and a standardized software architecture for automotive electronic control units (ECUs). Today, AUTOSAR is an alliance of key industrial players in the automotive industry,

[1] More information at: https://www.autosar.org/.

[2] More information at: https://itea3.org/project/east-eea.html.

establishing an industry standard for automotive software architecture. The standard defines mainly a Classic Platform for embedded systems with hard real-time and safety-critical constraints, and an Adaptive Platform for fail-safe high-performance computing ECUs for use cases such as autonomous driving. AUTOSAR defines a component-based software framework including a component framework (SW-C) for the definition of software components, their composition and interactions, and the Virtual Functional Bus (VFB) abstracting communications among components, within the same or different ECUs. The fundamental building block of an AUTOSAR model is the Runnable, describing an atomic block of computations which is described in terms of: its required inputs and provided outputs, constituting the provided interface of the Runnable; the required interface, detailing what dependencies the component has on other components or the environment; the resource requirements and additional NFR properties; the structure and composition of Runnables; constraints on the behavior, in terms of pre/post-conditions that have to be respected before/after each invocation of a Runnable, and possible invariants.

The AUTOSAR methodology provides support for the specification of the timing behavior of applications. Indeed, the AUTOSAR Timing Extensions (TIMEX) [ACTA19] are used in the AUTOSAR Classic Platform to associate timing requirements and specifications (e.g., worst-case response time of an end-to-end service chain) to functional blocks of the specification during the early-stage specification phases, as well as to the intermediate elements of the function networks obtained during the functional decomposition, and the final mapping to the underlying physical topology elements and Runnables. For example, TIMEX allows for the specification of worst-case execution times and worst-case transmission times for communications, so that, for each possible mapping to the physical topology, an end-to-end timing analysis of the system can be performed using appropriate tools, to verify that the being-implemented system will satisfy all the specified timing requirements.

However, regarding the possibility to run components on parallel and multi-core platforms, AUTOSAR lacks a detailed model enabling advanced simulation and verification techniques, which are becoming increasingly important for the complexity of modern automotive systems.

The latest AUTOSAR Adaptive Platform (AP) standard, version R19-11 at the moment, takes into consideration the possibility of deploying Runnables onto modern platforms including multi-core, GP-GPU and DSP acceleration in the execution. A specific document [AAPP19] of the AP standard series provides design guidelines for using parallel processing in the underlying hardware, including task, data and pipeline level parallelism, and instruction-level parallelism (typical of VLIW or SIMD instruction-set extensions in processors), through the use of a number of parallelization frameworks including OpenMP, OpenCL/CUDA, OpenCV/OpenVX and others, as well as OS-level parallelism for multi-cores in terms of POSIX threads. Following service-oriented architecture (SOA) principles, the guidelines identify elements at the application layer, where possibly accelerated components from the service layer can be accessed through their service interface. The components of the service layer can use parallel processing libraries, frameworks and OS services for their implementation, employing two possible models: the accelerator model is typical of GP-GPU accelerators (and probably also FPGA ones) invoked synchronously by the software running on the CPU; the CPU/co-processor model is typical of multi-core architectures, where a number of additional CPUs/cores are used to accelerate the execution through either an OpenMP run-time, or explicit handling of OS threads, e.g., via POSIX APIs for multi-threaded software components.

Also, the AUTOSAR Specification of Execution Management document [AASEM19] defines a number of attributes of the *DeterministicClientResource* class, related to the timing of the code to be executed. Not only the *numberOfInstructions* attribute is defined, corresponding to the WCET sequential execution, but also a parallel execution model is defined in terms of a single parallelizable segment of the execution starting after an initial sequential part of execution as long as specified by the *sequentialInstructionBegin* attribute, and terminating before the final sequential part of execution as long as specified by the *sequentialInstructionEnd* attribute. The parallelizable segment undergoes an acceleration as specified by the *speedup* attribute (float, expected to be greater than 1.0) when the segment is run in parallel using a number of threads as specified in the *numberOfWorkers* attribute of the *DeterministicClient* class, from

the “deterministic worker pool”. The specification refers to a worker pool dedicated to the single Runnable being specified, with physical platform cores dedicated to the worker pool. In said conditions, the expected WCET of the client is reshaped as:

$$WCET_{par} = SIB + SIE + \frac{NOI - (SIB + SIE)}{speedup}$$

where SIB, SIE and NOI are the *sequentialInstructionBegin*, *sequentialInstructionEnd* and the *numberOfInstructions* attributes, respectively.

When considering the need for supporting high-performance heterogeneous embedded computing platforms as arising in AMPERE, it is clear that the above model refers essentially to acceleration on symmetric multi-core platforms with identical CPUs, and it is incapable of handling multiple types of underlying CPUs on the same system, as well as GPU or FPGA acceleration. At the same time, the documents mentioned above about the exploitation of hardware accelerators seem very limited in scope, providing merely generic guidelines and principles which cannot be directly leveraged without more precise and technical elements adding language constructs to the models.

AUTOSAR provides a reference implementation called Adaptive Platform Demonstrator (APD) that is available only to partners of the consortium.

3.2 AMALTHEA

AMALTHEA[3] is an open-source modeling and development platform [AWP13], targeting development of efficient automotive applications for multi-core embedded boards, designed with AUTOSAR compatibility in mind. The vision embraced by AMALTHEA [AECE13, ADF11] includes the possibility to provide a formal and rich description of the software components and their interactions and communications, so that it is possible to perform a detailed and accurate simulation of the timing of the system execution, in order to verify that the overall system meets the specified design-level constraints, e.g., in terms of achievable end-to-end response times and the capability to respect deadlines. The AMALTHEA methodology is made available as the APP4MC[4] Eclipse-based application for the Eclipse modeling and development platform, which is able to perform both theoretical timing analysis of the modeled elements, as well as running simulations of their behavior thanks to its automated code-generation capabilities.

When dealing with timing analysis and parallelism in the hardware, works can be found in the research literature addressing real-time schedulability analysis of AMALTHEA Runnables once they have been mapped onto periodic real-time tasks running on multi-processor platforms. For example, a variant of the well-known response-time analysis (RTA) is proposed in [SBB16] for the case of mixed preemptive/cooperative tasks on partitioned multi-core systems where delays due to contention on memory accesses can be neglected.

It is noteworthy to mention the on-going efforts in the context of the PANORAMA EU project[5], where partners BOSCH and ISEP are also involved, which focuses on supporting heterogeneous architectures in automotive and aerospace environments, integrating the developed techniques within the APP4MC Eclipse framework. In [KBGW19], the problem is considered of extending RTA techniques for heterogeneous systems comprising chains of tasks that can either run on the CPU as Runnables enclosed in OS threads under preemptive fixed-priority scheduling, or on a GP-GPU, where delays due to memory contention and transfers between the main memory and the GP-GPU are also considered.

[3]See also the official AMALTHEA website: www.amalthea-project.org/.
[4]More information at: https://www.eclipse.org/app4mc/.
[5]More information at: https://www.panorama-research.org/vision/.

The capabilities of the AMALTHEA modeling approach, the openness of the standards and wide availability through the open-source APP4MC Eclipse project, its compatibility with the well-known AUTOSAR standards, and active adoption by partners in the aerospace and automotive fields, including the AMPERE partners BOS and ISEP, make this tool particularly promising for adoption within the AMPERE use-cases.

3.3 CAPELLA/ARCADIA

Eclipse Capella[6] is an extensible open-source model-based systems engineering (MBSE) tool based on Eclipse, allowing for the use of the ARCADIA [ROQ16, OGJH16] modeling methodology developed by Thales in 2005-2010 to design systems using a model-based engineering approach. The adopted modeling language is inspired to UML/SysML, with elements suitable for a number of modeling phases, from requirements (a.k.a., "needs") analysis and modeling, to architecture building and validation to verify the satisfiability of the needs, to engineering of the requirements, projecting the needs onto required capabilities and constraints for individual components of the architecture.

The CAPELLA/ARCADIA DSL supports and enforces an approach to system modeling based on successive engineering phases, from the collection and analysis of both customer needs (similar to UML and SysML use-case diagrams) and system/SW/HW needs (including functional and non functional analysis of required system components), to the final architecture design and development phases; this includes both logical and physical architecture design, as well as the possibility to clearly define development contracts, all in accordance with the IEEE 1220 standard [IEEE1220].

CAPELLA/ARCADIA has some limited support for non-functional requirements, in that maximum response times of the overall system actions, as well as individual components, can be specified at various levels throughout the various phases, considering also the natural end-to-end interactions that happen in complex systems comprising of function chains.

CAPELLA/ARCADIA tools guide engineers through the needed allocation of resources able to guarantee non-functional requirements such as the maximum end-to-end latency for function chains or given reliability levels. For example, it is possible to keep in check the allocation of computing capacity and communication bandwidth for the individual components so as to avoid over-allocation and prevent uncontrolled growth of response times. Moreover, the tools included in CAPELLA/ARCADIA make use of a subset of the MARTE OMG standard [MAR08] as a basis to represent a synthetic view of the system design model that captures all elements impacting the system timing behavior and is required to perform timing verifications. The MARTE profile defines precise semantics for time and resource modeling, that allow for automatic transformation of models to lower abstraction level models that can be used for simulation or implementation purposes. With the MARTE modeling tools, engineers can precisely define the timing requirements of each Runnable, shared resource, the interactions among components, and even scheduling policies and analytic models that can be applied to check the timing correctness (e.g., schedulability) at design time.

3.4 SIMULINK

MATLAB Simulink[7] by MathWorks is among the most used commercial modeling, analysis and development tools for model-based design (MBD) of control units in embedded systems. Simulink allows for the specification of system components in formal terms, using executable models that enable

[6] More information at: https://www.eclipse.org/capella/.

[7] More information is available at: https://it.mathworks.com/products/simulink.html.

continuous verification and validation of the design using simulation and formal methods. This way, possible errors can be detected at the early stages in the design and development process. A Simulink model can be translated into production code through automatic code generators such as the dSPACE TargetLink[8] or MathWorks Embedded Coder[9], which include, among their supported target boards, also some ones with programmable hardware and multi-core capabilities.

Automatic code generation from models is a key capability of an end-to-end MBD/MDE approach: it helps reducing the gap between the required and implemented features, enabling automated consistency checks and tracing of requirements from the models to the implemented final components; it allows for faster and more cost-effective development, reducing the time-to-market; it eases certification processes; and, generated code can have similar size- and time-efficiency, when compared to hand-written one [BRA19].

However, the jump from simulated models to generated software that runs on a real board involves crossing a non-trivial semantic gap between the ideal execution of the models, and the actual execution of the generated code within a real software architecture. Individual execution components at the modeling level are usually mapped to OS tasks that execute cyclically with fixed periods, and these are deployed with given scheduling parameters controlling how the OS scheduler resolves contention among multiple tasks trying to run at the same time on the CPU (this is not specific to Simulink models, but it is similarly done in AUTOSAR and AMALTHEA that also map statically sequences of Runnables into cyclic OS tasks). One key aspect of this semantic gap is the synchronization of the communications among the various model components. For example, a modeling element that activates periodically and can easily be simulated with a precise periodic activation, in the real implementation might see jitter in its activation times due to interferences from other tasks as scheduled by the OS on the underlying CPU(s).

Simulink includes a number of mechanisms that allow for keeping a precise semantics of auto-generated models, with respect to the ideal modeled behavior, however the tool is still lagging behind when it comes to supporting multi-core architectures. Beyond introducing an unavoidable further mapping between OS tasks and available hardware cores for a multi-core architecture, resorting to a statically partitioned scheduler as commonly available on a variety of OSes, it becomes cumbersome to ensure that the original offsets and synchronization among elements deployed onto different cores are what one expects, and to control the temporal interference due to multiple cores accessing the same hardware elements like shared caches, buses and memory controllers.

It is noteworthy to mention some research activities carried out by AMPERE partner SSSA in this area. For example, in [BRA19], a new code generator for Simulink models has been proposed, aiming to preserve the order of execution of blocks and data flows as defined for a purely functional model (that is, without any platform mapping). The generator has been made more easily re-target-able to a wide selection of platforms thanks to the introduction of an abstract OS API [BDN18] that allows for running the semantics-preserving generated models over different OSes (i.e., both Linux and PikeOS are supported). Furthermore, a similar line of research at SSSA [PBDN19] aimed at introducing into the abstract models the expectable interference and non-predictability due to the real deployment of generated models on multi-core platforms, proposing schedulability analysis techniques for partitioned multi-core systems relying on the Logical Execution Time (LET) paradigm [KS12]. According to LET, a task is provisioned with all of its needed inputs exactly at the beginning of its activation period, and it provides its computed outputs exactly at the time of its absolute deadline, i.e., the end of the activation period for classical periodic tasks with implicit deadlines. This way, the semantics of the computation is made not to depend on the actual schedule and inter-leaving of the tasks (as long as all of them respect their deadlines). With such a paradigm, not only tasks are partitioned and scheduled on CPUs, but also their accesses to memory

[8] More information is available at: https://www.dspace.com/en/pub/home/products/systems/targetimp.cfm.

[9] More information is available at: https://it.mathworks.com/products/embedded-coder.html.

can be exactly scheduled, so as to account precisely for, or prevent, the expectable contention due to tasks executing on multiple cores. However, the model fits precisely only on platforms with special hardware features like local per-core (instruction and data) scratchpads, while general-purpose multi-core platforms can somewhat approximate the ideal LET paradigm under certain constraints on the applications working-set sizes, and when employing cache partitioning techniques among cores/tasks (e.g., cache coloring).

4 Run-time architecture

This section contains a first sketch of the software architecture that is envisioned in AMPERE as being appropriate and suitable to cope with the requirements and specifications detailed in Section 4 and to be formalized through the MDE techniques detailed in Section 10. This section focuses particularly on predictable execution and real-time constraints in presence of parallelizable software and parallel hardware, time- and energy- efficient execution and the optimum exploitation of underlying heterogeneous hardware accelerators like GPUs and FPGAs.

In what follows below, a general overview of the run-time architecture is provided first (Section 15). Then, specific components of the architecture are discussed in greater detail, highlighting important features needed in the hypervisor (Section 19) and operating system (Section 24), including the handling of FPGA accelerators (Section 27), and finally those needed in the energy management components (Section 29).

Note that architectural details about the run-time support to parallel programming models and frameworks like OpenMP or CUDA are not provided here, as these are discussed in detail in the AMPERE Deliverable D2.1 [AD21].

4.1 High-level architecture overview

The high predictability and precise execution timing requirements of the AMPERE use-cases need to be reflected into an appropriate run-time execution environment capable of managing the underlying hardware while guaranteeing precise real-time guarantees to applications. This is generally possible using a *hard real-time operating system* (RTOS).

Moreover, we envision in the AMPERE eco-system the need for hosting software components with a significant diversity in terms of timing requirements, safety criticality, and processing demand. For example, these comprise: traditional control functions and high-performance control logic that make use of massive vector/matrix operations that need to be accelerated through GP-GPUs, e.g., as needed in optimum control or sophisticated computer vision algorithms based on machine learning and artificial intelligence; or, computations that need high levels of predictability or energy efficiency throughout their execution, that can only be achieved with FPGAs. This results in the need for a software stack that is capable of letting the co-existence of components with different criticality levels (“mixed criticality”) on the same physical platform. This is possible, for example, by adding a *hard real-time hypervisor*, guaranteeing the highest standards in isolation, including temporal and functional isolation among different guest operating systems, as well as hardware partitioning across different guests if needed.

In terms of guest operating systems, in the envisioned architecture we can find either hard real-time operating systems or, as the tightness of the timing requirements and/or the criticality of the requirements “smooth out”, also a complex operating system like Linux.

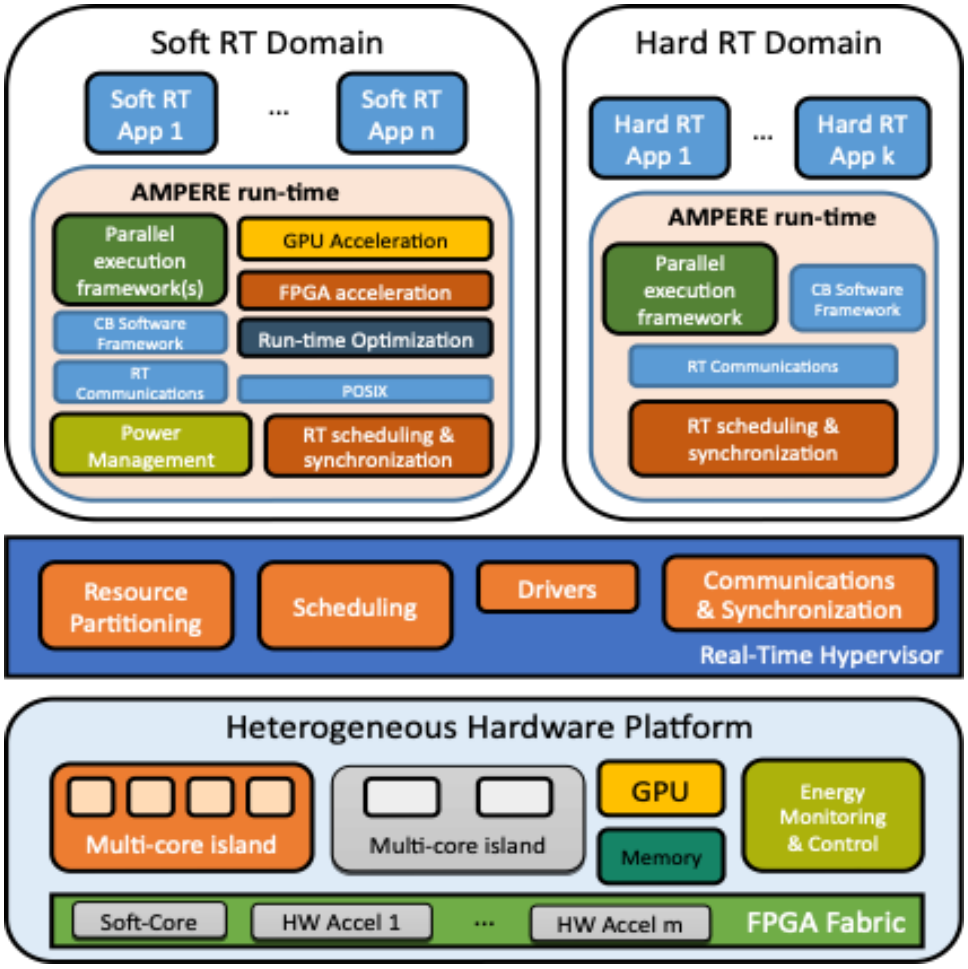

Figure 4: Abstract view of the AMPERE run-time architecture.

From the above considerations, we can sketch out a preliminary architecture for our envisioned run-time environment, which is capable of satisfying the requirements identified so far, as exemplified in Figure 4. Going from the bottom towards the top, we can see in the figure an abstract representation of our typical target board, which includes a number of heterogeneous elements.

One or more multi-core islands provide general-purpose processing capabilities. Cores within an island are symmetric and have a compatible instruction-set architecture (ISA), so that the OS can be free to migrate tasks across them. Different islands might also be ISA-compatible, as it happens with the ARM big.LITTLE architecture[10], massively deployed on nowadays' smartphones and other mobile devices. In this architecture, an island is specialized in high-performance computations through complex "big" cores with high frequency and a sophisticated pipeline, while another island is specialized in energy-efficient computations with "little" cores characterized by a simpler internal architecture. On a big.LITTLE system, the OS can freely migrate tasks across cores and islands, but it should be aware of the differences between the two islands. The dynamic voltage and frequency scaling (DVFS) capabilities are limited to a single clock per island, adding complexity to the OS scheduling and power management logics. The recent ARM DynamIQ architecture[11] extend big.LITTLE with the possibility of having a third island, additional clocks, and an additional cache layer.

[10] More information available at: https://www.arm.com/why-arm/technologies/big-little.

[11] More information available at: https://www.arm.com/why-arm/technologies/dynamiq.

Alternatively, different islands may specialize on different capabilities by simply sharing the same memory, while having incompatible ISAs. For example, on the Xilinx UltraScale+ board, the Cortex-A core island can be used for general-purpose processing, while the Cortex-R island specializes in real-time and safety-critical computing, with enhanced low-latency and predictability features that result in a faster reaction to hardware interrupts and more deterministic access to local RAM. Additionally, Cortex-R cores can be run in dual redundant lockstep mode for high-reliability processing.

Complex algebraic computations as needed in modern embedded applications performing optimized control, computer vision, machine learning and artificial intelligence, can be accelerated, besides leveraging the multi-core capabilities of the board, also by resorting to the on-board GPU or GP-GPU, for achieving significant speed-ups on vector and matrix operations.

Additionally, the board may have a FPGA fabric for similar acceleration purposes, and/or time-critical functions that cannot be realized in software given the very tight requirements in execution, or that can be more conveniently and energy-efficiently realized in the FPGA rather than in software. Optionally, an additional CPU might be made available instantiating a soft core within the FPGA fabric, like a Xilinx Microblaze or a RISC-V processor. In AMPERE, the envisioned complexity and dynamicity of the deployed applications implies a focus on FPGA fabrics having dynamic partial reconfiguration (DPR) capabilities, i.e., where individual FPGA slots can be dynamically reconfigured while the others keep working without disrupting their provided service. We will see later in Section 27 how to optimally exploit this feature.

Finally, we can find in the hardware energy monitoring and control features allowing for a control of the energy saving capabilities of the various elements of the board from the software stack. For example, the Xilinx Zynq UltraScale+ board has advanced power management capabilities that are exposed to the software through the Xilinx Power Management Framework [XPM16].

The hardware is under direct control of a Real-Time Hypervisor capable of guaranteeing precise resource allocation and temporal isolation, as well as partitioning and strict isolation, of different higher-level run-time environments with different predictability features and criticality levels so that they can access in a controlled way some of the available underlying physical resources without being able to interfere with each other. PikeOS from AMPERE's partner SYSGO is certainly a suitable component for this element of our architecture, albeit it needs adaptations for being used on the chosen target board for the AMPERE run-time, as planned to be realized in Task 5.4. Some details about PikeOS are enclosed in Section 19.

The real-time hypervisor will allow for the co-existence of at least a Hard Real-Time OS, where safety-critical software components will be deployed, and a Soft Real-Time OS, hosting soft real-time components with relaxed timeliness or criticality requirements, and/or complex components for which it may be impossible to acquire certain certifications. As hard Real-Time Operating System (RTOS), an interesting choice is the ERIKA Enterprise OS[12] from the AMPERE partner Evidence, which is a real-time kernel certified OSEK/VDX and designed for AUTOSAR compliance. On the other hand, the Soft RT OS can be a POSIX-compliant OS like Linux, which has better real-time capabilities than other POSIX-compliant open-source OSes (see Section 24).

In the AMPERE ecosystem, applications are developed atop a component-based software development framework that eases development and deployment of application components. For example, AUTOSAR provides a runtime middleware that needs to be deployed alongside applications on the platform, to support the correct execution of the deployed Runnables. Interestingly, an AUTOSAR run-time is available both on ERIKA Enterprise, in the Classic Platform variant, and also on the Linux OS, in the Adaptive Platform variant, albeit only to members of the AUTOSAR consortium. Also, a further requirement that emerged from the use-cases, is the one to have support for real-time and reliable publish/subscribe communications, which can be provided by services like the Scalable service-Oriented MiddlwarE over IP

[12] More information about the ERIKA Enterprise RTOS is available at: http://www.erika-enterprise.com/

(SOME/IP)[13], an automotive middleware solution designed to be integrated in AUTOSAR, or the Data Distribution Service (DDS) [DDS15] standardized by the OMG, possibly through the use of the ROS2[14] middleware that is increasingly gaining interest within the real-time control community.

The focus on parallel and high-performance embedded applications of the AMPERE project implies the use of parallel programming frameworks. These include a model and API to be used while developing applications, as well as additional middleware services needed at run-time by the deployed parallel application components. One typical example is OpenMP, which is widely supported on Linux. A very preliminary support of a subset of the OpenMP standard has been designed also for ERIKA Enterprise, through the UpScale[15] SDK environment developed in the P-SOCRATES EU project. This makes OpenMP an ideal element to be integrated in the AMPERE toolchain as a foundational block for building a Parallel Execution Framework for realizing high-performance, real-time control functions capable of exploiting parallelism in the underlying multi-core platform.

In AMPERE, we aim at supporting complex embedded systems where, besides traditional hard real-time and safety-critical components that may need a certified run-time as constituted by a RTOS + real-time hypervisor ensemble, we foresee co-existence of additional real-time components which can have a much higher complexity in terms of software architecture, as well as somewhat lessened requirements in terms of timeliness guarantees and safety criticality. This may be the case of complex computer vision algorithmic or ML/AI-enhanced control logic to be accelerated on GP-GPUs, or even software components responsible for deploying and controlling functions deployed as hardware logic blocks within the FPGA fabric. These components are typically more suitable to be developed on, and run within, a Linux operating system, due to its versatility and wide availability of libraries for a multitude of computational as well as communication purposes.

We foresee a number of customizations of the Linux kernel, in order to support better the computations as needed within AMPERE, for example specific energy-aware support for the SCHED_DEADLINE real-time scheduler on non-SMP platforms, and the adoption of a PREEMPT_RT kernel (more details on these aspects will follow later in Section 24).

The Linux environment is also where we can support a plethora of parallelization frameworks for high-performance software components to be heavily accelerated on multi-core and/or GP-GPUs, like OpenMP, OmpSs, OpenCL/CUDA or others. At the moment, OpenMP seems one of the most promising tools for integrating in the AMPERE ecosystem parallel components with hardware acceleration, thanks to its `target` directive (see AMPERE Deliverable D2.1 [AD21]).

Moreover, we plan to host within the Linux OS the framework for acceleration of real-time tasks on the FPGA fabric. For this, we plan to integrate our FRED prototype (see Section 27) which is capable of real-time replacement of hardware functions on FPGA slots dynamically, according to the needs of the running real-time tasks, when dynamic reprogrammability of FPGA slots is available, as on our target Xilinx UltraScale+ board. In AMPERE, parallel programming models like OpenMP will need to be integrated with FPGA-based acceleration, besides GPU or multi-core acceleration. Some prior literature exists on the topic, as surveyed in [MKP19], but this will be subject of further research across AMPERE WP2 and WP4.

Finally, we plan to retain on the Linux OS side a run-time optimization component that will constitute part of the multi-objective optimization framework developed in WP3, that will be capable to either configure the platform according to the optimum configuration computed off-line, or to reconfigure dynamically the set-up according to dynamically changing conditions. For example, this may happen in case of dynamic start/stop of some applications or services in the soft RTOS (in a hard RTOS, everything is normally quite static and pre-configured), or changes in the underlying conditions or constraints.

[13] More information is available at: http://some-ip.com/.
[14] More information is available at: https://index.ros.org/doc/ros2/.
[15] More information is available at: http://www.erika-enterprise.com/index.php/download/upscale-sdk.html.

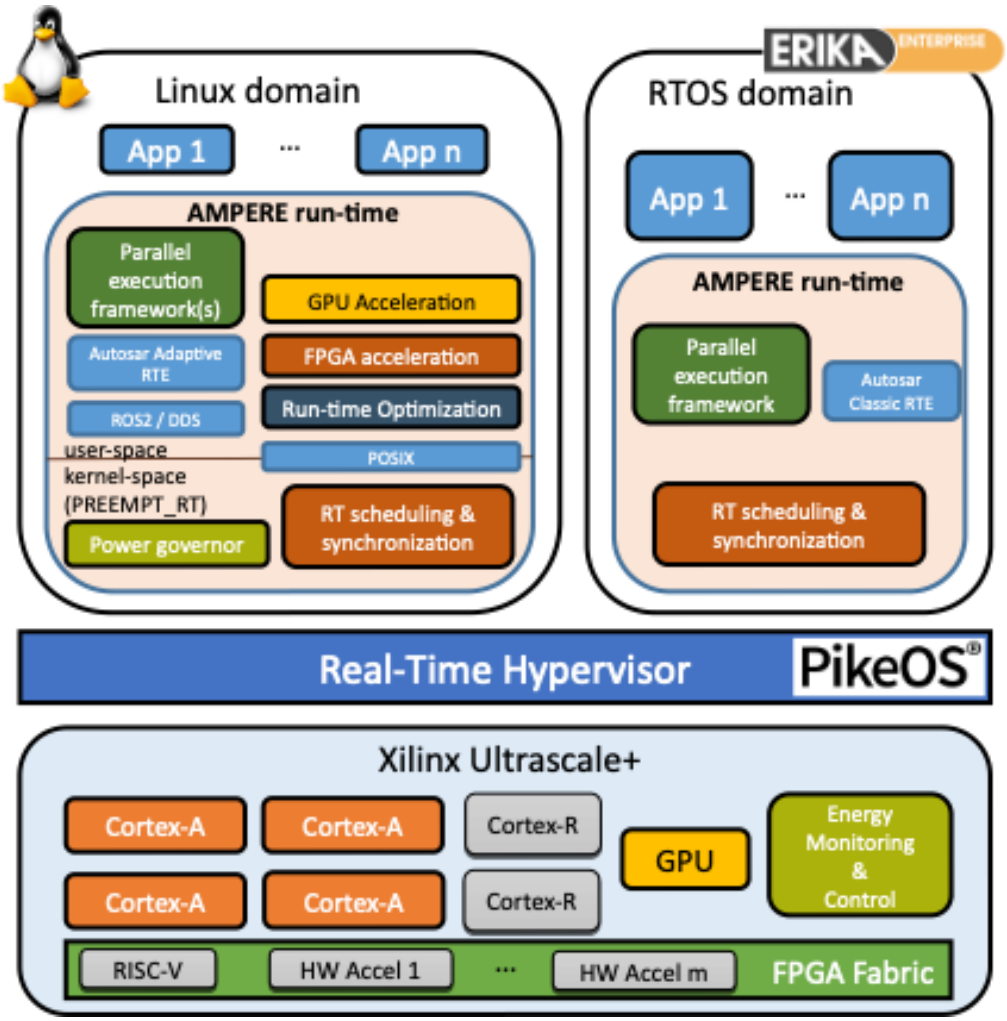

Figure 5: Possible instance of the AMPERE run-time architecture.

The real-time scheduling features of the run-time environments (RTOS and Linux), coupled with analysis techniques to be carried out off-line, and the platform optimization components, will enable energy-aware real-time performance on the platform with an optimum exploitation of the parallelism in the underlying hardware.

Considering the discussed possibilities for the components in the abstract architecture view of Figure 4, we can obtain the tentative sample architecture depicted in Figure 5, where we have chosen as hardware board the Xilinx Zynq UltraScale+ ZCU102[16], one of the two platforms selected by the AMPERE project in deliverable D5.1.

More details on the individual components envisioned in our overall AMPERE run-time architecture are provided in the next sections.

4.2 PikeOS Hypervisor

The concept of PikeOS combines a real-time operating system (RTOS), a virtualization platform and an Eclipse based integrated development environment (IDE) for embedded systems.

One of the key features of PikeOS is the capability to safely execute applications with different safety levels concurrently on the same platform. This is achieved by the strict spatial and temporal segregation

[16] More information at: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

of these applications by means of *software partitions*. A software partition can be seen as a container with pre-allocated resources (memory, CPU time, I/O access rights) which can host one or more applications sharing the partition's resources. With PikeOS, the term application refers to an executable linked against the PikeOS API library and running as a process inside a partition. Due to the nature of the PikeOS API, applications can range from simple control loops up to complete para-virtualized guest operating systems like Linux.

Software partitions are also called Virtual Machines (VMs), because it is possible to implement a complete guest operating system inside a partition which executes independently from other partitions.

PikeOS can be seen as a Type 1 Hypervisor, the Guest OS layer is called *Personality*.

Main PikeOS Features and Architecture

The main features and responsibilities of the PikeOS Hypervisor are:

- Hardware abstraction (processor and platform)
- First level exception and interrupt processing
- Address space management
- Thread management and scheduling
- Communication and synchronization
- Resource management and protection with respect to resources and processing time
- Health monitoring
- Inter-partition communication
- I/O device abstraction and access control
- File system support

For the aspect of resource management, protection and scheduling the following applies:

- PikeOS provides effective barriers through spatial and time partitioning mechanisms to ensure the independence between higher and lower safety integrity level applications. PikeOS is developed at SIL4 to ensure this independence.
- PikeOS provides means to ensure that any critical application is able to be executed in its own process with its own virtual memory space supported by hardware memory protection (MMU, IOMMU).
- PikeOS provides an IOMMU manager to protect the data memory of a critical application from wrong modification by a hardware device (i.e. DMA).
- PikeOS provides means to ensure deterministic scheduling methods: Strict priority-based scheduling implemented by a real-time engine with a means of avoiding priority inversion.
- PikeOS provides time fences through time partitioning means to allow terminating the execution of an application if it over-runs its allotted execution time or deadline and guarantee that no critical application can be starved of processor time.
- PikeOS provides means to control interrupts to ensure that any interrupt cannot corrupt, in an unsafe way, the execution of a critical application.

The features listed above are implemented by the PikeOS core (also called PikeOS Hypervisor). The PikeOS

core consists of the PikeOS Microkernel and the PikeOS System Software (PSSW). The generic part is, at source code level, independent from the CPU architecture, and, at object code level, independent from the underlying hardware platform. The PikeOS architecture is depicted in Figure 6.

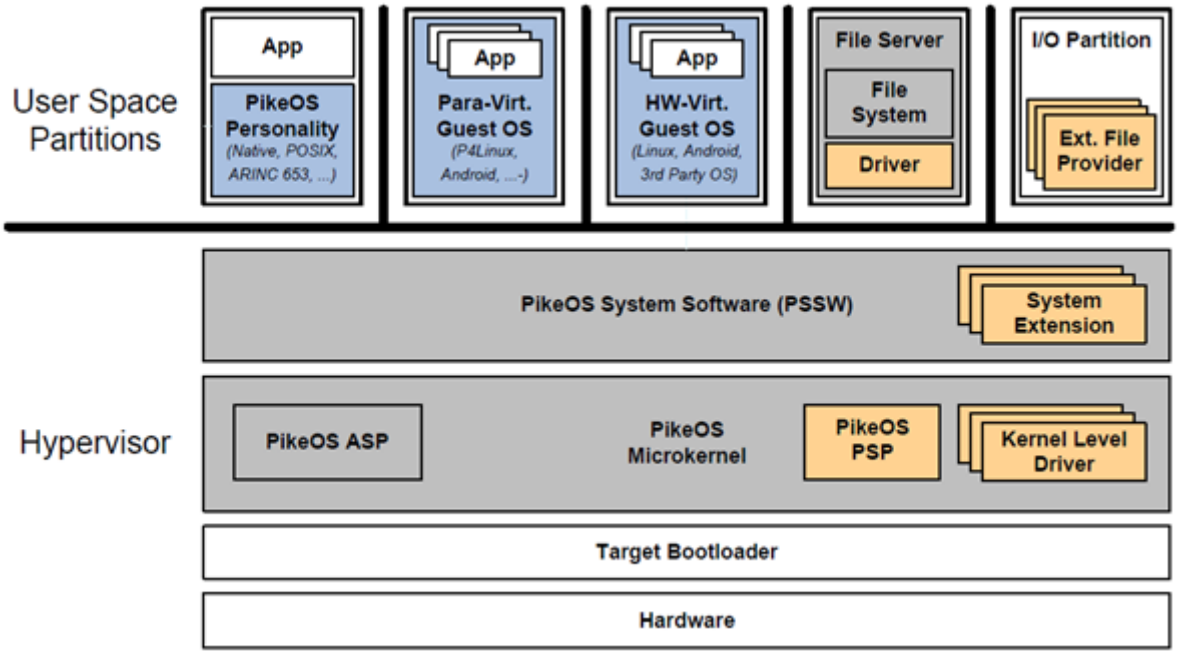

Figure 6: PikeOS Architecture

PikeOS Microkernel

The PikeOS Microkernel consists of a generic part, a CPU architecture dependent part called Architecture Support Package (ASP) and a platform dependent part called Platform Support Package (PSP). The generic part together with the corresponding ASP comes as a library, which is linked together with the PSP and the Kernel (Level) Drivers. The PikeOS Microkernel is the only component that runs with supervisor privileges. The PikeOS Microkernel provides:

- Hardware abstraction
- Resource and time partitioning
- Execution entities (*threads*)
- Separate address spaces (*tasks*)
- Communication primitives
- Timers
- Exception and interrupt handling

PikeOS System Software (PSSW)

The PikeOS System Software (PSSW) component is the first user space application launched by the PikeOS Microkernel. The PSSW component reads the Virtual Machine Initialization Table (VMIT) and initializes partitioning, inter-partition communication and health monitoring according to the VMIT settings. During run-time the system software component acts as a server providing the following services to the

applications executing inside the different resource partitions:

- Communication services via queuing and sampling ports
- File system services
- Health monitoring services
- Partition and process management services

PikeOS Personalities

Partitions are separated environments in which different application programming interfaces (API), runtime environments or guest operating systems can reside. These are called “personalities”. The partitions and the software within the partitions are separated by the operating system in the default configuration. Since each personality is executed inside a partition where it has only access to its allotted resources, it is safe to run safety-critical applications in some partitions, while non-certified personalities are running in parallel in other partitions.

PikeOS Core Concepts

The PikeOS Core concept combines the PikeOS Kernel and PSSW. These provide the main operating system services – like resource and time partitioning, task and thread management, and memory management, health monitoring, exception handling and communication-related services (e.g., IPC and synchronization primitives), and the initialization of the system.

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 or virtual machines. An application running in one partition is completely unaware of applications in other partitions and of system resources to which it does not have access. A partition can be stopped and restarted or reloaded with different applications without affecting other partitions. All resource partitions are created by the kernel at boot time.

At system start-up, all resource partitions except partition 0 are empty, i.e. the partitions do not have any resources assigned. Resource partition 0 is automatically initialized by the kernel to start an initial task. A resource partition is never deleted, only the tasks of a resource partition may be deleted.

Time Partitions

The Time partitioning is a mechanism for allocating CPU time amongst the partitions. It can be used to ensure that all partitions get a predefined amount of execution time and to prevent any thread from starving others, even in the case of a faulting thread.

In its simplest form, time partitioning can be used to allocate a certain CPU quota to each resource partition (though the PikeOS capabilities go beyond this, as discussed below). In this case, there is a one to

one relationship between time and resource partitions. This type of configuration, illustrated in Figure 7, is sufficient for many systems.

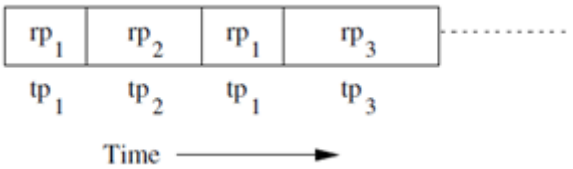

Figure 7: Basic time partitioning

The time partitions and resource partitions are independent. PikeOS time partitioning is not just a case of allocating a certain amount of CPU time to each resource partition. The one to one relationship described above is not mandatory. Multiple resource partitions can belong to the same time partition and in some circumstances, it is possible for different threads from the same resource partition to belong to different time partitions. The second point to understand is that the duration of a (time) window is not a property of the time partition. Windows and time partitions are, again, independent.

Key points of the time partitioning are:

- Static configured sequence of time windows are associated to a time partition.
- Sequences of time windows are cyclically repeated.
- Each thread belongs to a time partition and can only be scheduled when the corresponding time window is active.
- A special “Time Partition 0” is always active in parallel with the currently selected time partition for background or high priority processing.
- Multiple scheduling schemes can be configured and can be switched during run time.
- Privileged partitions are allowed to move threads between time partitions.

Scheduling

User applications started by the PSSW Module are called *processes*. The process can create threads which are the schedulable entities of a task. PikeOS supports multiple concurrent threads and the PikeOS scheduler determines the order in which threads execute. The scheduler is activated whenever an event occurs that alters the state of the thread to be executed next. The strategy implemented by the PikeOS scheduler is Preemptive Priority-based FIFO dispatcher.

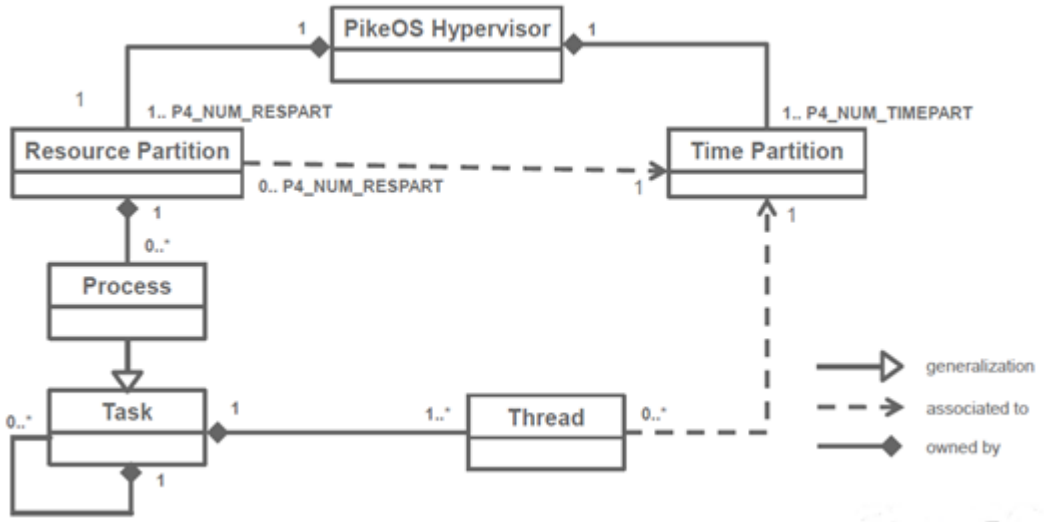

Figure 8: PikeOS Execution Entities

Figure 8 shows a relationship between the PikeOS execution entities which need to be considered for the Runtime architecture. Even though a PikeOS thread is the only object which actually executes application code, the other objects are also part of the execution environment and provide access to the needed platform resources.

- Process - A process instantiates an application (contains the mapping to the code and data sections). It can be seen as a specialization of a task. Depending on the configuration, a process may own additional child tasks.
- Task - A task is mainly representing an address space owning a set of threads and optionally the right to create child tasks.
- Thread - A thread is the (schedulable) entity (of a task) which actually executes the application code.

4.3 Linux as a (soft) Real-Time OS

The Linux kernel can be configured and compiled in a number of significantly different ways for a variety of purposes, making it suitable to a diverse set of use-cases, ranging in the whole spectrum going from high-performance web and cloud computing on multi/many-core servers, to embedded real-time and high-performance devices, to quite small and low-memory micro-controllers. Similarly to any other GPOS, the Linux kernel has been focusing on minimizing average OS overheads and optimizing in-kernel operations so to maximize the average performance for batch and high-performance workloads, while keeping a “good enough” responsiveness for interactive ones, notably user interactions and multimedia applications.

However, Linux has been consistently improving its support for real-time systems through a set of interesting features [Kis09]: the inclusion of POSIX real-time extensions [Obe00] and the support for real-time mutexes; high-resolution timers with nanosecond precision; removal of the Big Kernel Lock (BKL)[17]; enhancements to the kernel preemptibility options; the introduction of NO_HZ for reducing overheads of

[17] More information is available at: https://kernelnewbies.org/BigKernelLock.

the periodic bookkeeping timer; the integration into the mainline kernel of the SCHED_DEADLINE deadline-based scheduler [SAL18] for real-time tasks, initially developed [LFC11] by AMPERE partners SSSA and EVI, and still actively maintained and supported by both.

One of the most significant features that is being integrated into the mainline code base, is the PREEMPT_RT variant of the kernel [RH07, RMF19], which will soon be available as a compile-time switch. PREEMPT_RT introduces a number of enhancements that contribute to lower significantly the worst-case scheduling latency of the kernel, which depends mainly on interferences of hardware interrupt drivers and non-preemptible sections of the kernel on the applications. In PREEMPT_RT, interrupt drivers are turned into kernel threads whose execution and timing can be controlled by the CPU scheduler, and non-preemptible sections of the kernel are greatly reduced. Also, spinlocks are turned into real-time mutexes that are also capable of dealing with priority inversion issues.

This overall results in significant enhancements to the predictability of execution of software on top of the Linux OS, which make it usable as a RTOS for a number of use-cases including robotics and industrial control, automotive and others (at the moment we envision the use of Linux for functions that do not require expensive certification procedures which might not be available for software developed on said OS).

4.3.1 SCHED_DEADLINE on Linux

The mainline Linux kernel includes, since version 3.14, the SCHED_DEADLINE[18] CPU scheduler [LFC11, LSAF16] for real-time tasks, which provides a resource-reservation model based on a multi-processor extension of the constant bandwidth server (CBS) scheduler [AB98], relying on EDF. In resource reservation scheduling, a user-space task can request to the kernel, through a syscall, a scheduling run-time Q and period P, and, if the request is accepted, the task is granted by the kernel Q time units every time window of duration P. The scheduling run-time Q is used to initialize a maximum budget available for the task, which is decreased every time the task is scheduled on a CPU. Once the budget goes to 0, the task is forcibly suspended (throttled) till the beginning of the next period, when the budget is recharged to Q. With the CBS, the kernel keeps track of the current absolute scheduling deadline of each reserved task, and it applies an EDF scheduling policy. On single-processor systems, if the runtime Q is greater than or equal to the worst-case execution time (WCET) of the task C, and the scheduling period P is lower than or equal to the minimum inter-arrival time T between subsequent task activations, then the task is guaranteed to respect all of its deadlines, if all reserved computational bandwidths (Q/P ratios for all tasks) do not exceed the unity. The same property holds for a statically partitioned system, where tasks are partitioned across the available physical CPUs so that each CPU is not over-committed, and it is scheduled using the CBS.

Since kernel 4.16, SCHED_DEADLINE has been further evolved to reduce energy consumption on ARM platforms by implementing the GRUB-PA algorithm and DVFS [SAL18].

However, when it comes to multi-processor platforms, SCHED_DEADLINE implements a simple variant of the original CBS with the ability to be configured as either a GEDF scheduling of all SCHED_DEADLINE tasks across all of the available processors, or as a completely partitioned scheduler, where tasks can be partitioned across CPUs using the cpuset Linux control group, or as a clustered scheduler, where tasks and CPUs are partitioned in groups and each task group is globally scheduled onto each CPU group, which can be done again by a proper use of cpusets. When SCHED_DEADLINE is configured in GEDF (or similarly for the clustered mode), and the overall real-time utilization on the system is greater than 1, then the scheduled tasks are not generally guaranteed not to miss their deadlines, but rather a maximum *tardiness*

[18] More information is available at: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.

bound is guaranteed by the kernel, which can be computed from the parameters of the specific task-set that is being globally scheduled.

4.3.2 Energy-Aware Scheduling of Real-Time Tasks on Linux

When dealing with scheduling of real-time Linux tasks on non-SMP multi-core platforms with energy-aware capabilities (such as DVFS or DPM) as envisioned in AMPERE, a number of unsolved challenges emerge, some of which are planned to be tackled across the AMPERE WP5 and WP3 strands of activities.

First, the Linux scheduler design has a strong focus on SMP platforms. Only recently, a special support has been introduced for non-SMP platforms like the ARM big.LITTLE through the Energy-Aware Scheduling (EAS) framework[19], solely on the general-purpose scheduling policy SCHED_OTHER, and with a big emphasis on the Android variant of the kernel for mobile devices. EAS introduces meta-data information within the kernel about: 1) the different processing power achieved by each of the available cores in each core island, when configured at each operating performance point (OPP); 2) the power consumption associated to each core when it is computing at each possible OPP. This information is leveraged by the default Linux scheduler on Android devices, in order to minimize the energy consumption of the platform by appropriately scheduling tasks on big or LITTLE cores, as well as to enhance the power management logic thanks to capacity-aware improvements to the per-entity load tracking (PELT) mechanism within the kernel.

A recent line of research at AMPERE partner SSSA is aimed at exploiting the information provided by EAS within the SCHED_DEADLINE scheduler for real-time applications. Preliminary results recently obtained by simulation showed that the global EDF (GEDF) scheduling within SCHED_DEADLINE can be modified to realize an adaptive partitioning scheduler with very interesting resulting performance [AC20, MCA20], i.e., outperforming the deadline miss ratio that is achieved compared to the standard SCHED_DEADLINE policy at a variety of overall system utilization. Still by simulations, starting from power consumption and execution times gathered on a real big.LITTLE platform, we showed that an adaptive partitioning scheduler can be made energy-aware, leveraging the kind of information that the EAS framework makes available to the scheduler, obtaining a more energy-efficient execution compared to the regular GEDF in SCHED_DEADLINE [MCM19].

Another recent work from the same line of research [BPC18] showed a non-negligible customization of an application behavior, in terms of actually experienced changes in power consumption and execution time with the CPU frequencies, depending on the actual operations the processor is executing, which impact the intensity of activation of the internal CPU pipeline, as well as how often the CPU stalls waiting for data from the main memory due to cache misses. Further details on these aspects are also discussed in Section 29 below.

Considering the above highlighted issues, it is evident that performing an overall off-line optimization of the system as foreseen in WP3, as well as realizing in the OS kernel and/or middleware proper energy-aware real-time scheduling support, is all but trivial. These challenges will be tackled in the AMPERE WP3 and WP5, respectively.

[19] More information is available on-line at: https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling.

4.4 FPGA and DPR

4.4.1 Software and Hardware Tasks

To support heterogeneous computations with FPGA-based hardware acceleration, the AMPERE run-time environment must be capable of supporting two types of computational activities:

- *software tasks* (SW-tasks): they are computational activities running on the processors, e.g., Linux processes; and
- *hardware tasks* (HW-tasks): they are hardware accelerators implemented in programmable logic and executed on the FPGA.

SW-tasks can speed up parts of their computation by issuing acceleration requests, which in turn involve the request for the execution of HW-tasks. HW-tasks can be either statically programmed on the FPGA or dynamically programmed at run-time by means of DPR. To ensure timing predictability during FPGA reconfigurations and minimize the overhead related to the allocation and interconnection of HW-task, we rely on a *static partitioning* of the FPGA area into N^S slots. Each slot s_k ($k = 1, \dots, N^S$) is composed of b_k resources, with $\sum_{k=1}^{N^S} b_k \leq B$. Resource units are not shared among the slots.

The partitioning of the FPGA into slots depends on the timing and resource requirements of the HW-tasks, and necessitates taking into account floorplanning issues. Algorithms to address these problems are available in the literature [BIBU17, SBB19].

Hardware Tasks

Each HW-task has a static *affinity* to a single slot and can execute only if it has been programmed into it. HW-tasks execute in a non-preemptive fashion. Each slot can be reconfigured at run-time by means of an *FPGA reconfiguration interface* (FRI). As for most platforms (e.g., the Xilinx Ultrascale+), the FRI **(i)** can reconfigure a slot without affecting the execution of the HW-tasks currently programmed in other slots; **(ii)** is a peripheral device external to the processor (e.g., like a DMA) and hence does not consume processor cycles to reconfigure slots; and **(iii)** can program at most one slot at a time. To program a given HW-task τ_i^H into a slot, the FRI has to program all its resources, independently of the number b_i of resource units required by τ_i^H, because unused resources have to be disabled to "clean" the previous slot configuration. The slot hosting a HW-task τ_i^H is denoted as $s(\tau_i^H)$ and also referred to as *affinity*. For all HW-tasks with affinity $s(\tau_i^H) = s_k$, it must be $b_i \leq b_k$.

The FRI is characterized by a *minimum throughput* ρ^x for each type of resource $x = 1, \dots, N^{RES}$, meaning that at most $r_k^S = \sum_{x=1}^{N^{RES}} b_k^x / \rho^x$ units of time are needed to program a slot s_k.[20] Hence, the reconfiguration time r_a needed to program a HW-task τ_a^H is $r_a = r_k^S : s(\tau_a^H) = s_k$.

Software Tasks

Each SW-task can invoke multiple HW-tasks by alternating execution phases with *suspension* phases, where the SW-task is descheduled to wait for the completion of the corresponding HW-task. The same HW-task cannot be invoked by multiple SW-tasks. Every SW-task is activated periodically (or sporadically), with a period (or minimum interarrival time) T_i, thus producing a potentially infinite sequence of execution instances, denoted as *jobs*. Each SW-task is assigned a relative *deadline* D_i, meaning that each of its jobs must complete its execution within D_i units of time from its activation.

[20] For the sake of simplicity, the overhead introduced by the header of the bitstreams is neglected.

SW-tasks and HW-tasks communicate by following a shared-memory paradigm. That is, SW-tasks must first populate a set of input buffers, whose content will be consumed by HW-tasks, then can issue an acceleration request, and finally can retrieve the result of the accelerated computation by reading a set of output buffers.

SW-tasks can be scheduled on Linux either using the traditional POSIX-compliant real-time scheduler (SCHED_RR and SCHED_FIFO), employing multi-processor fixed-priority preemptive scheduling (that can be configured as a global, partitioned or clustered policy across CPUs), or using the SCHED_DEADLINE scheduler described in Section 25. These cases correspond to different analysis techniques that can be used to compute the worst-case response-time of the tasks. From the standpoint of the schedulability analysis, the invocation of a HW acceleration block by a SW task can be considered a suspension of the SW task, to some extent. However, generally, an application will need a direct acyclic graph (DAG) of (SW or HW) task activations, so the full computation of the end-to-end response-time of the application needs more involved techniques that will be studied throughout WP3 and WP4 in the project.

4.4.2 FPGA Acceleration API and Code Sample

An API shall be available to SW-tasks to perform the following operations:

- Initialize the run-time environment to serve acceleration requests;
- Initialize the HW-tasks;
- Allocate the input and output buffers for each HW-task;
- Issue an acceleration request related to a given HW-task with blocking semantic (i.e., the calling SW-task is suspended until the request completes);
- Issue an acceleration request related to a given HW-task with non-blocking semantic (i.e., the control is returned to the calling SW-task after the request has been issued);
- Synchronize a SW-task with the completion of a given acceleration request.

Example pseudo-code for a periodic SW-task that uses the above API is reported below:

```
SW_TASK(example)
{
    init_FPGA_accel_environment();

    // Initialization of the HW-tasks
    HW_Task myHW_Task1 = hw_task_init(hw_task_1);
    HW_Task myHW_Task2 = hw_task_init(hw_task_2);

    // Task body
    periodic_execution(<task_period>)
    {
      << ... >>
      << prepare input data for hw_task_1 >>
      execute_hw_task(myHW_Task1);
      << process output data of hw_task_1 >>

      << ... >>

      << prepare input data for hw_task_2 >>
      execute_hw_task(myHW_Task2);
      << process output data of hw_task_2 >>

      << ... >>
```

```
        }
    }
```

4.4.3 FPGA support

Hardware-level support is also required to serve the execution of HW-tasks. As introduced above, in order to support the deployment of dynamically-reconfigured HW-tasks, the FPGA area shall be divided into two main regions: a *static* region and a *reconfigurable* region. The static region shall contain part of the logic that is needed to realize the communication infrastructure and the HW-tasks that are not subject to partial reconfiguration. The reconfigurable region shall be organized into slots.

It shall be possible for HW-tasks to autonomously access the system memory, i.e., acting as bus masters, and share data with SW-tasks. Hence, bus and memory access represent a crucial contention point.

Bus and memory contention originated by HW-tasks shall be controlled in a predictable fashion. For this purpose, custom hardware modules for managing the AXI bus can be developed. Being the memory ports available to the FPGA typically limited in modern SoC, interconnects (such as the AXI SmartConnect by Xilinx) shall be present in the FPGA design to multiplex the access to memory performed by multiple HW-tasks. Their configuration (in terms of number of interconnections) can be bound independently of the number of HW-tasks as it depends on the total number of slots only.

During the FPGA reconfiguration process, the behavior of the reconfigurable slot is undefined since its logic cells may be in an inconsistent state. Hence the logic cells may generate temporary glitches that originate spurious transactions in other modules, such as the AXI interconnects or the interrupt controller. To solve this problem, each slot shall be protected by a partial reconfiguration *decoupler*, which binds the wires of the slot interface to safe logic values during the reconfiguration process [VUG18]. The runtime software shall be capable of controlling each decoupler through a control register, which can be mapped into the address space of the processors through an AXI-Lite slave interface.

4.5 Energy efficiency

The runtime requirements for energy efficiency are targeting the monitoring and sensing of hardware (and partially software) parameters that are relevant for online estimation of the current energy usage of the system. This information should be used to actuate the system to provide energy efficient operation. The requirements outlined here are based on the platform specifics of the ZCU102 and the NVIDIA Xavier, the commercial platforms selected for the project, as outlined in D5.1. If the runtime provides the necessary mechanisms and policies described here, they will also be applicable for the HERO research platform[21], which is a soft-core RISC-V implementation instantiated on an FPGA. As the HERO platform is fully open-source, hardware adaptations provide a possible means to provide the required input to the runtime monitor for energy efficiency. The HERO heterogeneous research platform also enables the exploration of hardware changes for energy-efficiency within the scope of the project, that would not be possible on the commercial platforms. A schematic overview of different system components involved in the energy efficiency runtime is presented in Figure 9, and their impact on the requirements of specific components of the AMPERE runtime are presented below.
The architecture for energy-efficiency monitoring and actuation in the runtime is based upon energy modeling using both linear and categorical models (see (C) in Figure 9), as described in Deliverable 3.1.

[21] HERO Platform https://pulp-platform.org/hero.html

Besides this, both commercial platforms, the NVIDIA Xavier and the Xilinx ZCU102 (see Deliverable 5.1) include an on-board current and voltage sensor (two INA3221 on NVIDIA Xavier, 4 INA226 on Xilinx US+ ZCU102, denoted as (E) in Figure 9). This hardware unit provides relevant information for energy efficiency, however, we base the requirement specification on linear and categorical models for several reasons. First, these provide a broader framework to collect relevant information about the system state, and designing the runtime around them provides more opportunities for optimization of the runtime as the project progresses. Second, these rely only on information that can be produced on all relevant platforms for the project, including the academic RISC-V-based HERO heterogeneous research platform. Third, initial exploration of the hardware platforms indicate that the INA values are subject to several sampling rate and quantization effects that limit the granularity of control over the system. Furthermore, the reading of the INA values over I2C is a significantly more expensive process than the counter-based approach. A comparison between the model and INA power numbers are presented in Figure 10.

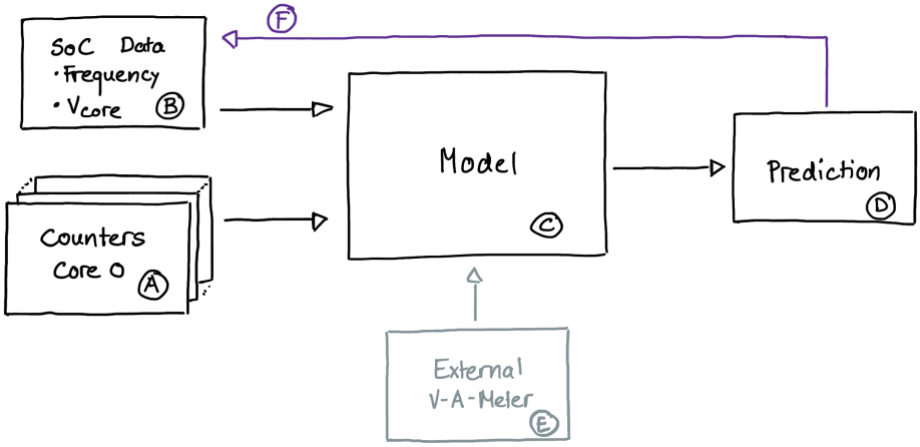

Figure 9: Overview. The model (C) periodically accepts values from the performance counters of the cores (A) and additional metadata from the SoC (B). The model is trained with an external voltage / current meter (E). The model predicts the current power consumed (D); this value can be used to actuate voltage, frequency, and power modes of the SoC (F).

The architecture outlined in this section is derived from prototypes of the energy model, based on measurements on publicly available benchmark suites (e.g., Rodinia) and custom synthetic benchmarks. We believe that this provides a representative scenario that is detailed enough to specify the requirements for the AMPERE runtime from an energy efficiency perspective. However, we recommend these requirements to be interpreted as the minimum requirements to provide a meaningful energy efficiency runtime component, and to provide extensible interfaces that support adaptations that may benefit the industrial use-cases at a later stage in the project.

The requirements described in this section are divided into three parts; Training, Monitoring, and Actuation, and are described in greater detail in the following subsections. Overall, to provide the core services of online monitoring and actuation w.r.t. energy efficiency, **the current intuition is that the sensing, energy modeling, and actuation is suitable for implementation as low-level C programs, without the need for non-standard libraries**.

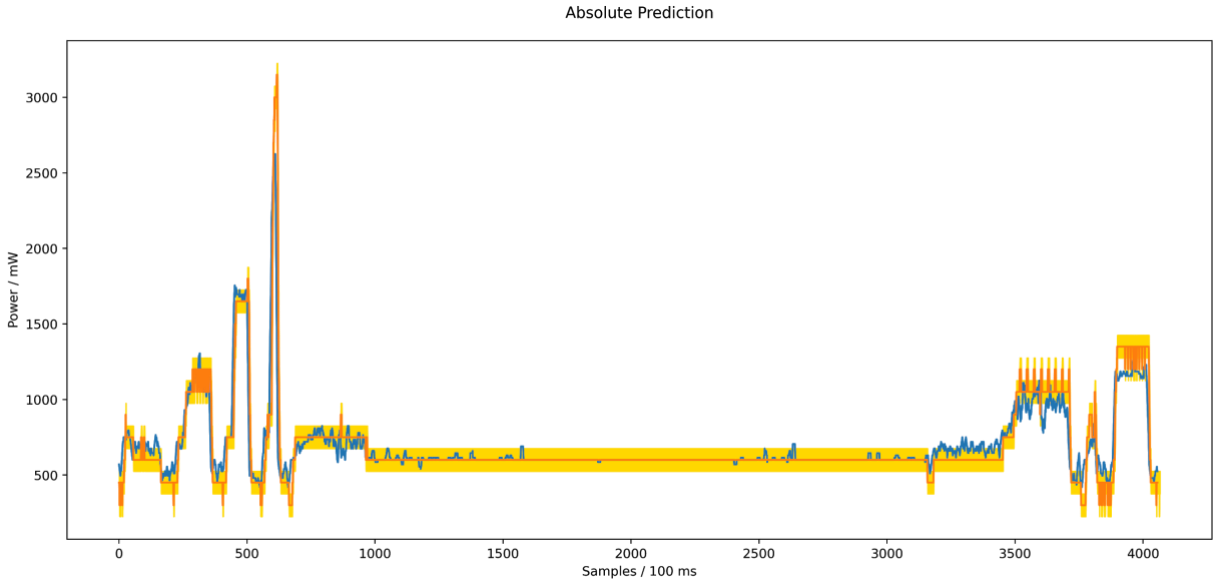

Figure 10: Prediction of the power consumed by the Carmel SoC on the Nvidia Xavier board. The predicted power is shown in blue, the power measured by the INA3221 in orange. Yellow denotes the quantization uncertainty of the power sensor.

4.5.1 Off-line profiling and model training

The online inference in the energy models (C in Figure 9) is using a pre-trained model, and as such the training of the model (and the implied profiling of the hardware) will not be part of the operational runtime (i.e., executed as part of a final AMPERE system). However, the runtime system may provide the necessary infrastructure to complete this profiling, to a) ease the generation of the trained model for the platform, and b) provide the possibility for re-training the model if the user-modeled application includes new workloads that are not well represented by the default model.

The training is done in two distinct steps: a preselection of the counters to include as the source of data, and the training of the model. While the latter of these processes could be done completely offline (e.g., using Python applications), the selection of the counter values and the collection of their values during the execution of the new training workload could be supported by the AMPERE runtime. This is done by tracing the performance counter values (to be discussed shortly), and periodically writing them to disk in a *power trace*, together with metadata such as the power measurements. For systems not yet realized in silicon, e.g., the HERO research platform, VCD-based power simulation can be leveraged to estimate the power consumption. For the commercial platforms used within the AMPERE project, we can either use the on-board current sensing infrastructure or an external current / voltage monitor.

In a system there are countless events that represent the system state. Usually not all of these events can be counted / profiled simultaneously, as this would impose significant hardware overhead in terms of area and power (i.e., one hardware counter per event). While there exists two-three dozen hardware events that can be counted on the ARM cores on the selected platforms (see Deliverable 5.1), only three (NVIDIA) or six (Xilinx) events can be counted at a time in each core. Therefore, the events that are relevant for generating good power estimates must be preselected, such that the counters can be

configured to track the events relevant to energy modeling during online inference.

Preselection can be done manually if there is enough understanding of the micro-architecture of the device. The Processor / SoC can be divided into functional units and it has to be ensured that a counter keeps track of the activity of every important functional unit. This approach is only feasible for open-source platforms as HERO, where there is a good understanding of the hardware operation at each level. However, on closed-source commercial platforms this approach is unfeasible, and automated processes are required.

For the commercial systems used within the project, the current approach is to perform a black-box profiling of all counters, measuring their respective correlation with the power for a specific training workload (currently based on benchmarks, but the approach is extensible to other workloads, e.g., from the project use-cases). As the number of counters available is significantly lower than the number of countable hardware events, this process is repeated for the same workload until all counters have been recorded. Once this information is available, statistical methods are used to identify the set of events that best correlate with the energy usage for the training workload, as shown in Figure 11.

To support this process, and allow for an easy retraining of the model as new workloads become available, **the runtime would need to provide access to the performance monitoring unit (PMU) of the processing systems (e.g., ARM cores, GPU), as well as to any custom counters implemented in the programmable logic (PL) on the Xilinx platform, that capture the behaviour of FPGA accelerators** (denoted as (A) in Figure 9)**.** The PMU can be made accessible from user space, by configuring the PMU through proper use of privileged OS operations. To access custom counters in the PL, the AMPERE runtime should be able to access the physical memory locations at which these counters are located.

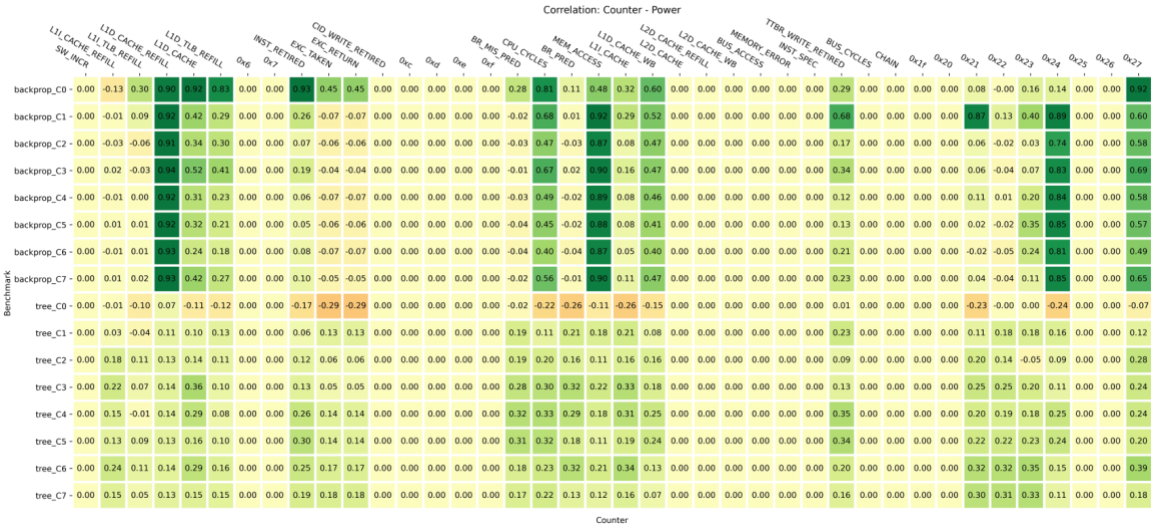

Figure 11: Correlation between the individual counters found in the ARM PMU and the power measured with the INA3221. The correlation is shown for each core and for two selected benchmarks.

4.5.2 On-line monitoring

Counters and hardware parameter monitoring

The energy efficiency monitoring is currently envisioned through the sampling of the system state at fixed points in time. As such, **the runtime environment (including OS/Hypervisor) must provide means to schedule and execute these sampling points at a regular interval**, e.g., through timers. During each execution of the monitoring task, the counter values (that were preselected during training, see above) must be recorded, as they provide the input to the energy model. Furthermore, to read out core-specific events, **it must be possible to schedule these periodic events on all/any cores of the system**.

In general, the finer granularity at which the monitor can run, the finer-grained information the system can be provided with. The trade-off between the accuracy of the prediction, and the runtime overhead has yet to be fully explored. However, in our prototyping, which also uses expensive Linux `/sys`-node readings for data collection, it is possible to collect counter data and all other relevant parameters for training, for all cores, every 100ms at only 5% overhead on the NVIDIA Xavier running at 1.2GHz at training time (including writing power trace to file). With more efficient access to the relevant hardware counters, and without trace writing to disk, it seems reasonable to assume that the runtime overhead can be held within the <1% overhead goal for the runtime, outlined in KPI 3.2, without significant loss in accuracy.

At each invocation of the monitoring runtime, the energy model requires as input the latest performance counter readings. Therefore, **during the online monitoring, the PMU must be made available by the runtime to the energy efficiency monitor**. In contrast to the training phase, however, only the preselected counters are required. Important to notice here is that, due to the limited amount of counters that are available, **there may exist conflicts with other parts of the runtime if they also require hardware events to be counted**. This must therefore be taken into account in the overall runtime design, such that each component can have their respective requirements fulfilled.

Besides the PMU counters, the online inference of the energy usage also depends on the current frequency of the system. This information is used to estimate the amount of cycles the core is power gated and to track Dynamic Voltage-Frequency Scaling (DVFS). In our current prototype we have read this information from the relevant `/sys`-nodes in Linux. While this works well enough for prototyping, it is not guaranteed that this provides the performance required to reach the <1% overhead goal for the runtime, outlined in KPI 3.2. Therefore, if for performance or other reasons this information can not be read from the sys-nodes, **the AMPERE runtime must provide alternative means to access the system frequency (B in Figure 9)**, e.g., through an abstracted (and possibly extensible) runtime API interface. How this is best done depends on the privilege level (supervisor, kernel, user) at which the AMPERE runtime monitor executes.

Furthermore, the Xilinx system provides means to read out the current core voltage via I2C. Unfortunately, we have not discovered any similar feature on the NVIDIA platform, although it seems likely that it also possesses similar features. In our current energy-efficiency monitoring, we are assuming fixed pairs of Voltage-Frequency configurations. However, if it turns out that this is an incorrect assumption, being able to **access the core voltage information (B) from within the AMPERE runtime**

would be beneficial.

Our current exploration of the commercial platforms in the project has not identified any possibility to influence the clock and/or power gating of the system from software, nor any way to directly read out this information. To the best of our knowledge, this is managed completely in hardware. Currently, we are estimating the impact of these features from performance counter values. However, if a way to read this information would be discovered, **having access to information about the current clock/power gating state of the system (B) would be beneficial to the management of the energy-efficiency** of the AMPERE system.

Non-functional requirements

To support the non-functional requirements of the applications, the AMPERE runtime must provide the energy efficiency monitor with the energy budget of the system, such that it is possible to determine if the current operational parameters are in line with the non-functional energy-efficiency requirements described for the application at model design time, as well as the numbers computed offline during the non-functional requirement verification process.

The granularity at which to provide the energy budget of the system depends on the models and structures used by the overall modeling flow, as well as the constructs used by the parallel programming model. Our intuition is that this granularity will be at a per-task level, which is also in line with the requirements outlined on the runtime w.r.t. real-time guarantees (see Section 26). As such, **the AMPERE runtime must inform the energy-efficiency monitor about the energy budget for the next scheduled task or work item**.

Actuation

The main way to actuate the system to provide higher energy-efficiency during online execution is the use of DVFS. Since this feature is typically managed by the Linux performance governors, it may lead to conflicts with the underlying Linux system. The AMPERE system would be significantly simplified if the AMPERE runtime was the only component in the system interacting with the DVFS settings. This would avoid adverse effects due to conflicting optimization goals with the underlying Linux system, and would also simplify the runtime itself, as the energy efficiency monitor estimates the clock/power gating within the processors based on the assumption that the target frequency is constant. Thus, **the AMPERE runtime would either have to be tightly integrated with the Linux performance governors, or completely overrule the decisions made by Linux** (i.e., disable cpufreq).

On the NVIDIA Xavier there exists further the possibility to put *individual cores* into deep sleep mode, allowing less energy to be spent for time quanta where there is no task to be scheduled. It still remains to be experimentally determined how costly this operation is, and at which granularity it can be applied. If this technique is determined to be feasible in practice, its use would **require the AMPERE runtime to inform the energy efficiency component about the workload mapping to computing units for the next period,** so that the relevant decision can be taken at runtime.

Further opportunities identified for actuating the system for energy-efficiency include the explicit management of clock and/or power-gating policies for the system. As outlined previously, no mechanism

to influence this has been found in the commercial systems used as part of AMPERE. However, thanks to the inclusion of the HERO Open-Source Heterogeneous Research System included in the project, there exists going forward the possibility to explore opportunities for online energy management in this direction. As the HERO system is fully open source, the interfaces can be tailored to fit with the previously outlined requirements on the runtime, and no further requirements are introduced. In particular, we envision such an exploration to make use of the custom PL counters interface discussed previously.

5 Conclusions

This document illustrated a first preliminary architecture of the run-time architecture envisioned in the AMPERE project to support the execution of the types of applications represented by the use-cases selected and to be realized in WP6. The document started recalling briefly what are the requirements on the run-time as derived by the two automotive and railway use-cases, with an emphasis on non-functional requirements like high performance, timeliness and predictable execution, and energy efficiency. Then, a preliminary architecture has been presented potentially able to fulfill such requirements, describing the various blocks of the architecture and sketching out the research that the AMPERE partners are undertaking in the various related areas.

6 Acronyms and Abbreviations

Acronym	Extended form
DPM	Dynamic Power Management
DPR	Dynamic Partial Reconfiguration
DSML	Domain-Specific Modeling Language
DVFS	Dynamic Voltage and Frequency Switching
E/E	Electric and Electronic
EAS	Energy Aware Scheduling
ECU	Electronic Control Unit
EDF	Earliest Deadline First
GEDF	Global Earliest Deadline First
GP	General-Purpose
GPU	Graphic Processing Unit
GP-GPU	General-Purpose Graphic Processing Unit
HPC	High-Performance Computing
ISA	Instruction Set Architecture
MDE	Model-Driven Engineering
NFR	Non-Functional Requirement
OPP	Operating Performance Point
OS	Operating System
RT	Real-Time
RTOS	Real-Time Operating System
SIL	Safety Integrity Level
SIMD	Single-Instruction Multiple Data
SOA	Service-Oriented Architectures
SoC	System-on-Chip
VLIW	Very-Long Instruction Word
WP	WorkPackage

References

[AASEM19] Specification of Execution Management, AUTOSAR Adaptive Platform, AP R19-11, 2019. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_SWS_ExecutionManagement.pdf

[AAPP19] Design guidelines for using parallel processing technologies on Adaptive Platform, AUTOSAR Adaptive Platform, AP R19-11, 2019. Available on-line at: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_EXP_ParallelProcessingGuidelines.pdf

[AB98] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems," in Proceedings of the IEEE Real-Time Systems Symposium (IEEE RTSS), USA, December 1998.

[ACTA19] Recommended Methods and Practices for Timing Analysis and Design within the AUTOSAR Development Process, AUTOSAR Classic Platform, CP R19-11, 2019. Available on-line at: https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TR_TimingAnalysis.pdf

[AC20] L. Abeni, T. Cucinotta. "*Adaptive Partitioning of Real-Time Tasks on Multiple Processors*," in Proceedings of the 35th ACM/SIGAPP Symposium On Applied Computing (ACM SAC 2020), March 30th, 2020, Brno, Czech Republic.

[AD11] AMPERE Deliverable D1.1, "System models requirement and use case selection," June 2020

[AD21] AMPERE Deliverable D2.1, "Model transformation requirements," June 2020

[AD31] AMPERE Deliverable D3.1 "Multi-criteria optimization requirements," June 2020

[AECE13] H. Mackamul, "AMALTHEA - An Open Tool Platform for Embedded Multicore Systems", EclipseCon Europe 2013, Ludwigsburg/Germany, Oct 2013. Available on-line at: https://www.eclipsecon.org/europe2013/sites/eclipsecon.org.europe2013/files/AMALTHEA_Project_-_EclipseCon_Europe2013.pdf.

[AWP13] Christopher Brink and Jan Jatzkowski, "AMALTHEA - White Paper", 2013. Available on-line at: http://www.amalthea-project.org/index.php/results

[ADF11] The AMALTHEA Consortium, "Deliverable D1.1 - State of the art of Design Flow and verification methods and tools", 2011. Available on-line at: http://www.amalthea-project.org/index.php/results

[BDN18] C. M. Brandberg and M. Di Natale. Enabling Flow Preservation and Portability in Multicore Implementations of Simulink Models. *Principles of Modeling*. Springer, Cham, 2018. pp. 206-222.

[BPC18] A. Balsini, L. Pannocchi, T. Cucinotta. "*Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures*," in Proceedings of the International Workshop on Embedded Operating Systems (EWILI 2018), October 10th, 2018, Torino, Italy.

[BRA19] C. M. Brandberg. Model-Based Design, Analysis, and Synthesis for Embedded System Deployment on Multicore Architectures. PhD Thesis, Scuola Superiore Sant'Anna, 2019

[BIBU17] A. Biondi and G. Buttazzo, "Timing-aware FPGA Partitioning for Real-Time Applications Under Dynamic Partial Reconfiguration", In Proceedings of the 11th NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2017), Pasadena, CA, USA, July 24-27, 2017.

[DDS15] Object Management Group (OMG), "Data Distribution Service, version 1.4," April 2015. Available at: https://www.omg.org/spec/DDS/.

[IEEE1220] IEEE Standard for Application and Management of the Systems Engineering Process 1220-2005. September 9, 2005 (superseded by IEEE 24748-4).

[KBGW19] L. Krawczyk, M. Bazzal, R. P. Govindarajan and C. Wolff, "Model-Based Timing Analysis and Deployment Optimization for Heterogeneous Multi-core Systems using Eclipse APP4MC," 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019, pp. 44-53, doi: 10.1109/MODELS-C.2019.00013.

[Kis09] J. Kiszka, “Towards Linux as a real-time hypervisor,” in Proceedings of the 11th Real-Time Linux Workshop. Citeseer, 2009, pp. 215–224.

[KS12] C. M. Kirsch and A. Sokolova, “The logical execution time paradigm,” in Advances in Real-Time Systems, pages 103–120. Springer, 2012.

[LFC11] J. Lelli, G. Lipari, D. Faggioli, T. Cucinotta, "*An efficient and scalable implementation of global EDF in Linux*," Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011), Porto, Portugal, July 2011.

[LSAF16] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, “Deadline scheduling in the Linux kernel”, Software: Practice and Experience, 46(6): 821-839, June 2016.

[MAR08] MARTE, A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, OMG Adopted Specification, Beta 2, 2008. Available on-line at: https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf

[MCM19] A. Mascitti, T. Cucinotta, M. Marinoni. "*An adaptive, utilization-based approach to schedule real-time tasks for ARM big.LITTLE architectures*," in Proceedings of the International Workshop on Embedded Operating Systems (EWILI 2019), October 17th, 2019, New York, USA.

[MCA20] A. Mascitti, T. Cucinotta, L. Abeni. "*Heuristic partitioning of real-time tasks on multi-processors*," in Proceedings of the 23rd IEEE International Symposium on Real-Time Distributed Computing (IEEE ISORC 2020), May 19-21, 2020, Nashville, Tennessee, USA.

[OGJH16] B. Ouni, P. Gaufillet, E. Jenn, J. Hugues. “*Model Driven Engineering with Capella and AADL*,” European Congress on Embedded Real Time Software and Systems (ERTSS), Jan 2016, Toulouse, France.

[Obe00] K. Obenland, “The use of POSIX in real-time systems, assessing its effectiveness and performance,” January 2000

[PBDN19] P. Pazzaglia, A. Biondi and M. Di Natale, "Optimizing the Functional Deployment on Multicore Platforms with Logical Execution Time," 2019 IEEE Real-Time Systems Symposium (RTSS), Hong Kong, Hong Kong, 2019, pp. 207-219.

[QRS20] E. Quiñones, S. Royuela, C. Scordino, L. M. Pinho, T. Cucinotta, B. Forsberg, A. Hamann, D. Ziegenbein, P. Gai, A. Biondi, L. Benini, J. Rollo, H. Saoud, R. Soulat, G. Mando, L. Rucher, L. Nogueira. "The AMPERE Project: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization," in Proceedings of the 23rd IEEE International Symposium on Real-Time Distributed Computing (IEEE ISORC 2020), May 19-21, 2020, Nashville, Tennessee, USA (held on-line due to the COVID-19 emergency).

[RH07] S. Rostedt and D. V. Hart, “Internals of the RT patch,” in Proceedings of the Linux symposium, vol. 2, 2007, pp. 161–172.

[RMF19] F. Reghenzani, G. Massari, and W. Fornaciari, “The real-time Linux kernel: A survey on Preempt_RT,” ACM Computing Surveys, vol. 52, pp. 1–36, 02 2019.

[ROQ16] P. Roques. “*MBSE with the ARCADIA Method and the Capella Tool,*” 8th European Congress on Embedded Real Time Software and Systems (ERTS), Jan 2016, Toulouse, France.

[SAL18] C. Scordino, L. Abeni, J. Lelli, “*Real-time and Energy Efficiency in Linux: Theory and Practice*,” ACM SIGAPP Applied Computing Review (ACR) Vol. 18 No. 4, 2018.

[SBB16] I. Sanudo, P. Burgio and M. Bertogna. Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System, Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'16), in conjunction with the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016), Toulouse, France, July 2016.

[SBB19] B. Seyoum, A. Biondi, and G. Buttazzo, "FLORA: FLoorplan Optimizer for Reconfigurable Areas in FPGAs", ACM Transactions on Embedded Computing Systems, Volume 18, Issue 5s, October 2019. Presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2019), New York, USA, October 13 - 18, 2019.

[VUG18] Vivado Design Suite User Guide: Partial Reconfiguration. UG909 (v2018.1). Xilinx , April 27, 2018. Available on-line at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf

[XPM16] Zynq Power Management Framework User Guide For Zynq UltraScale+ MPSoC Devices, UG1199 (v2.0), Xilinx, November 30, 2016

[MKP19] F. Mayer, M. Knaust, M. Philippsen. “OpenMP on FPGAs—A Survey,” In Fan X., de Supinski B., Sinnen O., Giacaman N. (eds) OpenMP: Conquering the Full Hardware Spectrum. IWOMP 2019. Lecture Notes in Computer Science, vol 11718. Springer, Cham