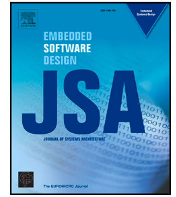
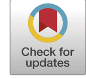
Journal of Systems Architecture 154 (2024) 103219

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc





The MATERIAL framework: Modeling and AuTomatic code Generation of Edge Real-TIme AppLications under the QNX RTOS[☆]

Matthias Becker [a,*], Daniel Casini [b]

[a] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden
[b] TeCIP Institute and Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56124 Pisa (PI), Italy

ARTICLE INFO

Keywords:
Real-time systems
Modeling
QNX
Distributed systems
Edge computing

ABSTRACT

Modern edge real-time automotive applications are becoming more complex, dynamic, and distributed, moving away from conventional static operating environments to support advanced driving assistance and autonomous driving functionalities. This shift necessitates formulating more complex task models to represent the evolving nature of these applications aptly. Modeling of real-time automotive systems is typically performed leveraging Architectural Languages (ALs) such as Amalthea, which are commonly used by the industry to describe the characteristics of processing platforms, operating systems, and tasks. However, these architectural languages are originally derived for classical automotive applications and need to evolve to meet the needs of next-generation applications.

This paper proposes an automatic framework for the modeling and automatic code generation of dynamic automotive applications under the QNX RTOS. To this end, we extend Amalthea to describe chains of communicating tasks with multiple operating modes and to consider the QNX's reservation-based scheduler, called APS, which allows providing temporal isolation between applications co-located on the same hardware platform. Finally, an evaluation is presented to compare different implementation alternatives under QNX that are automatically generated by our code generation framework.

1. Introduction

Distributed systems reliability requires continuous self-awareness of components and systems, as well as the ability to act on failures and unexpected behaviors, e.g., through mitigation or fallback operating modes [1]. This is becoming common in automotive systems, which are now called to support far more complex distributed systems to provide advanced driving assistance and autonomous driving functionalities [2,3]. These systems are essentially built of real-time components that are required to respond within predictable timing bounds; nevertheless, their intrinsic complexity and computational requirements seek the need to deploy them as distributed systems.

Clearly, this makes room for many more sources of timing unpredictability both at the software and hardware level, vulnerability to security attacks, and the presence of software bugs. To mitigate these issues, run-time monitoring and criticality-based decision-making strategies are key pillars to managing such systems and can conveniently leverage tasks offering multiple operating modes characterized by solutions of different quality and computational requirements [1]. For instance, the classic example is a velocity control algorithm that can be offered by two different techniques, e.g., a PID and an MPC controller, with a different degree of accuracy and computational requirement.

To tackle the immense intricacy of these systems, architectural languages (ALs) [4] are often used to design modern systems [5]. In automotive, Amalthea [6] is an AL used to design the non-functional features of applications. Amalthea has been originally defined in the European project Amalthea and later refined in other projects, e.g., AMALTHEA4public [7], Panorama [8], and AMPERE [9]. The Amalthea language allows specifying the characteristics of the hardware platform, operating system, and computational activities (tasks).

However, Amalthea has been designed to model classical automotive applications; instead, we are currently facing a paradigm shift that

☆ The authors would like to thank Paolo Pazzaglia for the insightful discussions about modeling. The work has been partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU and the European Union's Horizon Europe Framework Programme project NANCY under grant agreement No. 101096456 and by the Swedish Research Council (VR) under the project nr. 2023-04773 and by Sweden's Innovation Agency via the Advanced and innovative digitalization project 2021-02484 EARLY BIRD Seamless System Design from Concept Phase to Implementation.

* Corresponding author.
E-mail addresses: mabecker@kth.se (M. Becker), daniel.casini@santannapisa.it (D. Casini).

https://doi.org/10.1016/j.sysarc.2024.103219
Received 1 February 2024; Received in revised form 20 April 2024; Accepted 26 June 2024
Available online 29 June 2024
1383-7621/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

requires profoundly rethinking Amalthea to be conveniently used for next-generation real-time applications, which will be mostly based on POSIX-based OSes like QNX and Linux.

Therefore, ALs need to evolve, including new features, such as the aforementioned concept of multi-moded application, but also the support for chains of communicating tasks and for offloading strategies in edge systems.

The operating system can also offer interesting means of supporting modern applications. For example, the QNX operating system offers the Adaptive Partitioning Scheduler [10], a reservation-based scheduler providing timing isolation between different software components.

QNX is indeed the choice of many automotive OEMs, which favor QNX as their foundational real-time operating system (RTOS), given its ISO-26262 certification at the highest level of assurance, ASIL-D. Therefore, supporting the features of QNX is of utmost importance for modern automotive ALs.

1.1. This paper

This paper presents the Modeling And auTomatic Edge Real-tIme AppLications (MATERIAL) framework, an automatic toolchain for the modeling and automatic code generation of modern real-time applications under the QNX RTOS. It leverages and extends the Amalthea AL to include the aforementioned features and supports affinity-based scheduling and the possibility for multiple offloading strategies. We propose both the mathematical formulation of the model and the Amalthea modeling. We design our system to provide three modeling files: one for the application model, one for the platform model, and the last one for the mapping model. The three models are then condensed into a final *deployment model*, which incorporates all the information. This allows for guaranteeing logical separation between different models, as well as including in the final model additional tasks that depend on the task-to-core mapping. We leverage the models to build an automatic code-generation framework to generate the code of the modeled applications under different alternatives. In the evaluation, we assess the performance of different alternatives that can be generated with our toolchain by leveraging its monitoring capabilities.

1.2. Paper structure

The remainder of the paper is organized as follows. Section 2 provides the needed background on the QNX real-time operating system. Section 3 discusses the related work. Section 4 discusses the proposed framework for modeling and generation of time-predictable core for QNX platforms. Section 5 presents our mathematical modeling of a modern automotive application running on QNX. Section 6 shows how the model is realized in Amalthea and reports on how the elements of the mathematical models are included in Amalthea. The model is explained by leveraging a running example based on a realistic Brake-By-Wire application of a Swedish automotive Original Equipment Manufacturer (OEM). Section 7 reports on the details of the generation and implementation of templates in QNX. Section 8 presents our experimental results based on two case studies, the Brake-by-Wire application and an End-to-End Autonomous Driving Application, in terms of runtime required by the code generation process and footprint of the generated applications. We also show how our framework can be conveniently used to empirically evaluate the performance of different design alternatives by measuring the response times of automatically generated applications under different configurations. Finally, Section 9 concludes the paper.

2. The QNX real-time operating system

We start introducing the basic background information on the QNX Real-Time Operating System (RTOS).

2.1. Reservation-based scheduling in QNX

QNX implements the Adaptive Partitioning Scheduler, a reservation-based algorithm that organizes threads into virtual containers known as partitions. Reservation servers provide a portion of the overall processing bandwidth, while also guaranteeing a maximum service delay [11]. This system allocates a portion of the processing capacity to each partition by managing its execution budget. Each partition operates within a common 100 ms sliding window, with its budget determining the processing time. Upon the budget reaching 0, partitions are throttled, and the budget is gradually restored over time.

The QNX scheduler integrates APS with a fixed-priority scheduler that assigns a static priority to each task. Priorities range from 1 to 255 (which is the highest priority). The highest-priority task with a positive budget for execution is selected to run. Partitions are created by taking budget from the *system partition*, a partition that is initially assigned to 100% of the budget. The scheduler is affinity-based, meaning that for each task, an affinity mask is specified. The affinity mask determines the list of cores in which a task can be executed. Further details are available in [12], presenting the supply-bound function definition for APS used in this context.

Why using resource reservation for edge applications? Modern edge applications need to ensure different applications are isolated from a timing perspective, meaning that an overrun (e.g., due to a bug or a cyber attack) occurring within certain applications does not influence other applications. This is often achieved with a static assignment of applications to computing nodes or to a subset of cores within a node, which, however, can quickly cause resource underutilization in the case of lightweight applications and, anyway, cannot guarantee full timing isolation [13]. Resource-reservation algorithms [14] such as QNX-APS provide means to overcome the resource underutilization issue while still allowing to guarantee timing isolation [10], thus becoming an attractive option for modern edge systems.

2.2. Communication and offloading in QNX

QNX boasts a diverse range of inter-process communication mechanisms, encompassing synchronous message passing, signals, FIFOs, pipes, message queues, and shared memory communication.

While synchronous message passing and signals are embedded in the microkernel, external services handle the rest. Among those, we focus on the synchronous message passing, that relies on three key primitives: `MsgSend()`, `MsgReceive()`, and `MsgReply()`, forming the core of the *client–server* paradigm [15].

The client initiates communication through a blocking `MsgSend()`, awaiting a `MsgReply()` from the server. On the server side, `MsgReceive()` handles incoming messages, allowing the server to process and respond using `MsgReply()`. Unlike `MsgSend()`, `MsgReply()` is non-blocking, enabling the server to seamlessly continue normal processing while the kernel asynchronously transfers reply data to the client.

The synchronous message passing of QNX is an interesting option for implementing task offloading functionalities in edge systems.

Indeed, it seamlessly generalizes to distributed networks of QNX nodes through QNET [16]. QNET, the inherent network manager of QNX, streamlines resource access for system developers by providing a consistent interface across local and remote nodes. It seamlessly presents a unified perspective of interconnected QNX-based devices, creating the illusion of a singular logical computer.

Further information on the synchronous message passing mechanism can be found in [17], presenting a timing analysis that considers the effects of communication between different nodes via QNET as well as APS scheduling partitions.

3. Related work

The papers related to this work mainly target the modeling of systems and provide code generation frameworks. The literature on the

topic is utterly large: therefore, we focus only on a subset of the related papers that we deem more related to our work.

The key features that distinguish MATERIAL from previous proposals are the support for dynamic resource reservation mechanisms (APS), the consideration of the QNX real-time operating systems for automotive, and the support for arbitrary affinities. For example, Perrottin et al. [18] presented the TASTE framework, which allows modeling and code generation for distributed embedded systems. However, unlike MATERIAL, it targets RTEMS and Linux and does not provide support for reservation-based scheduling. Other works consider Time-Division Multiplexing (TDM) Scheduling, e.g., focusing on the ARINC-653 avionics standard [19] for which tools for modeling and automatic code generation exists [20,21]. Nevertheless, TDM scheduling is different than APS partitions: TDM divides the time into static slots, whereas APS is a dynamic reservation-based mechanism. Furthermore, ARINC-653 only targets partitioned scheduling: our work instead considers arbitrary-affinity scheduling, which is more general.

Another related proposal is Gericos [22], which targets space applications: instead, MATERIAL targets automotive systems. Since the context is different, it targets different operating systems (ThreadX, RTEMS, and FreeRTOS). Resource reservation mechanisms are not supported in Gericos, and it targets partitioned scheduling: MATERIAL, instead, supports the more general arbitrary affinity paradigm.

Other papers target the code generation for the AADL modeling language [23], e.g., using Ocarina [24,25]. AADL also considers a similar three-phased execution model (read-execute-write) and an event-driven communication mechanism with task activation (called data port) to those considered in this paper. Despite these similarities, AADL and related works do not support the QNX RTOS, resource reservations, and arbitrary affinity scheduling.

In addition, all the aforementioned works consider different modeling languages from Amalthea, which is widely used in the automotive domain.

Another work that is closely related to us is the work by Rehm et al. [26], which considers the performance modeling (also considering real-time constraints) of heterogeneous hardware platforms with hardware accelerators using Amalthea. Extensions to QNX are also briefly discussed. However, the work in [26] focuses only on the modeling perspective and does not provide any code generation framework. Munera et al. [27] extended the Amalthea language to support parallel computing and the OpenMP parallel programming language. Bambagini and Di Natale [28] presented a framework for the modeling and automatic generation of distributed automotive applications. However, it does not consider the QNX operating system, resource reservation, and offloading mechanisms. Bernardeschi et al. [29] extended the AUTOSAR automotive specification to add security features and provided code generation features. Cremona et al. [30] proposed TRES, a framework for modeling tasks and communications in Simulink, which aims at verifying the impact of execution times and scheduling delays on control performance. Wang et al. [31] proposed a method to automatically generate code for synchronous reactive communication using Simulink.

Overall, no previous work has proposed a framework to model modern multi-moded edge applications under QNX-APS reservation-based scheduling in Amalthea and generate ready-to-use code automatically.

4. The MATERIAL framework

Fig. 1 summarizes the framework presented in this paper. We extend the Amalthea AL to describe the features required by modern multi-mode applications organized in chains of communicating tasks, as well as the QNX reservation-based scheduler and the hardware platform. This information concerning the Application and Platform model is included in the Amalthea files. An Amalthea file is devoted to the Mapping model, i.e., to the mapping of tasks to nodes, cores, and reservations. The three files are combined using the Java API of the App4MC framework [32] into an overall deployment model, which includes all the information from the previous ones as well as additional workloads that become known only when merging the task information with the mapping model. For example, chains of communicating tasks spanning multiple nodes require listener tasks to receive data from the network and to restore the communication based on shared memory buffers, which is used in this paper. The deployment model is then used by our code-generation infrastructure, which leverages the Xtend language [33] to generate ready-to-use QNX code. The framework also provides monitoring capabilities for execution tracing and monitoring of runtime statistics, represented by the monitoring component in Fig. 1.

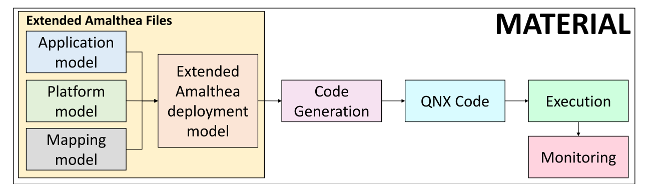


Fig. 1. The MATERIAL framework.

Table 1
Timing parameter of the case study.

Task	Period [ms]	Exec. [ticks]
pBrakePedalLDM	20	1 350 000
pBrakeTorqueMap	30	2 025 000
pGlobalBrakeContr.	40	2 700 000 (Mode 1)
		1 500 000 (Mode 2)
ABS_FL_Pt	50	3 375 000
ABS_RL_Pt	50	3 375 000
ABS_FR_Pt	50	3 375 000
ABS_RR_Pt	50	3 375 000
pLDM_Brake_FL	60	4 050 000
pLDM_Brake_RL	60	4 050 000
pLDM_Brake_FR	60	4 050 000
pLDM_Brake_RR	60	4 050 000

Case Study. We present our contribution by leveraging a running example of a Brake-By-Wire (BBW) application presented in [5]. The BBW application model is based on the implementation by an international Swedish automotive OEM. The case study is comprised of 11 runnables that are mapped to 11 tasks. Therefore, in the description of the case study, runnables and tasks are identical. The Brake-by-Wire application starts by reading of the brake pedal position by the task `pBrakePedalLDM`. The required braking torque is then determined by the task `pBrakeTorqueMap` before the brake signals for each individual wheel is computed in the task `pGlobalBrakeController`. For each wheel, the signal first passes through the Anti-Lock Brake System (ABS) in the task `ABS_XX_Pt` before the final brake signal is determined and sent to the brake actuator by task `pLDM_Brake_XX`. `xx` can be either `FL` (front left), `FR` (front right), `RL` (rear left), or `RR` (rear right). Fig. 2 illustrates the tasks of the application as well as data dependencies between tasks. Table 1 describes the task periods as well as the execution need of the task in ticks. We extended the case study by adding a second execution mode to the task `pGlobalBrakeController`, shown by the two execution times listed.

5. Mathematical modeling

Before discussing the Amalthea modeling, we present a mathematical model to formalize the systems considered in this paper.

5.1. System and platform model

This paper considers a distributed QNX-based system consisting of a set $\mathcal{H}$ of interconnected nodes. Each node is a (possibly heterogeneous)

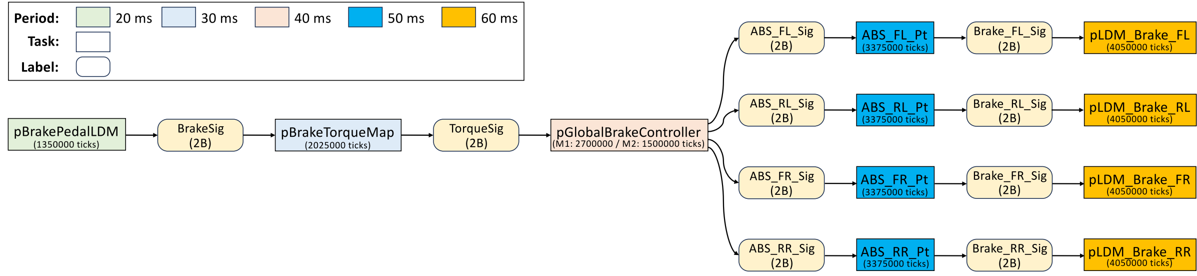


Fig. 2. Overview of the Brake-By-Wire case study. Tasks are shown as rectangles with their name and worst-case execution time in ticks. Communication labels are shown with rounded corners with name and size, respectively. An edge to a label denotes writing to the label and an edge from the label denotes reading the label. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

multicore platform composed of cores running at different speeds. The set of cores of each node $h_l \in \mathcal{H}$ is denoted with $\mathcal{C}_l$, and each core as $c_k \in \mathcal{C}_l$.

5.2. Application model

The system runs a workload that is composed of a set $\mathcal{T}$ of real-time tasks.

As in the AUTOSAR automotive standard [34], each task $\tau_i \in \mathcal{T}$ is composed of an ordered sequence of P_i *runnables* (i.e., functions). Each runnable $\rho_{i,x} \in P_i$ is multi-moded, meaning that it can provide the same functionality with a different degree of accuracy. Different modes of a runnable $\rho_{i,x} \in P_i$ are characterized by a different worst-case execution time (WCET) $e^k_{i,x}$ that depends on the core c_k. Runnable indexes are inversely proportional to the accuracy obtained in the corresponding operating mode (lower indexes correspond to more accurate modes).

Multi-moded applications are beneficial for flexibly adapting the workloads of modern distributed environments, where workloads are dynamic and it could be helpful to partially degrade the accuracy of an already running task to allow admitting and serving a new incoming task while satisfying its temporal constraints.

Each runnable $\rho_{i,x}$ is also associated with a *criticality index* $c_{i,x}$ that can be used to perform degradation decisions. For example, an orchestrator in a distributed system can decide to accommodate a new task on a node by "squeezing" workloads that are already allocated there if the criticality of the runnables of the incoming task is high. An overall criticality c_i can also be specified at the task level (instead of the runnable level).

Tasks are periodic: each task can release an infinite number of jobs (instances), each spaced by its period T_i. A task τ_i has a deadline D_i. Hence, each instance needs to be completed within at most D_i from the release time. We consider discrete time, assuming that time units are an integer multiple of the clock cycle on a specific hardware platform.

On each core, tasks are scheduled according to an *affinity-based fixed-priority scheduling policy*, where each task is characterized by a unique priority π_i, and an affinity mask $a_{i,l} \subseteq \mathcal{C}_l$, which states the cores in which the task can run, from those of the node in which τ_i is allocated to. Affinity-based scheduling [35] is a very general scheduling paradigm supported by QNX, and it can be configured to work both as a partitioned scheduler (by configuring each task to have affinity to one core only) or as a global scheduler (to configure all tasks to have affinities to all cores within a node).

5.3. Timing isolation model

To provide the appropriate degree of isolation between potentially diverse applications that may share the same core, tasks are grouped into *reservation servers* [14] that guarantee a given amount of supply to the tasks assigned to it. This paper focuses on the QNX Adaptive Partitioning Scheduler (APS) [36] as a relevant example of a hierarchical resource reservation algorithm. Under APS, each reservation

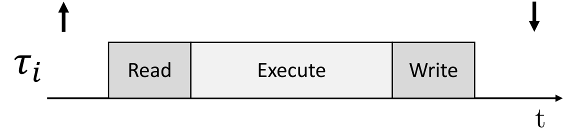


Fig. 3. Three-phase task model.

(called partition in QNX) $r_v \in \mathcal{R}$ is characterized by a nominal budget of B_v time units, which is replenished according to a sliding window technique [12]. The window has length W_l and is common to each reservation in a QNX-enabled platform h_l, and is typically set to 100 ms [36]. The set $\mathcal{R}$ denotes all the APS partitions in the system; the set $\mathcal{R}^x$ contains only those allocated to node h_x. Analogously to tasks, APS partitions follow an affinity-based partitioned scheduling scheme. Tasks are allocated to reservations that, in turn, are assigned to cores. Therefore, the affinity mask of an APS partition is the union of the affinity masks of all the tasks assigned to the partition.

5.4. Communication model

Tasks are characterized by communication dependencies modeled by means of a direct acyclic graph (DAG) where vertexes encode tasks and edges communication dependencies among them. Each communication relation is expressed in the form of a read-or-write dependency with respect to variables, called *labels* in the AUTOSAR standard.

Each runnable is associated with one or more labels, which it can either read or write. The size of an arbitrary label is denoted by $s(\ell_q)$.

When communicating tasks are allocated in the same node, the data exchange occurs by means of shared-memory buffers. Tasks use a three-phase implicit communication model [37]. At the beginning of its execution, each task creates a local copy of the input data (read phase) and updates the labels used for communication at the end of the execution (write phase). The three-phase execution model is shown in Fig. 3.

In the case of tasks running in different nodes, the QNX synchronous message passing mechanism (discussed in Section 2) is used to offload the data-copy operation to a special task (called listener tasks) on the node of the receiver to maintain a coherent communication scheme based on shared memory buffers among functional tasks, as shown in Fig. 4. This task makes the data received by inter-node message-passing primitives available to the consumer in a shared buffer.

Using time-driven communication paradigms (as opposed to data-driven) allows for improving predictability by reducing jitter propagation effects [38,39] and it makes the approach easier to be extended to communication paradigms based on Logical Execution Time (LET) [40] and System-Level LET [41].

The listener task $\tau_{(i,j)}$ implements the communication between a producer task τ_i and the consumer node h_j. Each listener task $\tau_{(i,j)}$ is composed of one runnable $\rho_{(i,j),x}$ that receives a message containing all labels that are written by $\tau_{(i,j)}$ and read by a task on h_j. The runnable

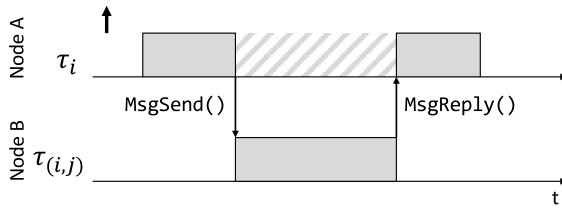


Fig. 4. Communication between task τ_i on node A and listener task $\tau_{(i,j)}$ on node B using the QNX synchronous message passing mechanism.

Table 2
Table of symbols for the mathematical model.

Symbol	Description
$\mathcal{H}$	Set of nodes
C_l	Set of cores in node h_l
c_k	kth core
$\mathcal{T}$	Set of real-time tasks
τ_i	ith task
P_i	Set of runnables for task τ_i
$\rho_{i,x}$	xth runnable of τ_i
$e^k_{i,x}$	Worst-case execution time of $\rho_{i,x}$ on core c_k
$c_{i,x}$	Criticality index of $\rho_{i,x}$
c_i	Criticality index of τ_i
T_i	Task period of τ_i
D_i	Task deadline of τ_i
π_i	Task priority of τ_i
$a_{i,l}$	Cores where task τ_i can run in node h_l
r_v	vth reservation
B_v	Budget of r_v
W_l	Window length of QNX APS instance of h_l
$\mathcal{R}$	Set of all APS partitions in the system
$\mathcal{R}^x$	Set of reservations in node h_x
ℓ_q	qth label
$s(\ell_q)$	Size of label ℓ_q
γ_x	xth event chain
Γ	Set of all event chains

has only one mode and it is characterized by its (data-dependent) execution time $e^k_{(i,j),x}$, where j is the index of the receiving node and i is the index of the sending task. Similar to the other tasks, listener tasks are characterized by a priority $\pi_{(i,j)}$, which is equal to the priority of the sender task. To facilitate the sending of data, a runnable is added to each task that communicates across the node boundary. This runnable is called last and transmits all labels connected to its runnables to $\tau_{(i,j)}$.

The mechanism described above to offload data-copy operations to another node can also be used for general computational offloading operations, which are common in edge-distributed systems.

When two arbitrary communicating tasks τ_i and τ_j communicate, they are also characterized by a communication delay $\lambda_{i,j}$, which depends on the task-to-node (and core) allocation and may also depend on other factors, such as network congestion [42] or memory contention [43].

5.5. Event chains

Tasks in the DAG without incoming/outgoing edges are called *source* and *sink* tasks, respectively. Each source task gives rise to an event chain $\gamma_x = (\tau_f, \dots, \tau_l)$, i.e., a path in the graph. The set of all event chains is denoted as $\Gamma = \{\gamma_1, \dots, \gamma_a\}$. Depending on the task-to-node allocation, the chain can contain listener tasks.

Table 2 summarizes the symbols used in this paper.

6. Amalthea modeling

This section reports on how the running example presented in Section 4 is modeled with our extensions to the Amalthea AL.

Furthermore, it shows how the elements of the Amalthea model are connected to those of the mathematical model, thus establishing a link between the two models. In fact, while Amalthea is more used for software engineering purposes [5], mathematical models are much used for deriving timing analysis bounds of real-time applications [12].

6.1. Modeling the application software

Fig. 5 depicts the software model elements of the case study model and illustrates how different parts of the model relate to each other. In this section, the model elements are presented for the example of the `pGlobalBrakeController` task, denoted with τ_i. This task is activated with a period $\mathrm{T}_i = 40\,\mathrm{ms}$ and is implemented by a runnable called `GlobalBrakeController`.

`GlobalBrakeController` reads the communication label `TorqueSig` and writes the four labels `ABS_xx_Sig` one for each wheel position. `xx` can be either `FL` (front left), `FR` (front right), `RL` (rear left), or `RR` (rear right).

In the mathematical model, this is represented as a task τ_i associated with a single runnable $\rho_{i,1} \in P_i$, accessing four labels ℓ_j, with $j = 1, \dots, 4$.

6.1.1. Communication labels

Communication is modeled via data labels that are accessed by the different tasks via read and write operations. In Amalthea, communication labels are modeled as `Label`. Each label ℓ_j has a distinct name and is assigned a data size $s(\ell_j)$. All labels of the case study are shown in Fig. 5(f). For the `TorqueSig` label, the figure also reports its size, which is 2 bytes.

6.1.2. Task structure

As in AUTOSAR [34], tasks are modeled as sequences of runnable calls. This is shown in Fig. 5(c). A task can thus be thought of as a bin to execute runnables at a specified period. For each task, several properties can be configured. Most importantly, the task stimuli is used to describe the activation pattern.

6.1.3. Task activation

Task activation is modeled via periodic stimuli (Fig. 5(b) and (d)), i.e., each task is periodically activated. Each stimulus is associated with a time value that indicates the period. A task is then connected with the stimuli via the task properties, as shown in Fig. 5(b).

In the mathematical model, this corresponds to the period T_i of an arbitrary task τ_i.

6.1.4. Timing constraints

Each task can have an assigned deadline that describes the allowed upper bound on the task's worst-case response time, as shown in Fig. 5(a). A task deadline is described by a timing requirement of type `ResponseTime`. The constraint is linked to the respective task, and the deadline value is specified as an upper-bound time value.

In the mathematical model, this corresponds to the deadline D_i of an arbitrary task τ_i.

The element property `Severity` can be used to specify the criticality c_i of the task τ_i.

6.1.5. Runnables

Runnables denote the basic unit of execution in the system and describe communication dependencies via access to shared data labels (Fig. 5(e)). In addition, a runnable can operate in different modes.

Communication via labels is modeled via read-and-write access to a label. If the reader and the writer are in the same node, the implementation then uses communications based on shared memory to access the label data. Otherwise, if the reader and the writer are in different nodes, the communication operation in shared memory is offloaded to the node of the receiving task by leveraging the QNX Synchronous message-passing mechanism (as described in Section 5.4).

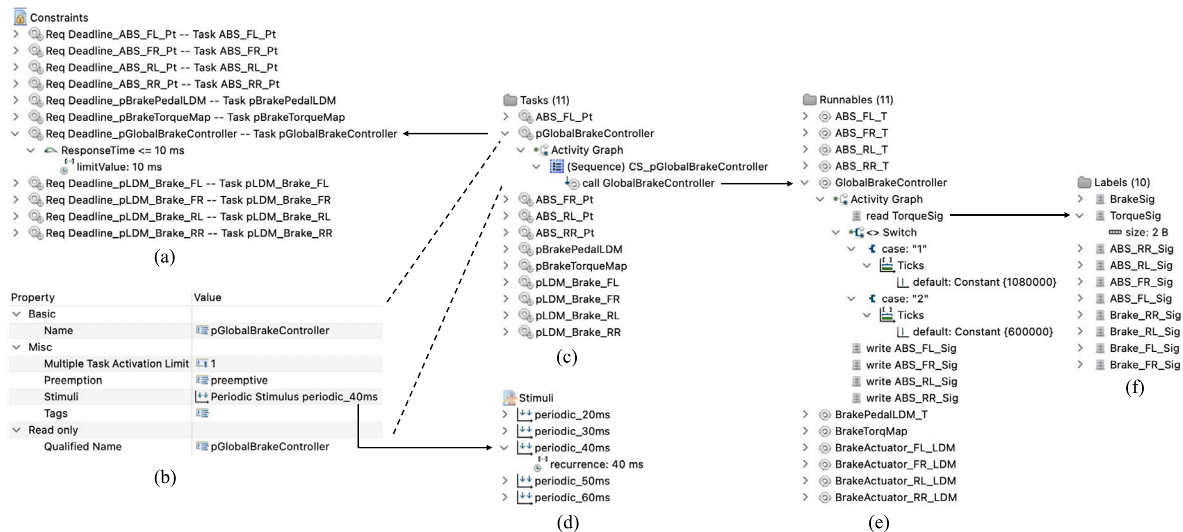


Fig. 5. Elements of the software model. Inset (a): Constraint model to specify deadlines. Insets (b) and (c): Task model and specification of activation period through stimuli model (d). Inset (e): Runnable model. Inset (f) Label model. Arrows indicate how different model elements connect to each other in the example of the task `pGlobalBrakeController` of the case study.

In the MATERIAL Framework, different runnable modes are modeled using the `Switch` and subsequent `Case` elements of Amalthea. Each `Case` element refers to a different execution mode. The name of the `Case` element is used as a condition to execute the mode in the final application (see discussion in Section 7.4.1).

The execution time of each mode is modeled using the `Ticks` element and represents the required CPU cycles to perform the computation. It is worth noting that this allows modeling a core-dependent execution time $e^k_{i,x}$ of a runnable $\rho_{i,x}$ on a core c_k, as required by distributed systems with heterogeneous processing platforms and in platforms with heterogeneous processing cores. This can be achieved by accounting for the frequency domain of the core in which the task is allocated, as explained later in Section 6.2.1. The execution times on high-performance hardware platforms additionally depend on the memory hierarchy, which adds additional latency to the execution [43]. For simplicity, in this work, we do not model the memory hierarchy and memory access pattern of tasks explicitly.

The mode is independent of the communication labels that are accessed by the runnable. Indeed, multi-moded applications typically use different algorithms to produce the same output (e.g., think of different implementations of control algorithms for velocity, which anyway all provide the same output variable). Therefore, label access is described outside the `Switch` element. If a runnable has only one mode, the `Switch` element can be omitted.

Lastly, the criticality $c_{i,x}$ of a runnable $\rho_{i,x}$ can be defined by the property `ASIL Level`, which allows to set criticality levels according to ISO26262 [44]. Note that, in automotive systems, defining a separate criticality value for runnable rather than having just a single value for the whole task may be useful in design activities that provide the functionalities-to-task mapping (and thus the runnable-to-task mapping) to be not fixed but an outcome of the design phase, e.g., see [45,46].

6.2. Modeling the platform and operating system

Besides the application software, the Amalthea model also contains the hardware and operating system model, which are essential for the MATERIALS framework:

- The hardware model describes the physical platform and interconnections between different nodes.
- The operating system model describes the QNX RTOS as well as all configured APS partitions.

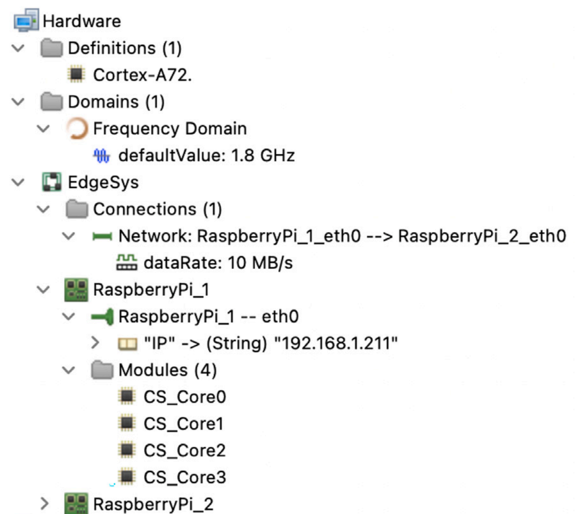


Fig. 6. Hardware model.

6.2.1. Hardware model

The hardware platform is modeled using the Amalthea hardware model. This is shown in Fig. 6, for the case of a Raspberry Pi 4B platform. The top-level entity is a `System` structure that describes the edge system. In addition, definitions are provided for the used CPU cores and frequency domain. The edge system then consists of a number of hardware structures of type `ECU` (which stands for Electronic Control Unit, from the automotive systems' terminology). An ECU is essentially a processing platform, i.e., a node h_l. Each ECU h_l hosts several CPU cores $c_k \in C_l$ instances assigned to the frequency domain. The Ethernet port is also modeled to enable connectivity. It includes a custom property that holds the node's IP address. The connection between hardware nodes is described using a `Network` element that connects the respective hardware ports. A data rate is assigned to denote the link latency, which can be leveraged to derive the communication latency $\lambda_{i,j}$ between communicating tasks τ_i and τ_j.

6.2.2. Operating system model

The operating system model describes the different operating system instances. Fig. 7 shows the operating system model of the case study.

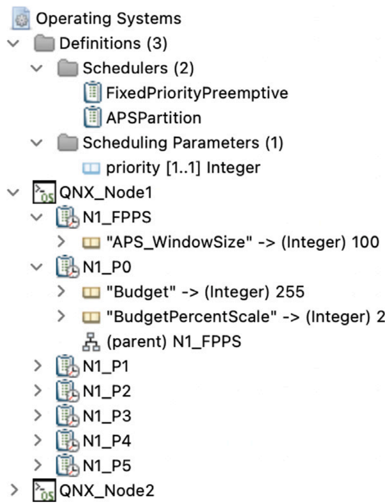


Fig. 7. Operating system model.

In the MATERIAL Framework, each operating system represents the QNX RTOS. Two instances are modeled. The RTOS is modeled as a hierarchical scheduler to describe the APS scheduling of QNX [26]. The first level of the scheduler is a Fixed Priority Preemptive Scheduler (FPPS). A custom property describes the APS window size W_l that is used in the OS (in ms), which is common to all APS partitions on node h_l.

The second scheduling level represents the APS partitions. We model all user partitions. As the budget of the system partition depends on the budget assigned to user partitions, the system partition is not modeled explicitly. All partitions have a custom property that describes the partition budget via the same parameters that are also present in the QNX API, namely `Budget` and the budget percent scale via the parameter `BudgetPercentScale`. As both values are integers, the budget percent scale is used by QNX to indicate at which position the fractional part of the budget starts. For example, a budget value of 15 with a budget percent scale of 1 would indicate a budget of 1.5%.

In the mathematical model, this part of the Amalthea model corresponds to a $r_v \in \mathcal{R}^x$ of a node h_x.

6.3. Software to hardware mapping

The final essential model part describes how the different model elements are connected.

The mapping model is responsible for two main tasks: assign the scheduler to the hardware platform and the tasks to a scheduler. Fig. 8 shows the mapping model and details the scheduler and hardware mapping using an example.

6.3.1. Mapping of scheduler to hardware

Scheduler allocation is used to assign the QNX FPPS scheduler to the platform. The top part of Fig. 8 shows the mapping of the FPPS scheduler of the first QNX instance to node 1. Since the affinity-based scheduler can, in principle, work also as a global scheduler, it is assigned to all four cores of the platform. This is done by setting the `Responsibility` property of the mapping with all four cores. The scheduler is assigned to the first CPU as `Executing PU`. All APS partitions are child partitions of the FPPS scheduler and must, therefore, not be assigned to dedicated hardware nodes.

6.4. Mapping of tasks to scheduler

The task allocation assigns one task to a responsible scheduler. The bottom part of Fig. 8 shows this at the example of the task `pGlobalBrakeController` which is assigned to the APS partition `N1_P1` by setting the respective property `Scheduler`. In addition, the core affinity of the task can be selected using the property `Affinity`. One or more executing cores are assigned to the `Affinity` property of the task mapping element. Note that each core included in the affinity group must be a core for which the scheduler (or its parent scheduler) is responsible for. In the mathematical model, this is related to the task affinity $a_{i,l}$ for an arbitrary task τ_i.

If a task is instead managed by an APS partition, it is assigned to one of the APS partitions created in the Operating System model. Furthermore, tasks managed by the System Partition are also supported by assigning them to the FPPS scheduler of the node.

The set of selected cores are then possible candidates to execute the task. The priority π_i of a task τ_i is assigned to the mapping as scheduling parameter of type `priority`.

7. QNX implementation and template-based code generation

This section describes the code generation process and the implementation of the application on the QNX platform.

7.1. Overview

All required system functionality to implement an application according to the application model is realized in a static QNX application that provides mechanisms for managing tasks, runnables, APS partitions, and inter-node communication. Additional source code and header files are generated to describe the application-specific configuration.

The automatic code generation process can be divided into several parts:

1. Parsing the Amalthea input files to the internal model.
2. Extending the internal model with communication channels.
3. Generating source code and header files for configuration and runnables.
4. Compiling of the QNX application for each node.

The complete code generation framework is implemented in App4MC [32] using the Java interface to operate on Amalthea models. In the following, each step is described in detail.

7.2. Base application

A QNX base application is used to provide all files that are independent of the code generation. In fact, many application modules are static and only require generated configuration files. The software architecture of the application is shown in Fig. 9. The *APS* and *Timer* modules provide an abstraction to manage the creation and modification of APS partitions in the system and to access different time-related functionalities, such as the implementation of the periodic activation of tasks. Concerning tasks' communication, two communication modules are implemented: the *Label* and *Channel* modules. The former provides functionality to manage communications between tasks in the same node through data labels and the corresponding read and write operations. The channel module implements communication across nodes using the QNX synchronous message-passing API. The framework is designed in a modular way such that alternative implementations of the channel module can be provided that utilize different communication forms (e.g., POSIX sockets). Runtime monitoring and execution tracing is supported by the *Monitoring* module. The `task` module realizes task functionality and the `runnable` module implements the main application logic. All but the `timer` modules need to be configured for realizing a specific application. The implementation of these main components (excluding the generated files) amounts to 2577 lines of code.

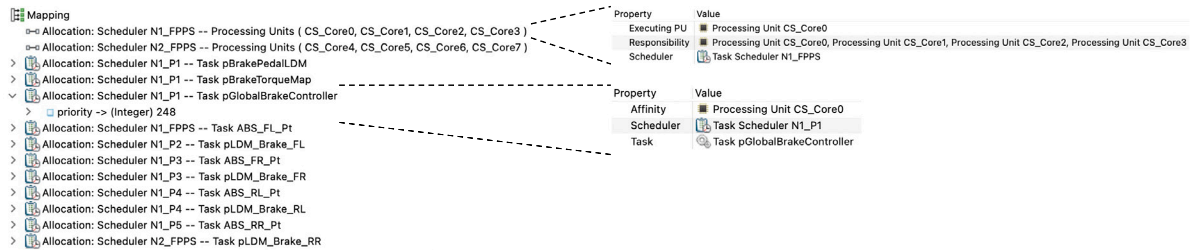


Fig. 8. The mapping model depicts the mapping of the FPPS scheduler to hardware resources (top) and the mapping of tasks to a scheduler (bottom).

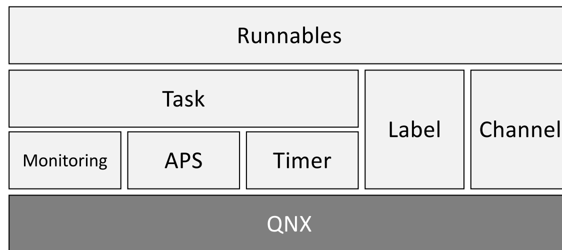


Fig. 9. Main modules of the application architecture.

7.2.1. Task implementation

As seen in the application model, all tasks follow the same operation sequence and implement a periodic or sporadic (in the case of listener tasks) activation. We leverage this fact by providing a common task function that is used by all tasks. The task code first sets the task period and assigns the task to the correct APS partition, before a barrier function is called to wait for the common release of all tasks on the node during the startup phase.

The main part of the task consists of a while loop that first waits for the next release of the task (in the case of listener tasks, waiting is not performed). The remaining part implements the 3-phase model we assume. For each iteration of the while loop, which corresponds to a task instance, copies of all input labels for runnables of the task are created, followed by the execution of all runnables. Finally, all output labels are written back to the main memory for each runnable.

Task-specific configurations are stored in a data structure that is used to manage the task. Among other parameters, this data structure links to a list of data structures that describe each runnable called by the task, as well as the task period, core affinity, and APS partition. The configuration of these task structures is a result of the code generation process.

7.2.2. Runtime monitoring and execution tracing

The application performs two possible types of runtime observation, *execution tracing* and *monitoring of runtime statistics*. Execution tracing is based on the QNX event tracing infrastructure. If enabled, scheduling and user events are stored in a buffer of the kernel. An external application, such as the QNX data capture utility, can then retrieve those events. If event tracing is enabled in the application, an event filter is configured to only collect relevant events for the application, and the QNX trace logger application is started during program initialization and triggered to collect events. Event tracing adds minimal overhead to the kernel [15].

Our implementation also offers the collection of task-specific runtime statistics. If enabled, maximum observed response times, as well as execution times and deadline misses, can be logged, or alternatively, all values for each job can be saved (suitable for short experiments only). Response time values are obtained by subtracting the finish time of the task's job from the release time of the job.

Since response times include the QNX RTOS overheads (e.g., context switches), the MATERIAL monitoring infrastructure allows for empirical experimentation of a selected configuration (e.g., QNX APS budgets) in such a way as to determine if the timing constraints of the application are satisfied or to allow fast re-deployment with a new configuration otherwise.

To reduce the measurement overheads, the standard timer functions of QNX are used to record timestamps, with a measurement granularity of 1 ms.

7.3. Parsing and generation of internal data structures

As highlighted in Fig. 1, three Amalthea files are used as input to describe application, platform, and mapping-related aspects, respectively. The application input file contains the Amalthea models: `Software` (containing tasks, runnables, and labels), `Stimuli`, and `Constraints`. The platform input file contains the Amalthea models: `Hardware` and `Operating System`. Finally, the mapping input file contains the Amalthea model `Mapping`. With this, all aspects of the software application can be described independently from the platform or specific deployment. This achieves separation of concerns between software modeling and deployment on a specific hardware platform.

All input Amalthea files are parsed using the App4MC Java interface and the different model elements are combined into a new internal Amalthea model analogous to the model described in Section 5. This allows for direct operations on the model in relation to the mathematical model assumed for this work. It also allows the generation of entities that are not present in the application model provided as input, such as, for example, the APS System Partition discussed next.

7.3.1. APS system partition

The APS system partition in a QNX platform is the base partition that always exists. Initially, it has all the available processing time as budget. When a new partition is created, the budget assigned to such a partition is subtracted from the system partition. Since its budget depends on the budget of all other partitions, to avoid redundancy, the system partition is not explicitly modeled in the Amalthea model but is added during the code generation process. The budget of the system partition is set such that it receives all remaining budget that is not assigned to any of the user partitions on the specific node. All tasks that are mapped to the FPPS scheduler of the node are then assigned to the system partition.

7.3.2. Generating listener tasks

Listener tasks are not explicitly modeled in the Amalthea model but are a result of the mapping of tasks to nodes and of runnables to tasks. A dedicated runnable is added to each task for each node that hosts runnables that consume labels written by the task. This runnable is then responsible for sending the labels that are read on the other node using the synchronous message passing API of QNX. Similarly, a listener task is added on each receiving node. Such a task is not periodic, but it is triggered by the synchronous message-passing mechanism. Each listener task consists of a dedicated runnable that receives the data through the network and stores the label values in shared memory.

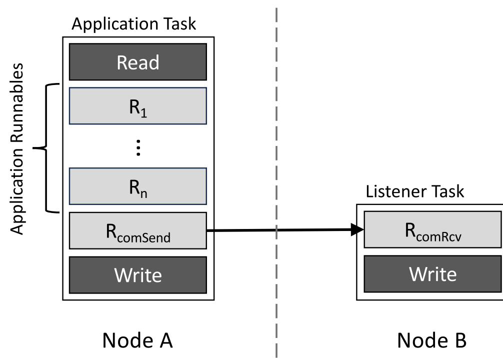


Fig. 10. Communication across node boundaries.

Listing 1: Generated code for the runnable execution

```
__attribute__((weak)) void r_GlobalBrakeController(runnable_spec_t*
      spec) {

   //Simulate the execution of each mode
   switch (spec->mode) {
      case 1:
         burn_cycles(600, spec->cid); // Execute 600 us
         return;
      case 2:
         burn_cycles(333, spec->cid); // Execute 333 us
         return;
      default:
         return;
   }
}
```

Fig. 10 depicts an example scenario with an application task on node A and a listener task on node B. The communication is encapsulated in the sender and receiver runnables which does not require any specific logic in the task function to support inter-core communication.

7.4. Code generation

During the code generation phase, all required source files are generated to configure the different modules. The Xtend framework is used for template-based code generation, which can directly operate on the internal model created in step 2.

7.4.1. Runnable code

For each runnable in the application, three functions are generated to handle initialization, execution, and deinitialization, respectively. If the runnable is a user runnable, those functions are generated with the attribute `weak`, which allows users to include their own implementation of these functions to the code-base without the need to modify generated code. This is done by adding a function definition to the project that has the same name but omits the `weak` attribute. In this case, during linking, the user-supplied functions are linked instead of the generated functions. The generated function for the runnable's execution phase calls a timer function that simulates the execution time by busy waiting. That way, runtime characteristics of applications can be evaluated even without the actual user logic of each runnable.

A dedicated data structure is used to keep track of each runnable's information. This includes function pointers to the runnable's functions as well as lists of references to all labels that are read and written, respectively, and memory for their local copies that are created during the runnable execution. An example function is shown in Listing 1 of runnable `GlobalBrakeController`. The runnable has two execution modes which are selected based on the `mode` variable in the runnable data structure, which is given as an argument to the function.

Listing 2: System Configuration Excerpt

```
#define REGISTER_APS_PARTITIONS \
aps_add_partition_description("N1_P0", 250, 1);\
...

#define REGISTER_THREADS  \
thread_addThreadDescription("pGlobalBrakeController", 40, 10, 248,
      CORE1, N1_P0);\
...\
/* Register runnables for all tasks */\
thread_registerRunnable("pGlobalBrakeController", &
      GlobalBrakeController);\
...
```

Listing 3: Example of a Channel Configuration

```
typedef struct {
   msg_common_t common; // Message header and type
   uint8_t payload[8];  // Payload of the Message
} pGlobalBrakeController_to_RaspberryPi_2_t;
```

If a runnable is generated for inter-node communication, the `weak` attribute is omitted, and the functional code to send/receive the message is generated instead.

7.4.2. System configuration

The system configuration header file includes the required information on all tasks and APS partitions of the node with the assignment of runnables to tasks.

This is done by defining the two different multi-line macros `REGISTER_APS_PARTITIONS` and `REGISTER_THREADS`, as shown in the excerpt of the configuration file in Listing 2.

For each APS partition, `REGISTER_APS_PARTITIONS` includes one function call to a function that adds a new APS partition to the application. The listing shows the example of the APS partition named `N1_P0`, which has a budget of 25% (`Budget` of 250 and `BudgetPercentageScale` of 1). For each task, `REGISTER_THREADS` calls a function to add a new task to the system. Parameters of the function call include the task name, period, deadline, and priority, followed by the assigned CPU and APS partition. Later, runnables are linked to the tasks. The generated macros are called during system initialization.

7.4.3. Label configuration

The label configuration describes the communication labels present in the application as well as their size. Each label is protected from concurrent access using a double buffer implementation. This has the advantage of low overheads compared to lock-based approaches [47].

7.4.4. Channel configuration

The channel configuration consists of the definition of data structures that are used as messages across the different channels. Each struct has a distinct payload size which is large enough to send all labels that are communicated over the channel. Listing 3 shows the generated data type for the message between `pGlobalBrakeController` and `RaspberryPi_2` in mapping 2. The struct has two members, `common` is itself a struct that holds the QNX message header as well as the message type information. `payload` is a simple buffer of a size large enough to hold all data elements sent via the channel. In this case, four labels are sent. Each has a size of 2 B.

7.5. Compilation and deployment

As a final step of the framework, the applications for each node are compiled for the target platform. Dedicated compilation scripts are used to compile the QNX projects. The compilation is automatically triggered after the previous steps of the framework are completed.

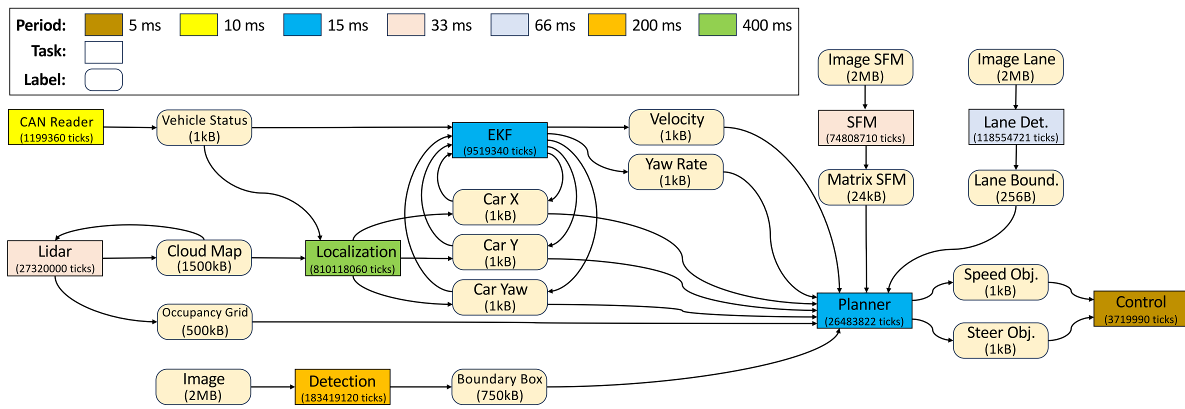


Fig. 11. WATERS 2019 application model. Tasks are shown as rectangles with their name and worst-case execution time in ticks. Communication labels are shown with rounded corners with name and size, respectively. An edge to a label denotes writing to the label, and an edge from the label denotes reading the label. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In addition to compilation, scripts are also provided that allow an automatic transfer of the final application to each QNX node. This requires that the QNX nodes are reachable from the development platform using the same IP address as defined in the Amalthea model.

8. Evaluation

The evaluation assesses experimentally the performance of different, automatically-generated, implementation choices for the realistic case study described in Section 4 and a second case-study based on an automotive end-to-end autonomous driving application described in this section.

The experiments are performed on a distributed hardware platform consisting of two Raspberry Pi 4B nodes that are connected by an Ethernet network. Each Raspberry Pi node has 4 cores and 4 GB RAM. The QNX 7.1 Software Development Platform (SDP) is used. The implementation of the MATERIAL framework, including all evaluated application models, is publicly available.[1]

8.1. Description of case studies

The BBW case study was originally described considering a microcontroller platform with a clock speed of 300 MHz and a static embedded real-time operating system. To get more representative period and execution times for the type of edge applications that are targeted with the MATERIAL framework, the worst-case execution times tick values of the original case study are multiplied with a factor α and the task periods with a factor β. Different values for α and β have been evaluated. For the presented experiments, we considered $\beta = 10$. This results in period values of at most 60 ms. Such values are representative of realistic period values of real-time edge applications [48]. α is set to 25, which results in execution times between 0.75 ms and 2.25 ms. Note that these execution times refer to runnable execution times only. Execution on the target platform additionally includes the read and write operation to memory. Values in Table 1 represent the adjusted values used here. Each task is further assigned a distinct priority. Rate monotonic priority assignment is applied. Tasks `ABS_XX_Pt` and `pLDM_Brake_XX` exist for each wheel, i.e. `XX` can be `FL` (front left), `FR` (front right), `RL` (rear left), or `RR` (rear right). In this case priorities are assigned in this order (high to low).

As a second larger case study, an end-to-end autonomous driving application is investigated. The application was originally presented as part of the 2019 WATERS Industrial Challenge [48] and is representative of next-generation Advanced Driver Assistance Systems (ADAS). The case study represents all necessary tasks to realize driving functionality and to react on stimuli from the environment. In total 9 tasks are described that communicate via 16 labels. In contrast to the BBW case study, the workload of this case study is more demanding and the size of communication labels are significantly larger (between 256 B and 2 MB). Fig. 11 shows the application model. The task periods are indicated with different colors. The size of labels and the execution time of tasks (in ticks) is shown. An arrow from a task to a label describes that this task writes to the label, and an arrow from a label to a task describes reading from the label.

The application obtains the estimated pose of the vehicle from a point cloud (provided by the `Lidar` task) that is initially processed together with the vehicle status information (received from the CAN bus via the task `CAN`) by a particle filter (task `Localization`). Finally, this information is processed by an Extended Kalman Filter (task `EKF`). The main task of the application is to compute and follow a trajectory. Several tasks provide necessary input data. The Structure-From-Motion (task `SFM`) task computes 3D models from 2D images. Lane boundaries are identified by the task `Lane Detection`, and objects on the road are identified by task `Detection`. Based on this information, the vehicle trajectory and required speed and steering signals are computed by the `planner` task before the `control` task computes the final parameters to control the car and sends them via the CAN bus. A more in-depth explanation of the case study can be found in [26].

The original case study reports execution times (in ticks) for the different compute nodes on a Nvidia Tegra TX2 platform. We scale the values reported for the ARM Cortex-A57 cores of the Nvidia Tegra TX2 by a factor $\gamma = 0.775$ to account for the performance difference on the ARM Cortex-A72 cores on our platform [49]. Fig. 11 reports the scaled values. The resulting application has a total utilization of 4.78 and can thus not be scheduled on a single Raspberry PI 4B. Each task is assigned a distinct priority, where rate monotonic priority assignment is applied. In the case of `EKF` and `Planner`, which both have a period of 15 ms, `Planner` has a higher priority than `EKF`. Similarly, in the case of `Lidar` and `SFM`, which both have a period of 33 ms, `Lidar` has the higher priority.

We initially evaluate three scenarios: **(i)** a mapping of the BBW application to a single Raspberry Pi node, **(ii)** a mapping of the BBW application to two Raspberry Pi nodes and **(iii)** a mapping of the WATERS 2019 application to two Raspberry Pi nodes. In all mappings, each task is statically assigned to a core (following the partitioned scheduling [50] approach) by selecting only one task in the property `Affinity` of the task mapping elements in the Amalthea model. Each of the three mappings is generated such that all tasks can meet their deadlines.

Mapping 1 - BBW-Local: Single Raspberry Pi Node

The first three tasks of the application `pBrakePedalLDM`, `pBrakeTorqueMap` and `pGlobalBrakeController` are assigned to the APS partition named `N1_P0`

[1] https://github.com/ESRTS/MATERIAL_Framework.

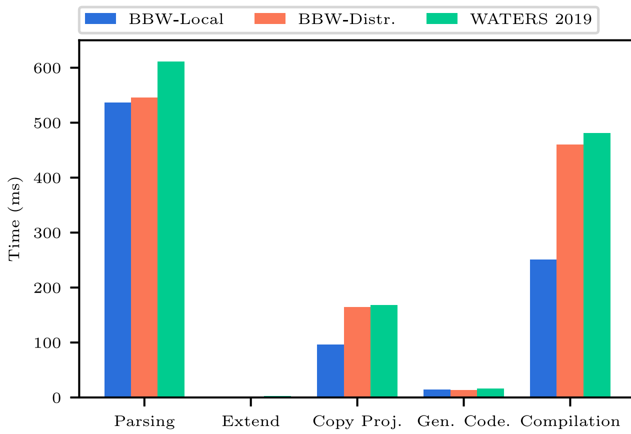


Fig. 12. Runtime of different steps for mapping of the BBW application on one and two nodes, as well as the WATERS 2019 application on two nodes.

with a budget of 25%. All tasks are executed on core 1. The remaining tasks are assigned to the APS System Partition. Tasks are assigned in pairs `ABS_xx_Pt` and `pLDM_Brake_xx` (where `xx` stands for the respective wheel position). Tasks that handle the front left wheel are assigned to core 0. Tasks that handle the front right wheel are assigned to core 1. Tasks that handle the rear left wheel are assigned to core 2, and tasks that handle the rear right wheel execute on core 3.

Mapping 2 - BBW-Distr.: Two Raspberry Pi Nodes

The mapping on two nodes maintains the mapping of the initial three tasks on core 1 on node 1 and the APS partition `N1_P0`. The remaining tasks are assigned to node 2, where one core is reserved for the tasks pair of each wheel direction. All tasks on node 2 are assigned to the system partition. The additional listener task for `pBrakeTorqueMap` is mapped to core 0 of the system partition.

Mapping 3 - WATERS 2019: Two Raspberry Pi Nodes

For this mapping, partitioned scheduling is used. The following tasks are allocated to node 1. Tasks `CAN Reader` and `Lidar` are located on core 0. `Localization` is allocated on core 1, and `EKF` and `Detection` are allocated on core 2. On the second node, the remaining tasks are allocated to individual cores, as their respective utilization does not allow co-allocation. With this assignment, seven labels with a total data size of 1.255 MB are communicated between the nodes.

8.2. Runtime of framework steps

In this experiment, the runtime of the framework is compared for the three mappings. The runtime is separately reported for the different steps outlined in Section 7.1, namely, the *parsing*, the *extension* of the internal model, *copying* of the base application, *generating code* files for the applications, and finally *compiling* the projects. Fig. 12 presents the resulting running times. The results show that the parsing of the Amalthea model and the final compilation of the QNX application require the largest amount of time. All other steps are performed in 168 ms or less. Step 2, where the model is augmented with additional elements to represent the APS system partition and inter-node communication is performed in 0.74 ms, 1.03 ms and 1.97 ms for the mapping of BBW-Local, BBW-Distr. and WATERS 2019, respectively. The overall runtime of the three approaches does not differ significantly. Steps that operate on the parsed model and generate the final code are very fast and do not result in visible differences in the final runtime. The largest impact on the observed runtime is due to the number of nodes in the system.

The results highlight the applicability of the framework to practical engineering tasks without adding significant overheads to the design process.

Table 3
Footprint of the application, in byte.

Application	Text	Data	bss	Total
BBW-Local - RPI 1	44 168	3596	640	48 404
BBW-Distr. - RPI 1	40 180	2428	576	43 184
BBW-Distr. - RPI 2	43 240	3228	592	47 060
WATERS'19 - RPI1	45624	3844	26 586 824	26 636 292
WATERS'19 - RPI2	45380	3668	24 131 328	24 180 376

8.3. Footprint of the QNX application

Besides the runtime of the framework to generate the QNX applications, the resulting footprint of the application is of importance as well.

Table 3 shows the code size of each compiled application in the three mappings: the first line reports the code size of the single Raspberry PI 4B (RPI 1) used in Mapping 1, the second and third lines report the code size of the two applications deployed in the two Raspberry platforms (RPI 1 and RPI 2), and the fourth and fifth line report the code size of the two deployed applications in the WATERS 2019 application on both nodes, respectively. All presented values follow the Berkley format, i.e., read-only data is counted in the text segment. It can be seen that the memory segments of the application under mapping 1 are always larger than the segments of the two applications under mapping 2. Under mapping 2, the application on Raspberry Pi 2 is larger, as the majority of the tasks are allocated here. For all mappings of the BBW applications, the total application size is below 50 kB. The memory requirements of the WATERS 2019 application are comparable for the text and data segment but significantly larger for the bss segment due to the large label sizes of the application. The memory overhead of applications is, therefore, small in comparison to the available memory on platforms in the target area of edge computing.

8.4. Execution of the BBW application

Here, the execution of the final BBW applications on the real platform is evaluated. For all experiments, the application is executed for 30 s on the target platform.

Fig. 13 shows an execution trace of Mapping 1 recorded for the first 85 ms of execution. The different cores the tasks are allocated to are highlighted by a shaded background and labeled on the right side of the diagram.

The monitoring of runtime statistics is used to observe the response times of all periodic tasks under both mappings. This is done to evaluate the effect of the different mapping, as well as the benefits of the framework to collect runtime statistics of the application. Furthermore, the collection of execution traces can be extremely useful for troubleshooting activities.

Fig. 14 show the recorded maximum response times of all periodic tasks of the BBW case study. In addition to mapping 1 and mapping 2, we evaluate four additional mappings of the same task on one node. In particular, we utilize the core affinity of a task to allow all tasks to execute on the same set of cores (varying the number of available cores from 1 to 4). With this configuration, the tasks are then scheduled globally on the allowed cores. This further demonstrated the flexibility of the framework to utilize different allocation strategies. Comparing mapping 1 and mapping 2, task `pGlobalBrakeController` experiences different response times, where the execution with mapping 2 yields a larger response time. This is due to the additional communication overhead the task has for sending the label values to the second node. Similarly, it can be seen that the tasks that handle the front right wheel (suffix `FR`) share core 1 under mapping 1 with three higher priority tasks, which is not the case under mapping 2. This can also be observed on c_1 in Fig. 13, where the two tasks that handle the front right wheel must wait for the execution of the higher priority tasks. No deadline

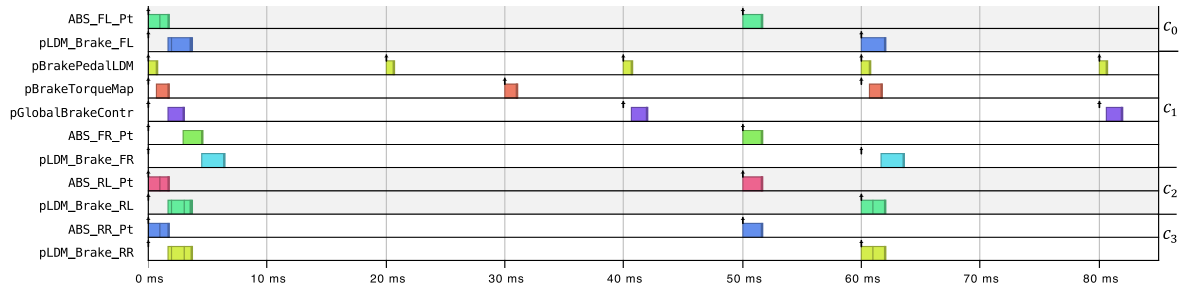


Fig. 13. Execution trace of the first 85 ms under mapping 1. Different cores are highlighted with background shading and annotated on the right side.

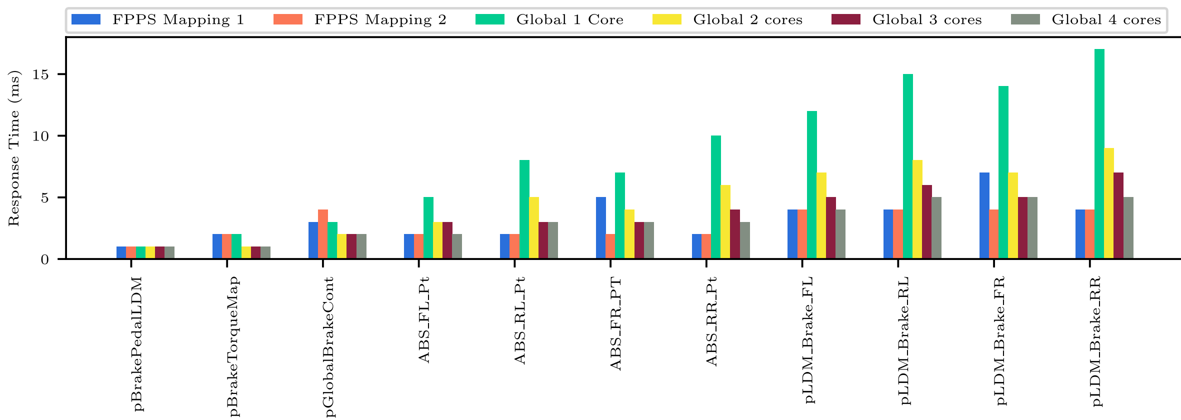


Fig. 14. Recorded maximum response time values for the tasks under both mappings.

misses are observed during the experiment duration. The results of the four settings under global scheduling demonstrate the advantages of additional compute cores, gradually reducing the observed response times. This is especially noticeable for low-priority tasks. The highest priority task `pBrakePedalLDM` experiences the same response time under all evaluated mappings, which is to be expected.

This experiment highlights how the MATERIAL framework can be conveniently used to empirically evaluate the effect of different design choices (e.g., task-to-node and task-to-core allocation) on different metrics (e.g., task response times).

8.5. Execution of the WATERS 2019 application

This section discusses the results of executing different configurations of the WATERS 2019 end-to-end autonomous driving application on the platform. Also here, each configuration is executed on the target platform for 30 s.

8.5.1. Response times under different configurations

Two mappings are examined. In addition to the mapping already described, we investigate the application under global scheduling, i.e., tasks can be executed on all cores of their assigned node. This is realized by adding all cores of the respective node to the tasks' affinity mask.

Fig. 15 presents the results of the experiment. Recorded values are normalized by dividing each response time by the corresponding deadline. From the left side, the first five tasks execute on node 1 and the remaining tasks execute on node 2. All tasks meet their individual deadlines in both scenarios. Differences can be seen for the tasks `EKF` and `Lidar`, which have a larger response time in the partitioned case. This is due to the mapping, which, in the partitioned case, only utilizes three of the four cores, while the global case can execute tasks on all four cores.

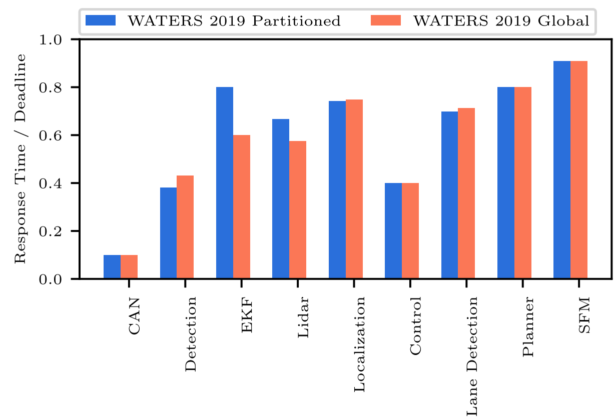


Fig. 15. Recorded maximum response time/deadline of the WATERS 2019 application under partitioned and global scheduling.

8.5.2. Isolating the interference of listener tasks

Listener tasks are activated by the arrival of messages. This can cause interference to the nominal tasks of the application that share the same core. The execution time of the listener task is proportional to the size of the communicated payload. To evaluate the impact, we select the listener task of task `EKF` on node 2 and modify it in a way that allows controlling its execution time. We then examine the response time of task `Lane Detection`, which is mapped to the same core, while the execution time of the listener task is varied in the range $[750, 2000]$ ms, in steps of 250 ms. Two configurations are examined. The first configuration is equivalent to Mapping 3. In the second configuration, we utilize APS partitions to create scheduling reservations to separate 20%

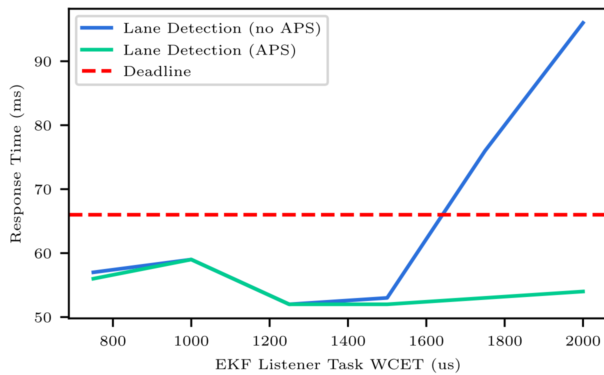


Fig. 16. Observed maximum response time of the task `Lane Detection` under varying execution time of the `EKF` listener task with and without APS reservations. The deadline on task `Lane Detection` is shown by a dashed red line.

of the budget of core 0 for the listener tasks. All remaining budget is used by the application tasks.

Fig. 16 shows the results of the experiment. In the case without APS reservations, the observed response time of the task `Lane Detection` is increasing and eventually exceeds the deadline of the task. In contrast, if the task executes within a APS reservation, guaranteed CPU time is provided to the task and the timeliness of the task is not affected. This demonstrated the importance of the system configuration to guarantee timeliness of the application and the strengths of the MATERIALS framework in supporting QNX's APS reservations.

9. Conclusions

This paper presented the MATERIAL framework for the modeling and automatic code generation of edge real-time applications using the QNX operating system. First, we presented a mathematical formalization of the considered system, which can be leveraged in future work to analyze the real-time behavior of emerging distributed software systems. Later, we also showed the correspondence between the mathematical model and the Amalthea model, which is used more for software engineering and code generation purposes. A code generation tool has been presented, which allows the automatic generation of code from Amalthea models.

We reported the results of an experimental evaluation based on two realistic applications from the automotive domain: a Brake-By-Wire application from a Swedish automotive company, and an End-to-End Autonomous Driving Application as an example for a next-generation embedded system, which was proposed by the industry (Bosch) as the WATERS 2019 industrial challenge. With these case studies, we demonstrate the runtime needed for code generation and the footprint of the resulting applications.

Furthermore, we use both case studies to illustrate the seamless application of our framework to evaluate the performance of diverse design alternatives empirically. This is accomplished using the monitoring capabilities of the framework, which allow the measurement of response times in automatically generated applications under various configurations.

There are many directions for future work. An interesting direction for future research also considers more complex multi-moded applications, in which a task under different modes does not only vary for the execution time requirement but also in terms of inputs, outputs, and task dependencies. Furthermore, a useful new feature to include in the future is the fine-grained modeling of network delays and memory contention to allow modeling these important sources of delays for a wide class of heterogeneous processing platforms. Other directions for future research include further extensions of Amalthea to express the functional parameters affecting the real-time performance of frameworks for artificial intelligence [51,52] and publish/subscribe communication [39,53], as well as to represent memory hierarchies in embedded platforms and hardware accelerators [27].

CRediT authorship contribution statement

Matthias Becker: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Daniel Casini:** Writing – review & editing, Writing – original draft, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] P. Mundhenk, A. Hamann, A. Heyl, D. Ziegenbein, Reliable distributed systems, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, IEEE, 2022, pp. 287–291.
[2] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, T. Azumi, Autoware on board: Enabling autonomous vehicles with embedded systems, in: 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS, 2018, pp. 287–296.
[3] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, G. Buttazzo, A multi-domain software architecture for safe and secure autonomous driving, in: 27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2021, pp. 73–82.
[4] D. Garlan, R. Monroe, D. Wile, Acme: Architectural description of component-based systems, 2000.
[5] A. Bucaioni, M. Becker, Enabling automated integration of architectural languages: An experience report from the automotive domain, J. Syst. Softw. 184 (2022).
[6] Amalthea, Horizon europe project, 2012, URL: https://itea3.org/project/amalthea.html.
[7] AMALTHEA4public project, Horizon europe project, 2012, URL: https://itea3.org/project/amalthea4public.html.
[8] PANORAMA - Boosting design efficiency for heterogeneous, in: Horizon Europe Project, 2012, URL: https://itea3.org/project/panorama.html.
[9] E. Quiñones, S. Royuela, C. Scordino, P. Gai, L.M. Pinho, L. Nogueira, J. Rollo, T. Cucinotta, A. Biondi, A. Hamann, et al., The AMPERE project:: A model-driven development framework for highly parallel and EneRgy-efficient computation supporting multi-criteria optimization, in: 23rd IEEE International Symposium on Real-Time Distributed Computing, ISORC, 2020, pp. 201–206.
[10] D. Dasari, A. Hamann, H. Broede, M. Pressler, D. Ziegenbein, Brief industry paper: Dissecting the QNX adaptive partitioning scheduler, in: 27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2021, pp. 477–480.
[11] A. Mok, X. Feng, D. Chen, Resource partition for real-time systems, in: 7th IEEE Real-Time Technology and Applications Symposium, RTAS, 2001, pp. 75–84.
[12] D. Dasari, M. Becker, D. Casini, T. Blaß, End-to-end analysis of event chains under the QNX adaptive partitioning scheduler, in: 28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2022, pp. 477–480.
[13] N. Borgioli, M. Zini, D. Casini, G. Cicero, A. Biondi, G. Buttazzo, An I/O virtualization framework with I/O-related memory contention control for real-time systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2022).
[14] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, third ed., Springer, 2011.
[15] Blackberry QNX, QNX® neutrino® RTOS - System architecture, 2021.
[16] QNX system architecture guide, QNET section, 2020, URL: https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.sy_arch/topic/qnet.html.
[17] M. Becker, D. Dasari, D. Casini, On the QNX IPC: Assessing predictability for local and distributed real-time systems, in: 29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2023, pp. 289–302.
[18] M. Perrotin, E. Conquet, J. Delange, T. Tsiodras, TASTE: An open-source tool-chain for embedded system and software development, in: Embedded Real Time Software and Systems, ERTS2012, 2012.
[19] P.J. Prisaznuk, ARINC 653 role in integrated modular avionics (IMA), in: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, IEEE, 2008, pp. 1–E.
[20] Á. Horváth, D. Varró, Model-driven development of ARINC 653 configuration tables, in: 29th Digital Avionics Systems Conference, 2010, pp. 6.E.3–1–6.E.3–15, http://dx.doi.org/10.1109/DASC.2010.5655451.
[21] P. Han, Z. Zhai, B. Nielsen, U. Nyman, Model-based optimization of ARINC-653 partition scheduling, Int. J. Softw. Tools Technol. Transfer 23 (5) (2021) 721–740.

[22] P. Plasson, C. Cuomo, G. Gabriel, N. Gauthier, L. Gueguen, L. Malac-Allain, GERICOS: A generic framework for the development of on-board software, in: DASIA 2016-Data Systems in Aerospace, Vol. 736, 2016, p. 39.
[23] P.H. Feiler, B. Lewis, S. Vestal, E. Colbert, An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering, in: IFIP World Computer Congress, TC 2, Springer, 2004, pp. 3–15.
[24] J. Hugues, J.-M. Gauthier, R. Faudou, Integrating AADL and FMI to extend virtual integration capability, 2018, arXiv preprint arXiv:1802.05620.
[25] J. Hugues, B. Zalila, L. Pautet, F. Kordon, From the prototype to the final embedded system using the Ocarina AADL tool suite, ACM Trans. Embed. Comput. Syst. (TECS) 7 (4) (2008) 1–25.
[26] F. Rehm, D. Dasari, A. Hamann, M. Pressler, D. Ziegenbein, J. Seitter, I. Sañudo, N. Capodieci, P. Burgio, M. Bertogna, Performance modeling of heterogeneous HW platforms, Microprocess. Microsyst. 87 (2021).
[27] A. Munera, S. Royuela, M. Pressler, H. Mackamul, D. Ziegenbein, E. Quiñones, Fine-grained adaptive parallelism for automotive systems through AMALTHEA and OpenMP, J. Syst. Archit. 146 (2024).
[28] M. Bambagini, M. Di Natale, A code generation framework for distributed real-time embedded systems, in: 17th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2012, pp. 1–10.
[29] C. Bernardeschi, M. Di Natale, G. Dini, D. Varano, Modeling and generation of secure component communications in AUTOSAR, in: Proceedings of the Symposium on Applied Computing, SAC, 2017, pp. 1473–1480.
[30] F. Cremona, M. Morelli, M. Di Natale, TRES: a modular representation of schedulers, tasks, and messages to control simulations in simulink, in: 30th Annual ACM Symposium on Applied Computing, SAC, 2015, pp. 1940–1947.
[31] G. Wang, M. Di Natale, P.J. Mosterman, A. Sangiovanni-Vincentelli, Automatic code generation for synchronous reactive communication, in: 9th International Conference on Embedded Software and Systems, EMSOFT, 2009, pp. 40–47.
[32] Eclipse APP4MC, 2022, URL: https://www.eclipse.org/app4mc/.
[33] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing Ltd, 2016.
[34] The AUTOSAR standard, 2018, URL: http://www.autosar.org.
[35] F. Cerqueira, A. Gujarati, B.B. Brandenburg, Linux's processor affinity API, refined: Shifting real-time tasks towards higher schedulability, in: 2014 IEEE Real-Time Systems Symposium, RTSS, 2014, pp. 249–259.
[36] Blackberry QNX, Adaptive partitioned scheduler user guide – QNX® software development platform 7.1, 2021.
[37] S. Kramer, D. Ziegenbein, A. Hamann, Real world automotive benchmarks for free, in: 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, WATERS, Vol. 130, 2015.
[38] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, R. Ernst, System level performance analysis - the SymTA/S approach, IEEE Proc. - Comput. Digit. Tech. (2005).
[39] G. Sciangula, D. Casini, A. Biondi, C. Scordino, M. Di Natale, Bounding the data-delivery latency of DDS messages in real-time applications, in: 35th Euromicro Conference on Real-Time Systems, ECRTS, 2023, pp. 9:1–9:26.
[40] T.A. Henzinger, B. Horowitz, C.M. Kirsch, Giotto: A time-triggered language for embedded programming, in: International Workshop on Embedded Software, Springer, 2001, pp. 166–184.
[41] R. Ernst, L. Ahrendts, K.-B. Gemlau, System level LET: Mastering cause-effect chains in distributed systems, in: 44th Annual Conference of the IEEE Industrial Electronics Society, IECON, 2018, pp. 4084–4089.
[42] P. Axer, D. Thiele, R. Ernst, Formal timing analysis of automatic repeat request for switched real-time networks, in: Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES, IEEE, 2014, pp. 78–87.
[43] M. Hassan, R. Pellizzoni, Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (11) (2018) 2323–2336.
[44] International Organization for Standardization, ISO 26262: Road Vehicles : Functional Safety, ISO, 2018.
[45] M. Lowinski, D. Ziegenbein, S. Glesner, Splitting tasks for migrating real-time automotive applications to multi-core ecus, in: 2016 11th IEEE Symposium on Industrial Embedded Systems, SIES, IEEE, 2016, pp. 1–8.
[46] P. Pazzaglia, A. Biondi, M. Di Natale, Optimizing the functional deployment on multicore platforms with logical execution time, in: 2019 IEEE Real-Time Systems Symposium, RTSS, 2019, pp. 207–219.
[47] H. Huang, P. Pillai, K.G. Shin, Improving wait-free algorithms for interprocess communication in embedded real-time systems, in: USENIX Annual Technical Conference, General Track, 2002, pp. 303–316.
[48] A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, M. Bertogna, Waters industrial challenge 2019.
[49] ARM, Cortex-A72: Next generation performance, 2015, Presentation.
[50] N. Fisher, S. Baruah, T.P. Baker, The partitioned scheduling of sporadic tasks according to static-priorities, in: 18th Euromicro Conference on Real-Time Systems, ECRTS, IEEE, 2006, pp. 10–pp.
[51] D. Casini, A. Biondi, G. Buttazzo, Timing isolation and improved scheduling of deep neural networks for real-time systems, Softw. - Pract. Exp. 50 (9) (2020) 1760–1777.
[52] T. Amert, J.H. Anderson, CUPiD RT: Detecting improper GPU usage in real-time applications, in: 24th IEEE International Symposium on Real-Time Distributed Computing, ISORC, IEEE, 2021, pp. 86–95.
[53] E. Shahri, P. Pedreiras, L. Almeida, End-to-end response time analysis for RT-MQTT: Trajectory approach versus holistic approach, in: 19th IEEE International Conference on Factory Communication Systems, WFCS, 2023, pp. 1–8.