Performance modeling of Heterogeneous HW Platforms

Falk Rehm[a], Dakshina Dasari[a], Arne Hamann[a], Michael Pressler[a], Dirk Ziegenbein[a], Jörg Seitter[a], Ignacio Sañudo[b], Nicola Capodieci[b], Paolo Burgio[b], Marko Bertogna[b]

[a]*Robert Bosch GmbH*
[b]*University of Modena and Reggio Emilia*

Abstract

The push towards automated and connected driving functionalities mandates the use of heterogeneous hardware platforms in order to provide the required computational resources. For these platforms, established methods for performance modeling in industry are no longer effective or adequate. In this paper, we explore the detailed problem of mapping a prototypical autonomous driving application on a Nvidia Tegra X2 platform while considering different constraints of the application, including end-to-end latencies of event chains spanning CPU and GPU boundaries. With the given use-case and platform, we propose modeling concepts in Amalthea, capturing the architectural aspects of heterogeneous platforms and also the execution structure of the application. These models can be fed into appropriate tools to predict performance properties. We proposed the above problem in the Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) Industrial Challenge 2019 and in response, academicians came up with different solutions. In this paper, we evaluate these different solutions and summarize all approaches. The lesson learned from this challenge is then used to improve on the simplifying assumptions we made in our original formulation and discuss future modeling extensions.

Email addresses: `falk.rehm@de.bosch.com` (Falk Rehm), `dakshina.dasari@de.bosch.com` (Dakshina Dasari), `arne.hamann@de.bosch.com` (Arne Hamann), `michael.pressler@de.bosch.com` (Michael Pressler), `dirk.ziegenbein@de.bosch.com` (Dirk Ziegenbein), `joerg.seitter@de.bosch.com` (Jörg Seitter), `ignacio.sanudoolmedo@unimore.it` (Ignacio Sañudo), `nicola.capodieci@unimore.it` (Nicola Capodieci), `paolo.burgio@unimore.it` (Paolo Burgio), `marko.bertogna@unimore.it` (Marko Bertogna)

1. Introduction

Emerging automated and assisted driving applications are radically redefining the way automotive systems are designed and deployed. The computational power and communication bandwidth required by these new functions (such as path planning, object recognition, predictive cruise control), exceed the capabilities of legacy micro-controller based compute nodes and the newer E/E architectures are shifting towards 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), reconfigurable logic (e.g., FPGA) and dedicated application-specific integrated circuits (ASIC). Systems designers thereby need to grapple with conflicting demands of these applications like predictability and efficiency, while dealing with the intricacies of more complex hardware platforms and complex software-hardware interactions between different compute elements. Mastering the system complexity has become increasingly difficult and there is an increasing need for modelling and analysis tools to meet these challenges.

Although much research has been conducted on the formal analysis of heterogeneous multiprocessor architectures, we believe there are some gaps that need to be addressed primarily on the modeling perspective. Such gaps are represented by scheduling complex tasksets into multi-core CPUs that work in concert with compute accelerators. Existing response time analyses must be therefore extended to consider the added complexity of such heterogenous platforms, with specific emphasis on memory contention, programming models and different scheduling policies enabled by the different computing elements.

In order to master such a complexity an important prerequisite is the need for expressive *performance models* capturing the heterogeneity of the hardware-software system. These performance models can be fed into simulation tools to replicate different execution scenarios and predict the performance at design time. To our knowledge there do not exist models capturing the intricacies of heterogeneous systems in the current state of the art.

With this work, we make the following key contributions:

- We highlight the need for expressive performance models capturing the heterogeneity of hardware-software systems. As a use case, we con-

sider an autonomous driving application that is deployed in a modern heterogeneous SoC (an Nvidia Tegra platform).

- We provide a detailed description of an autonomous driving application designed to run in a modern heterogeneous SoC and in particular focus on the problem of mapping such an application to the underlying platform such that its timing requirements are guaranteed.

- We demonstrate how a heterogeneous application can be modelled in Amalthea [1] by capturing both the hardware and software aspects. In order to model the hardware, we describe the architecture of the underlying platform and show how key information needed for a timing analysis can be captured in Amalthea. Similarly, on the software front, we discuss the different modes of software interactions occur between the CPU and the GPU and describe how these interactions can be captured in Amalthea.

- We summarize and discuss the proposed solutions for the timing analysis challenge regarding this use case, posed by us at the WATERS Industrial Challenge 2019, held in conjunction with the 31st Euromicro Conference on Real-Time Systems [2, 3].

- We eventually discuss the lessons learned by comparing how different academicians approached such a challenge: more specifically, we propose modeling extensions in order to capture software and hardware in greater detail. These extensions include i) a more detailed modeling of the GPU hardware ii) refined memory access modeling iii) extended support for practical schedulers like the QNX Adaptive Partitioning Scheduler iv) extensions to handle middleware systems using the publish-subscribe communication paradigm. This enhanced modeling approaches will be used to formulate a future version of the challenge.

1.1. Organization of the work

The remainder of the article is organized as follows. The following section presents the Amalthea modelling framework and related work, focusing on similar challenges proposed in the community and other modelling frameworks that are used for modelling real-time constraints. Section 3 describes

the automotive use case and timing constraints. In Section 4 and 5 we provide hardware details of the Jetson TX2 platform and an overview of GPU programming concepts. A description of the 2019 Waters challenge is then presented in Section 6. Then, in Section 7 we present the modelling mechanisms that are needed to address the challenge and the modelling artifacts used to describe the use case from a hardware and software perspective. Section 8 presents a summary of all the proposed solutions and a comparison between all the different solutions. Then in Section 9, we provide multiple modelling extensions that lay the basis of a new Industrial Challenge. Finally, Section 10 presents our conclusions and concludes with a summary of the provided contributions.

2. Need for Modelling

Model-based techniques are integral in mastering the complexity of heterogeneous systems. Models capturing the software and hardware architecture are essential in the early design phases for several reasons, including

- (Early) system performance evaluation before target hardware is available
- (Glue) code generation for system configuration parameters like scheduling configurations and communication infrastructure
- Design space exploration (what-if analysis) before implementation
- Data exchange between costumers, suppliers and system integrators as well as singe input for multiple (commercial) analysis tools

As systems get more complex in the automotive arena, it becomes imperative to be able to have rich and expressive models of the hardware, software and design decisions and incorporate it into the design process, as early as possible. In this work, we used Amalthea[1] for specifying the characteristics of our system.

2.1. Amalthea Modelling Concepts

Amalthea is an open source data model which focuses on the non-functional dynamic system architecture of a hardware/software system enriched with performance data embedded in a rich tooling platform [1]. The model is

primarily used for performance analysis and performance simulation. By abstracting from the specific functionality of model elements, Amalthea models or sub-models can be exchanged between companies or with academia without disclosing confidential information. Amalthea is capable to express complete hardware/software models for multi- and many-core systems and is currently extended to cover the increasing heterogeneity of modern embedded systems.

The granularity of the modeled system always depends on the use case, which means a performance simulation including memory accesses needs way more system model information compared to an abstract event chain analysis where only runnable execution times are used. In turn, this means that based on the specific use case only parts of the system model may be required. Amalthea is organized in a set of sub models which closely interact witch each other, as an example the mapping model maps task of the software model to schedulers of the operating system model which are in turn mapped to processing units of the hardware model. An Amalthea model typically consists of the following sub models:

- *Software:* Represents the application in terms of runnable entities (in our case study described in section 3, including the GPU offloading mechanisms).

- *Hardware:* Represents the HW platform in terms of processing elements, memories and interconnects (in our case study a simple representation of the Nvidia Jetson TX2 platform including latencies).

- *Operating Systems:* Defines the operating systems together with the schedulers and the scheduling algorithms.

- *Stimuli:* Periodic and interprocess triggers for the software model.

- *Constraints:* In this part of the system model can deadlines for different software tasks be specified and other system constraints.

- *Mapping:* Connect the software, operating system and the hardware model for a specific task mapping and assigns data elements (labels) to memory.

A complete documentation about the Amalthea version 0.9.3 [1] can be found on the official webpage of the open source project Eclipse APP4MC [1]. Examples for modeling multi-core microprocessor-based systems with Amalthea are given in [14] and [15]. However as we shall describe in the upcoming sections, there were gaps in Amalthea in the context of expressing software interactions like different offloading computations from the CPU to the GPU, which have been covered by the described in Section 7.

2.2. Related work

In this section, we will firstly cover different modeling tools and secondly other industrial challenges that have been proposed and what exactly differentiates the described WATERS industrial challenge in this paper.

One of the most fundamental problems when proposing real industrial settings is the hardware and software representation of the system. Many different frameworks for modelling real-time systems have been used and proposed in the industry and scientific community. Capella [10] is a modeling workbench to master architectural design and manage complexity. However it lacks details on the dynamic architecture and performance aspects of embedded systems. Unlike Amalthea that is focused on performance modelling of the dynamic system architecture, Capella is meant to capture the static system architecture. MAST (Modelling and Analysis Suite for Real Time Applications) [11], is an open source set of tools that enables engineers developing real-time applications to check the timing behaviour of their application, including schedulability analysis for checking hard timing requirements. Commercial simulation tools like Vectors Timing Architect [9] and InChron ChronSIM [12] provide a dedicated model-based simulation and optimization tools for real-time systems. They are used for the simulation and performance requirements validation n of embedded systems. Similarly Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [8], introduces a domain view for time modeling and defines standard UML elements to express defined timing concepts of real-time and embedded systems. But MARTE highly abstracts from the hardware architecture. All the aforementioned tools (including the earlier version of Amalthea) are not particularly adapted to handle heterogeneous architectures and capture the

[1]There are newer versions of Amalthea available, 0.9.3 refers to the version in which most of the changes were implemented

semantics of CPU-GPU interactions involving modern parallel frameworks like CUDA or OpenMP. In this work, propose extensions in Amalthea in Section 7 and Section 9 to handle the requirements of modern heterogeneous applications.

2.3. Other Industrial Challenges in the Real-time systems domain

The 'Formal Methods for Timing Verification' FMTV industrial challenge, proposed in conjunction with the European Conference on Real-Time Systems (ECRTS) has a long, well-known history in the real-time community. The first challenge was proposed in 2014 and 2015 by Thales Research [5]. The challenge consisted in the end-to-end latency analysis and priority assignment optimization of an aerial video tracking application. In 2016, Robert Bosch GmbH proposed the challenge [6] focused on determining end-to-end latency bounds for different cause-effect chains and response time of the tasks of a engine management automotive software. An Amalthea model containing the software and hardware model of the system was provided. Then in 2017, Robert Bosch GmbH extended the problem by proposing the analysis with two different inter-task communication semantics, namely, Implicit Communication and Logical Execution Time [7]. The model follows the same hardware and software specification of the 2016 model. In 2018, Dassault Aviation took up the challenge [2]. The challenge focused on end-to-end timing analysis and compositionality of a drone-like muti-system case study. The organizers provided a SysML architectural model and the generated C source code for the participants interested in performing WCET estimation. Then, in 2019, Robert Bosch GmbH and HiPeRT Lab (University of Modena and Reggio Emilia) proposed the challenge that is subject of this work [2].

Recently, PerceptIn, have proposed an industrial challenge in the context of the 42nd IEEE Real-Time Systems Symposium (RTSS) conference [3]. In this industrial challenge, it is presented an end-to-end latency and schedulability analysis problem that must be applied on an autonomous driving use case. However, in the proposed case study, the task-to-engine mapping problem is neither introduced nor task timing constraints like WCET or deadlines are provided. Such missing elements, make the challenge and solutions presented in this paper more concrete and complete from a scientific point of view.

---

[2]https://github.com/AdaCore/RESSAC_Use_Case/

[3]`http://2021.rtss.org/industry-session/`

Also we re-iterate that the WATERS challenge on which this paper is based, serves to reiterate the need for modeling, the challenges in capturing the software-hardware interactions in modern heterogeneous applications spanning general purpose CPUs and also custom accelerators (GPUs).

In the next section we present the autonomous driving use case and the timing constraints of all the software components. Each of these tasks have been modeled in AMALTHEA, they form the basis of the partitioning and end-to-end response time analysis.

3. Application Use Case

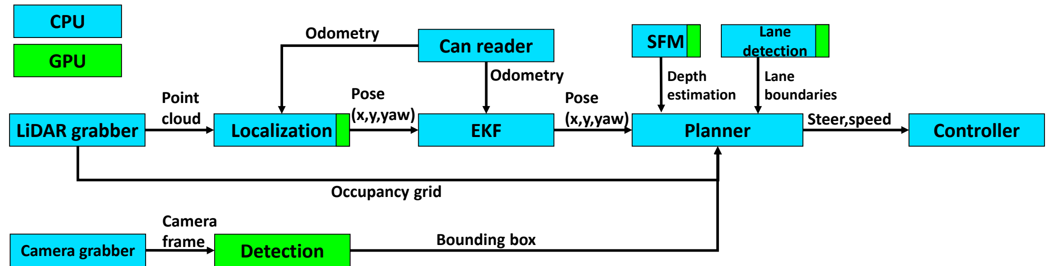


Figure 1: Autonomous driving task pipeline

Researchers in next-generation embedded systems agree that there is a clear need and urgency to demonstrate and bound the predictability of complex heterogeneous systems in terms of end-to-end latencies. As a prerequisite, there is a need to be able to formally model these systems (software and hardware), which can then be fed into appropriate tools for performance analysis. In this section, we first describe the features of a relevant application and later we describe how such an application can be modeled.

We developed the prototype of an end-to-end autonomous driving application running on the Tegra TX2 platform. The goal was to create and model an application that is representative for next-generation Advanced Driver Assistance Systems (ADAS) and automated driving systems. It uses state-of-the-art algorithms from robotics and automotive, mixes different workloads with different criticality levels on the same computing platform, and exploits architectural heterogeneity by concurrently running tasks on the most suitable computational units (i.e. CPU and GPU).

On the highest level, the reference application provides the proper throttle, steering, and brake signals to drive a vehicle through a predetermined

map of way-points. Moreover, the vehicle runs specific high-priority tasks to break and/or perform emergency manoeuvres to avoid obstacles like pedestrians or bicycles, the so-called Vulnerable Road Users (VRUs). It is important to mention that the application works without using Robotic Operating System (ROS), and all algorithms are implemented from scratch.

An overview of the key software tasks running in the system can be seen in Figure 1. The green boxes with the dashed borders represent tasks executing on the GPU, while the blue ones execute on the CPU hosts. Please note that there are tasks that can be partitioned by using either a GPU or CPU implementation of it, e.g., lane detection or localization.

In the following a short description of the individual tasks is given:

LIDAR Grabber. The grabber reads the information from the LiDAR sensor, and builds a point cloud that is shared with the *Localization* task. Additionally, a so-called occupancy grid is produced capturing the free space in front of the vehicle. This information is sent to the *Path Planner* task.

Localization (CPU/GPU). The localization is implemented using a *Particle Filter* algorithm. This method adopts a probabilistic model to determine the position of the car on the given environmental map. This position information is then merged with the motion estimation coming from the vehicle odometry, to obtain an accurate and predictive estimate of the vehicle's position. The current pose of the vehicle is sent to the *EKF* task.

CANbus polling. This task snoops the key vehicle information (steer, wheel, brake, acceleration status, etc) from the on-board CAN bus and sends it to the *Localization*, *Planner*, and *EKF* tasks.

Path Planner. The main purpose of this component is to calculate and follow a vehicle trajectory. This trajectory is represented as a spline defining the sequence of positions and orientations that the vehicle shall follow. The spline can be enriched with additional information such as speed to hold, stops, etc. The *Planner* task sends the goal state of the vehicle (i.e., target steer and speed) to the *DASM* task that is in charge of writing the actuation commands to the CAN bus.

Car Controller + Can Writing. This task computes and establishes the speed and steer that must be effectively employed from the information that is provided by the *Path Planner* task. In order to implement this functionality two different controllers are used:

- A *Pure Pursuit controller* calculates the steering angle to follow the given trajectory.

- A *PID controller* to follow the speed profile defined by the given trajectory.

The calculated commands are sent via CAN bus to the engine control unit in order to perform the final actuation.

EKF. The *Extended Kalman Filter* is the nonlinear version of the Kalman filter method. This algorithm estimates the poses of the ego vehicle and objects repeating two stages: prediction and correction. The data produced is a matrix that corresponds to the object and ego vehicle estimation which is sent to the *Path Planner*.

Lane detection (CPU/GPU). This task provides accurate locations of the road boundaries and the shape of each lane. The output of this task is a matrix of points representing the lane boundaries within the road, which is sent to the *Planner* task.

Detection (GPU). This task is responsible for detecting and classifying the objects on the road. It uses a machine learning approach, with an optimized version of YOLOv3 [26] for the TX2 platform that exploits CUDNN and TENSOR-RT [4]. The neural network was re-trained in order to extend the classification classes of the baseline version. The following object categories are supported: pedestrians, cars, trucks, buses, motorbikes, bicycles, riders, traffic lights, and traffic signs. All the objects detected are visualized and the information produced is sent to the *Planner* task.

Structure-From-Motion (SFM) (CPU/GPU). Structure From Motion is a method for estimating 3D structures (depth) from vehicle motion and sequences of 2D images. This task returns a matrix of points representing the distance to objects in the image, and sends it to the *Planner* task.

Please note that each task shown in Figure 1 is characterized by period, execution cycles, and a deadline which is in our case equal to the period. Communication dependencies between tasks have sampling semantics, meaning that each task is independently activated once per period and its output

[4] https://github.com/ceccocats/tkDNN

Task	PU	Avg. Ticks	Min Ticks	Max Ticks	Period(ms)
Lid_Grab	A57, (Dvr)	23520000	20320000	27320000	33
CAN	A57, (Dvr)	999360	799360	1199360	10
EKF	A57, (Dvr)	8799340	7959340	9519340	15
Planner	A57, (Dvr)	22743822	19243822	26483822	15
Car_Control	A57, (Dvr)	3219990	2599990	3719990	5
SFM	A57, (GPU, Dvr)	53775310	48274300	59003000	33
Localizn.	A57, (GPU, Dvr)	754439355	733039355	774839355	400
Pre_Det	A57, (Dvr)	7178560	6573410	7951921	200
Det	GPU	165000000	162000000	174000000	200*
Post_Det	A57, (Dvr)	7561630	6999284	8513680	200*
Lane_Det	A57, (GPU, Dvr)	98689120	95689120	102089120	66
OS_Ovhd	A57, (Dvr)	100000000	100000000	100000000	100

Table 1: Task Characterization. Here PU refers to the processing unit, Min and Max Ticks refer to the lower and upper bound of the ticks. Dvr is short for Denver. The Detection task is split in three tasks due to the offloading to the GPU. The three tasks are chained via inter process triggers. Det refers to detection, Ovhd is abbreviation for overhead,Localizn. is short for localization and Lid_Grab refers to the Lidar Grabber task

data is placed into an intermediate buffer (of size 1) such that the receiving task always works on the most recently produced data. The core execution time is of course mapping dependent. Table 1 shows the different tasks of the provided model with the different mapping targets. Additional to the mentioned tasks there is also an additional task to represent the OS overhead. The measured tick values are always related to A57 except for the *Detection* because this can only be executed on the GPU, therefore there are also two additional tasks for pre- and post-processing. All necessary information e.g. tick values for Denver is specified in the model. Further information about the modeling can be found in the following Section 7 and the concrete Amalthea model with all necessary information in [2] .

4. Overview of the Nvidia Tegra X2 Platform

We use the Jetson TX2 platform from Nvidia [13] as an example heterogeneous embedded board as the modelled HW in which the target application will run. In this section we introduce the details of the hardware aspects of the Nvidia TX2 platform needed to reason about the performance. At a high-level abstraction, embedded heterogeneous SoCs (System on Chip) featuring GPGPU accelerators are characterized by the following hardware components: 1) Host Subsystem 2) Accelerator and 3) Memory hierarchy as seen Figure 2.

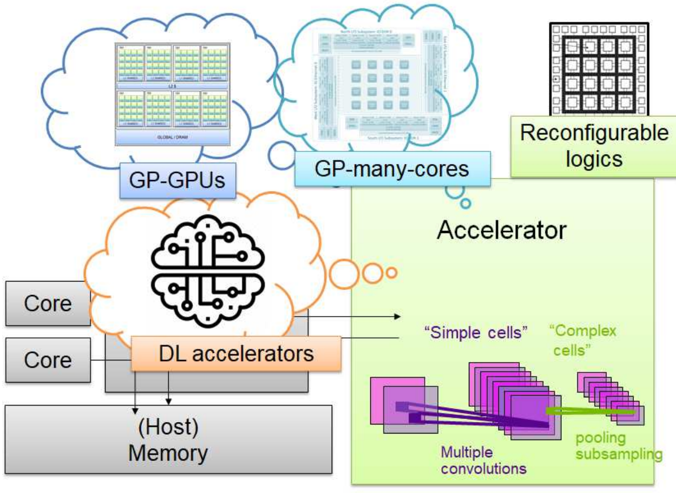


Figure 2: Multi-heterogeneous hardware platform

Host Subsystem. As shown in Figure 3, the Nvidia Tegra X2 is composed of two different CPU islands, a quad-core 1.9GHz ARMv8 A57 and a dual-core 2GHz ARMv8 Denver (NVIDIA ARM-based proprietary technology). Each of the cores in the A57 subsystem integrates a private 32KB L1 data cache and a 48KB L1 instruction cache, while the Denver features a per-core 32KB L1 data cache and 48KB L1 instruction cache. Moreover, each of the islands has a 2MB L2 cache.

Accelerator. The platform features an integrated GPU or iGPU Pascal-based architecture "GP10b" with 256 CUDA cores grouped within two streaming multiprocessors (SM). Each of the SM has a 64KB L1 cache, and both SMs share a 512KB L2 cache. The GPU integrates two major hardware components: an Execution Engine (EE) responsible for performing the parallel workload execution and a Copy Engine (CE) responsible for high bandwidth memory transfers.

Memory. In embedded devices, for example NVIDIA Tegra based boards, the GPU and the CPU have access to a common shared memory. Allocations in global memory are managed by the CPU host. The GPU is organized as an array of Streaming Multiprocessors (SM) where the SMs share a common L2

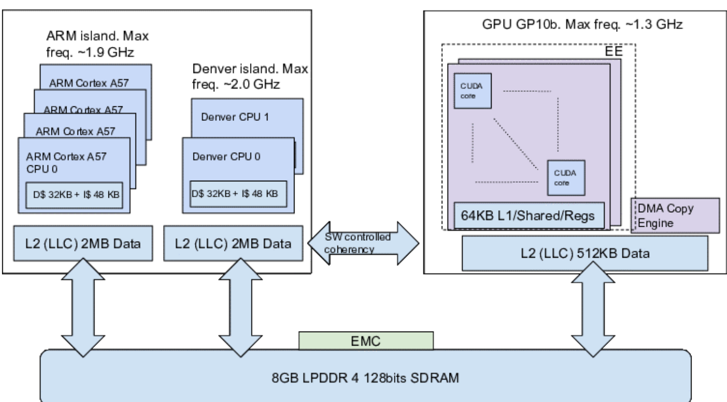


Figure 3: Nvidia Tegra X2 hardware architecture

cache. With regard to memory, it is important to consider the bandthwidth and how to allocate buffers to be accessed by CPU and GPU.

Memory bandwidth: The Tegra X2 SoC features a 8GB LPDDR4 128 bit DRAM with a bandwidth of 58 GB/s. The Denver cores have approximately three times the memory bandwidth of the ARM-A57 cores, i.e. approximately 14 vs 4.5 GB/s, hence memory accesses from the Denver cores have shorter latencies for a single read operation throughout L2 and system RAM accesses. The GPU can access the shared RAM banks with a bandwidth of 20GB/s.

Generally, the CUDA programming model allows the user to handle memory (allocation and data transfer) in many ways. GPU-side allocations might involve data buffers that are pageable or pinned [16], or data can be simply shared by both CPU and GPU (i.e. Unified Memory Models, UVM). We argue that explicit copies in real-time applications are to be preferred to unified memory models [34], as the latter is more difficult to benchmark and forces the analyses to account for hidden coherency mechanisms between CPU and GPU address spaces.

As shown in Section 6, most of the issues in performance and predictability in such complex hardware connect to concurrently accessing the shared system resources. For this reason, the key challenge, and nearly most of the proposed solutions, partly or completely focus on the memory hierarchy, both on the host and accelerator complex (e.g., shared LLCs), and on the integrated DRAM banks. Also, the hierarchical structure of GPGPUs compute engines, where CUDA cores are clustered into SMs, is a challenging aspect,

mostly because the task scheduling strategy is somehow hidden inside the (opaque) CUDA drivers. Nonetheless, one challenge tries an approach based on a supply bound function, to model this aspect.

5. GPU Programming Concepts

A GPU function, also called command or kernel, refers to a C++ function that is executed on the GPU. An application may be realized with one or more GPU kernels in addition to regular CPU functions. A stream is a queue of both compute and copy operations issued by an application [17]. Each stream can therefore be composed of multiple kernels and copy operations. An application can delegate its kernels and copy operations across multiple streams; typically kernels that can run in parallel are assigned to different streams. Kernels within a stream are executed in FIFO order whereas kernels from different (custom) streams within the same application can execute concurrently. Concurrency and parallelism are subject to the availability of enough resources on the GPU [45]. Note that the above description regarding streams is Nvidia specific, although the concept of streams can also be found in other programming languages like OpenCL where it is referred to as "command queues".

Streams can be categorized as DEFAULT streams or custom streams. When unspecified, all kernels are executed in the DEFAULT stream.

Streaming Multiprocessor (SM) resource occupancy relates to the number of threads per block, per-thread registers footprint and allocated shared memory. Kernels dispatched from the same stream are implicitly synchronized, whereas kernels in different streams must be synchronized with specific CUDA artifacts (stream events). Potentially, a kernel on a stream can utilize all the CUDA cores of all the SMs of the GPU. CUDA allows the user to designate a stream to have either a high priority or a lower priority.

5.1. GPU Scheduling Model

A GPU-based application is a process that creates a so-called GPU-context. For instance, two GPU applications belong to two different GPU contexts reside in different memory spaces, following a standard UNIX-like process execution scheme. Each application is associated with one (or more) channels [18]. Only a channel associated with one application can execute on the GPU at any given time. So, two applications cannot co-execute at the same time, they may however have interleaved executions.

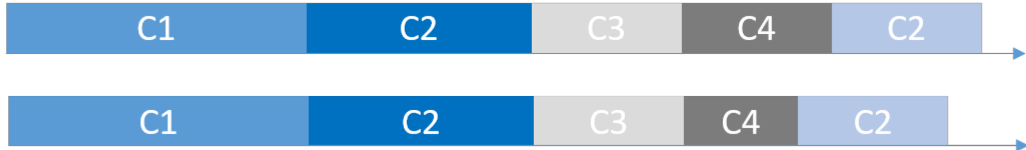


Figure 4: Illustration of channel scheduling with time slice allocation to channels in a run-list. Notice in the lower diagram, how channel C2 is scheduled if C4 finishes earlier than its time slot

The timeslice within a channel could be used to execute the copy or the execution phases. When the channel has multiple streams, the copy and execute phases may run in parallel as seen in Figure 5, depending on availability of resources.

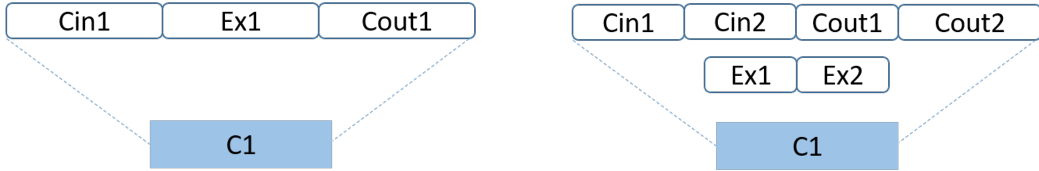


Figure 5: Parallelism with multiple streams. The figures on the left and right illustrate the time distribution within C1's timeslice when C1 has one or two streams respectively

Once a channel is selected, the GPU scheduler dispatches kernels of the selected channel for execution to the SMs (or copy phases to the copy engine). The GPU Host dispatches commands to the copy and/or execution engine. By the time a kernel command arrives to the execution queue, commands are translated into CUDA programming artifacts called thread blocks that are allocated into SMs. Thread blocks, which in turn are composed of warps, are managed by the Warp Scheduler. The warp scheduler organizes ready-to-execute instructions from a set of available and ready warps. By the time a kernel command arrives to the SM, the Warp Scheduler assigns instructions to warps.

5.2. Thread block-to-SM mapping

The thread block scheduler is in charge of assigning thread blocks to the SMs. This distribution is transparent to the user and it plays a key role since it is the component that decides how applications effectively runs on

the GPU cores. To accomplish the thread block-to-SM allocation, the scheduler performs an occupancy test. In this test, the condition of each SM is examined to determine its current degree of resource utilization. Specifically, the test returns whether the current occupancy is such that a new block can be allocated into the target SM. The metrics considered by the scheduler are the following resource occupancy factors (i) number of threads/warps per thread block(ii) shared memory per thread block and (iii) number of registers per warp.

If kernels are dispatched onto a single stream, then, kernels are launched sequentially and thread blocks are equally distributed through all the available SMs, first to the even numbered SMs and then to the odd numbered SMs in increasing ID order. If kernels are however dispatched onto different streams, multiple kernels can execute concurrently. More details can be found in [45], in this paper authors present a model describing how the thread block scheduler considers the occupancy metrics of each kernel to perform the block-to-SM mapping.

5.3. GPU-CPU Interaction

In a typical GPU-CPU interaction, when a program starts, the CPU copies the data and instructions from host memory domain to GPU memory, this copy is performed using the Copy Engine (CE) as seen in Figure 6. After this operation, the CPU launches one or more kernels: this execution can be either synchronous or asynchronous. If synchronous, the CPU dispatches the work to the GPU and then the CPU waits until the end of the kernel execution. In the asynchronous case, it is possible to perform computations on the CPU, immediately after dispatching the work to the GPU without waiting for completion, in a work-conserving manner.

Note: Although copy engines are NVIDIA terminology, asynchronous executions and copies on a separate device are a generic abstraction, required in any host-accelerator programming model.

After having described the application use-case, the hardware model and the programming model, we next describe the WATERs challenge.

6. Problem Description: The WATERS 2019 [21] challenge

The *Industrial Challenge* of the WATERS workshop 2019 [2] was conducted in conjunction with the ECRTS [20] conference with the key objective to share ideas, experiences and solutions to concrete problems arising

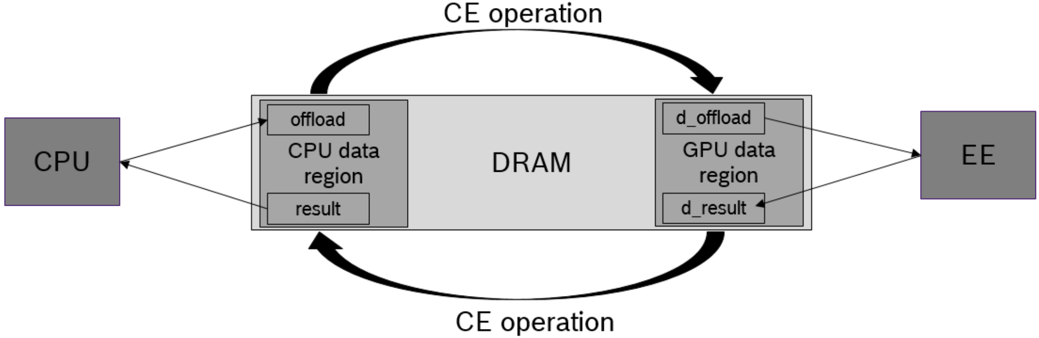


Figure 6: Data transfer handled by the copy engine

from real industrial case studies. The key intention of the challenge was to firstly model a real-world application spanning heterogeneous compute nodes and demonstrate how this model could be used to solve an associated timing analysis problem. The users were presented with an Amalthea model [21], which captured details of the application use-case, programming model and the hardware model.

6.1. Description of the Challenge

Given the above hardware platform and the application use-case, the challenge proposed two main problems: response time computation and task mapping.

Response Time Computation. Given the autonomous application described above consisting of a set of dependent tasks and a given mapping, the problem is to calculate its end-to-end response time [49, 48]. The response time should account for the time for the copy engine to transfer data between the CPU and the GPU. It should also account for the time for the data transfers between the CPU and the shared main memory considering a read/execute/write semantic. Note that while memory accesses are already accounted within ticks of GPU tasks, CPU task response times must account memory contention caused by tasks running on different processing units that use different memory controller clients and/or the GPU's copy engine, and memory access latencies in addition to ticks and preemption.

Optionally different offloading mechanisms (synchronous/asynchronous) may be considered to further optimize the end-to-end latencies. The response

time analysis should also be memory contention aware. A model must be devised to account for the memory interference when multiple CPU cores and the GPU copy engine is accessing memory at the same time.

Mapping Challenge. Some tasks can either be mapped to different type of CPUs (A57 or Denver Cores) or a GPU, defining a design space for exploration. Find mappings that (Pareto-) optimize the latency of the different task chains.

We made the following assumptions to better define the scope of the problem.

On the CPU side. CPU tasks follow the read, execute and write semantics similar to the implicit communication of AUTOSAR [22]. A CPU task offloads workload synchronously or asynchronously to the GPU. For the CPU, priorities have to be set so as to meet deadlines. We assume that the task period is equal to its deadline.

On the GPU side. We assume that applications do not use unified/paged memory but only pinned memory. We consider that each application has one custom stream associated with it. Furthermore, we assume that there is a single copy engine to carry out host-to-device and device-to-host memory transfers. We assume a channel per context (application), so as to have one time slice per application. Different applications might have different time slices. We assume no fine granular modeling of memory accesses from the execution engine. Memory accesses from the EE stage are not explicitly modeled (memory accesses are included in ticks, because copy engine also make use of Level 2 Cache).

In general for the CPU and GPU, for simplicity, we assume that cache lines are not evicted and tasks access labels only once per activation, from memory. Although caches are not explicitly modeled, we assume that data is always transferred as a complete cache line (64 bytes).

6.2. Memory Contention Model

For the memory contention model, we refer to the work in [34, 35]. The idea is that the length of memory phases (read and write) during contention, depends on how many other memory controller clients are accessing main memory at the same time. The model accounts for increasing latencies for GPU cpoy engine activity during the observed time window. The method to compute memory contention for CPU and GPU tasks is presented below.

Modeling Memory Contention for CPU Tasks. Let $\pi \in \{ Denver, A57\}$ refer to the CPU type and K_π refer to the increase in memory access latency of a task running on the observed core π when interfered by one external core. It does not matter if the interfering core is Denver or A57: this number only depends on the observed CPU core; so $K_{A57} = 20ns$ and $K_{Denver} = 2ns$.

For each CPU type π, we also define a baseline latency B_π which is the time taken to read or write a cache-line of 64 bytes from main memory to the CPU registers in isolation (without any interference). From empirical measurements, $B_{A57} = 20ns$ and $B_{Denver} = 8ns$.

Let n_C refer to the number of interfering cores which ranges from 0 to 5 (as one of the 6 cores is the observed CPU core. An n_C of 0 implies no interference from other CPUs).

Since we also need to consider the sensitivity to the GPU copy engine activity, we introduce the term δ_π which represents an increase in memory access latencies of a task running on the CPU type π if the GPU is performing operations on the copy engine. Here, $\delta_{A57} = 100ns$ and $\delta_{Denver} = 20ns$. Also let bG denote a boolean value which is 1 when the GPU operates the copy engine, else 0.

Then the Latency L_π is defined as the time to read/write a cache-line (here assumed to be 64 bytes) from main memory to the cpu registers of CPU type π when interfering with n_C cores is given by

$$L_\pi = B_\pi + K_\pi * n_C + \delta_\pi * bG$$

So the first term accounts for the basic time (without contention) while the second and third terms account for the interference from other CPUs and the GPU.

Modeling Memory Contention for GPU Tasks. From the GPU side, we only consider interference from the copy engine data movements. A GPU CE data movement is a 100% memory bound runnable. Every memory access is modeled as sequential access. Then the Latency L_{CE} incurred by a GPU operation when contending with n_C other cores on the shared memory is given by

$$L_{CE} = G + 0.5 * n_C$$

with L_{CE} is defined as the time taken to transfer 64B using the copy engine (cudaMemcpy) and the GPU baseline G is the time taken to transfer 64 bytes using the copy engine with no interfering CPUs. Here $G = 3ns$.

We acknowledge that the model is rather simple and static in nature on multiple accounts; it does not take into consideration the actual memory traffic generated by the contending cores and makes not assumptions on the interconnect arbitration mechanism. Also memory controllers have optimization mechanisms that could speed up the access latencies. However, we believe this is a good starting point for the discussion on the contention effects.

The Amalthea model representing the hardware-software system described in this paper along with a description of the challenge formulation can be found in the WATERS community forum [21]. The challenge was also published later in [3].

7. Amalthea Modeling Concepts

This chapter focuses on the modeling mechanisms which are necessary to model a system in Amalthea which is capable to represent the ADAS application described in section 3 on a heterogeneous platform. In the following subsections, different modeling concepts in Amalthea are explained. We use existing mechanisms of Amalthea to represent the offloading of a task from a CPU to a GPU. For describing the system model, we use the Amalthea meta-model version 0.9.3.

7.1. CPU Memory Access Semantics

As mentioned we assume that tasks executing on a CPU *(in case ot the TX2 platform a A57 or Denver core)* follow a read/execute/write semantic. In this model, the execution is decoupled into three different phases: a read phase where the data is fetched from memory, an execution phase, where the task performs pure computation and a write phase, when the core writes the computed result back to memory. Memory accesses are usually represented as label accesses in Amalthea (within a runnable). The size of the label as well as the operation itself (read or write), are specified in the software model. The mapping to memory is specified in the mapping model, while the access latencies/data rates and access path can be specified in the hardware model. Figure 7 shows an example for a Runnable within a Task following the read (label access to l1) → execute (Ticks) → write (label access to offload1) policy.

Then the total execution time of a task is the sum of the

1. Label Read access time

Workload1 [Runnable]
Runnable Items
read l1 [LabelAccess]
Ticks [Ticks]
default: DiscreteValue Constant (value: 5000) [DiscreteValueConstant]
write offload1 [LabelAccess]

Figure 7: Runnable example for a CPU

2. Core execution time
3. Label Write access time

F is the frequency of the executing processing unit (CPU/GPU). The core execution time T is calculated as:

$$T = Ticks/F$$

The label read access time R and write access time W is a function of the label size L, read latency RL or write latency WL, the frequency of the processing unit and the cache line size S. The label read access time R is calculated as follows:

$$R = (L/S) * (RL/F)$$

The write access time is computed similarly. Note that due to simplicity, read and write latencies are always the same (we ignore store buffers, etc.). The read and write latencies are specified for every processing unit. In the model it can be found under *Processing Unit* $\rightarrow$ *Access Element*. Using different read and write latencies and considering the write backs of a cache could also be future extensions.

7.2. GPU Modeling

A GPU can be represented by a processing unit within the Amalthea hardware model. Due to the assumption that every application consists of one stream, it is not necessary to separate the copy engine from the execution engine as this essentially implies that interleaving between copy and execution phases cannot occur. For the abstract modeling of a GPU, we do not model the internal architecture like thread blocks, or the internal memory hierarchy explicitly. Due to its complexity, we also abstract the GPU cache

system, this means that the explicit copy operations are executed (see Section 7.3) but all data accesses during the execution on the graphics/compute engine are already accounted in the *Ticks* specified in the runnable.

7.3. GPU Offloading

In Amalthea parallelism is expressed on task level, this means all items (e.g. runnables) within a task are executed sequentially. Therefore, in case a workload is offloaded to the GPU, this workload is modeled as a separate task, which is shown in Figure 8. In addition to the offloaded workload to the GPU, there is still some pre- and postprocessing to be executed which is done on the host CPU and represented as two individual tasks. The tasks are chained via interprocess stimuli, an example is shown in Table 1 for the *Detection* task.

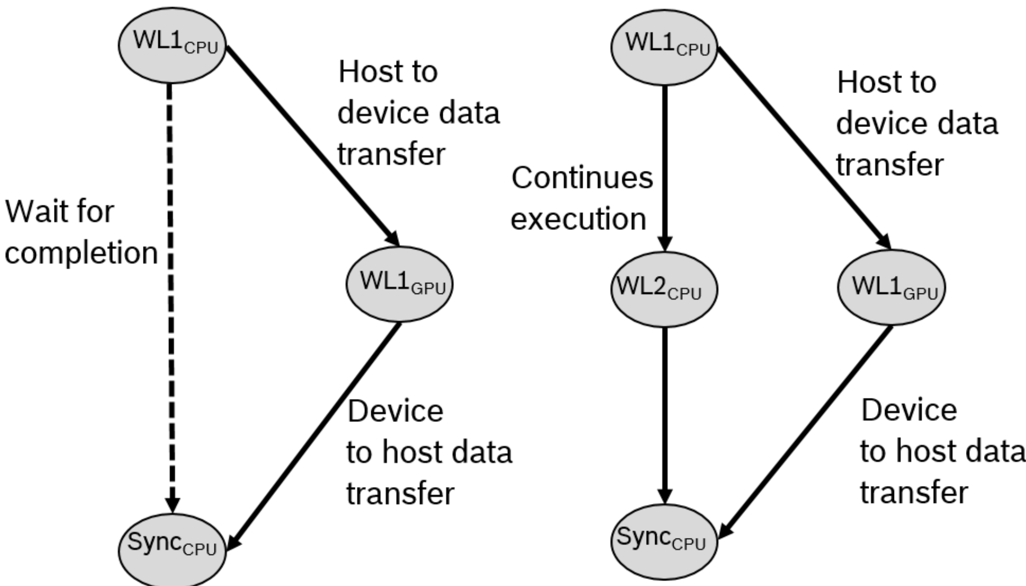


Figure 8: Synchronous and asynchronous offloading

The offloading to the GPU requires data to be transferred from the CPU memory region to the GPU memory region (explained in Section 5.3). This is accomplished by separating this offloading in three different runnables:

1. host to device copy operation
2. execution

3. device to host copy operation

Figure 9 shows an example with three runnables and the task calling the runnables. In addition, the offloading can be performed in synchronous or asynchronous manner. Figure 8 shows the two different mechanisms. The workload $WL1_{CPU}$ is executed on a CPU. $WL1_{CPU}$ contains label accesses or ticks (to specify the computational cost).

- In the synchronous case, the CPU task offloads the work to the GPU, and then actively waits for the execution on the GPU to be completed. After completion the execution on CPU side can directly start with a latency of 0 due to the active waiting.
- In the asynchronous case the CPU task offloads the work to the GPU and goes into a passive waiting mode, this implies that the scheduler can schedule other workloads on the CPU. In case the offloaded GPU task has finished and the CPU task is active again (responsibility of the scheduler), additional costs due the async offloading have to be accounted before the execution on the CPU can be continued.

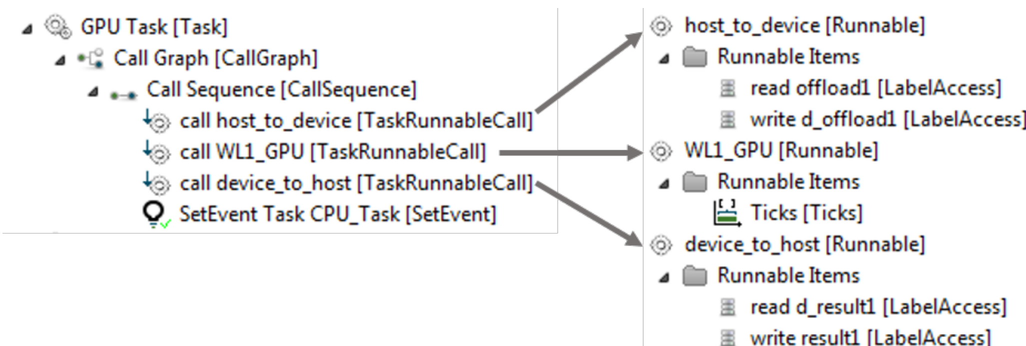
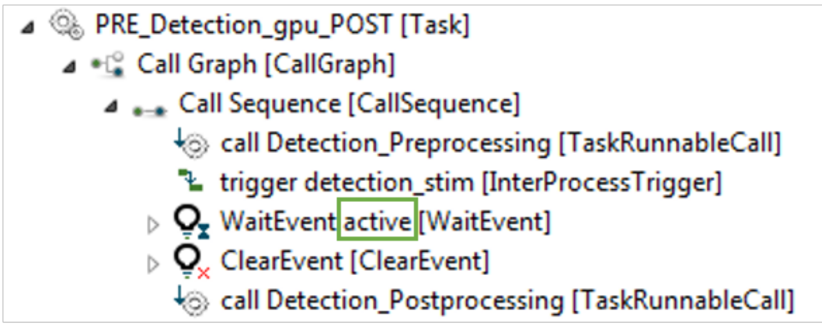


Figure 9: GPU workload example with copy runnables

The concrete modeling is shown in Figure 10. The offloading is triggered with the inter process trigger *(detection_stim)* within the task mapped to a CPU. This trigger activates a task which is mapped to a GPU (e.g. Figure 9). The corresponding *WaitEvent* is entered, active in case of the synchronous and passive in case of asynchronous offloading. After the workload of the GPU is finished, the resulting data is copied back from the GPU to the CPU

and the corresponding event *SetEvent* is set by the GPU task (see Figure 10). In the asynchronous case the additional offloading costs are modeled with an extra runnable called *asyncOffloadindCosts* which include additional ticks. In both scenarios, the *ClearEvent* resets the synchronization mechanism.



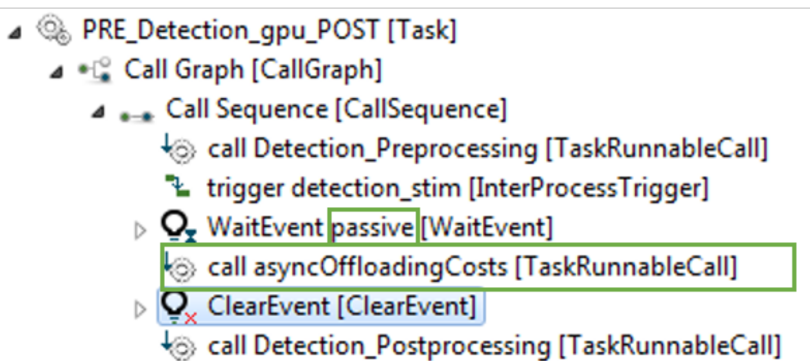


Figure 10: Modeling synchronous and asynchronous offloading of the Detection-Preprocessing Task described in Table 1

7.4. GPU Scheduling

For the GPU Scheduling, we assume one combined scheduler for the GPU (instead of having a separate scheduler for the copy engine and the execution engine), this is possible due to the assumption that every application has just one stream (currently represented with one task executed on the GPU). Therefore, given an application, the time slice allocated to its channel is used for both copying and execution. We do not explicitly model channels

in Amalthea but associate one GPU task to each application. These tasks are scheduled by the custom time slice scheduler of the GPU with minimum and maximum time slice of 1 ms and 500 ms respectively. These attributes are set in the Amalthea model under *Operating System* → *GPU_Cluster* → *GPU_Sched* → *User Specific Scheduling Algorithm*. The scheduling policy is explained in Section 5.1. The aforementioned weighted round robin scheduler policy is not directly supported in Amalthea, therefore a custom scheduler is used. The individual time slices for the different tasks can be specified in the mapping model when mapping a task to a scheduler *TaskAllocation*. An example is shown in Figure 11. Note that this is just an example and do not represent the best time slice configuration for the model. The time slices have to be adapted based on the mapping of the tasks to the targets (A57, Denver, GPU).

Allocation: Scheduler GPU_Sched -- Task Lane_detection [TaskAllocation]
TimeSlice_Lane_detection -> 50000µs [ParameterExtension]
Allocation: Scheduler GPU_Sched -- Task Detection [TaskAllocation]
TimeSlice_Detection -> 50000µs [ParameterExtension]
Allocation: Scheduler GPU_Sched -- Task SFM [TaskAllocation]
TimeSlice_SFM -> 7200µs [ParameterExtension]
Allocation: Scheduler GPU_Sched -- Task Localization [TaskAllocation]
TimeSlice_Localization -> 50000µs [ParameterExtension]

Figure 11: Specifying individual time slices for tasks

7.5. Mapping-dependent Resource Consumption

The number of ticks for a runnable is dependent of the executing processing unit. To be able to express different tick values for different processing units multiple potential mapping targets with their tick values can be specified. This can be used for analysis to optimize end-to-end latencies by changing the task mapping of the given model. Figure 12 shows an Amalthea modeling example, which includes multiple tick values for different mapping targets: e.g. workload WL3 requires a default of 6000 ticks when mapped to the Cortex A57 (default), and 3000 ticks when mapped to the GPU. Note these entries also implicitly define the possible mapping targets. An overview about the different tasks of the presented application in section 3 with potential mapping targets is shown in Table 1.

WL3 [Runnable]
Runnable Items
read inputData [LabelAccess]
Ticks [Ticks]
default: DiscreteValue Constant (value: 6000) [DiscreteValueConstant]
Definition Denver -- DiscreteValue Uniform Distribution[5000, 6500] [TicksEntry]
Definition GPU_def -- DiscreteValue Constant (value: 3000) [TicksEntry]
write outputData [LabelAccess]

Figure 12: Extended ticks in Amalthea

The modeling concepts refer to Amalthea version 0.9.3; In the meanwhile the WATERS example model was added to the included examples which are part of the official Eclipse APP4MC distribution (starting from version 0.9.8). This means the model is also available in newer Amalthea versions and is migrated to every new Amalthea release. The challenge model is available in the latest Eclipse APP4MC version (starting from 0.9.8) which can be directly downloaded one the official APP4MC homepage [1]. The model is part of the included examples which are migrated to every new Amalthea version.

Also, currently the modeling was done with respect to the Nvidia platform. However most of the above concepts are common to most heterogeneous architectures and they can be modeled similarly with Amalthea.

8. Proposed Solutions

Different solutions for the above challenge were proposed during the Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2019), which was collocated with the Euromicro Conference on Real-time Systems (ECRTS 2019). In the following, each solution is briefly summarized. Please note that the challenge was not posed in a way to allow for a quantitative comparison of the proposed solutions. The goal was rather to explore the suitability of approaches from the real-time community for different real-time analysis questions in the context of a complex automated driving software stack that is executed on a modern heterogeneous HW platform. For this reason, the comparison rather focuses on the utilized approaches as well as on which (sub-)aspects of the challenge have been addressed. It is to be further reiterated that the objective of the

challenge was not to deeply assess the technically solutions provided, but rather to validate if the presented model presents sufficient information to system designers or analysis tools towards performance analysis.

8.1. Solution 1: Applying an extended classical response time approach [27]

Authors in [27] employ conventional RTA for fully-preemptive tasks under rate monotonic scheduling running on CPUs using the windowing technique and combines it with weighted round robin (WRR) RTA for GPU tasks. Additionally, they define a memory contention model for GPU copy engines as well as differences of asynchronous and synchronous GPU offloading mechanisms.

- CPU Response Time analysis considers task access latency, contention latency and locking latency. A busy window analysis is used based on the work in [36], where the interference from high priority tasks and the different label access latencies are integrated.

- Additionally, the cases for synchronous and asynchronous offloading are addressed for CPU RTA analysis, again using a busy window analysis technique.

- The GPU response time analysis is based on the work in [36] and considers the copy engine costs (to read from memory, write to GPU memory and vice versa). It takes into account queueing delays from earlier copy operations and also contention with other CPU tasks while accessing the shared memory. Finally this contention delay is integrated into the GPU RTA assuming a weighted round robin mechanism among tasks on the GPU.

- Task mapping is done using genetic algorithms with a view to meet 3 different criterion. Three different fitness functions are employed to minimize the response time, minimize the task-chain latency and balance the load across the CPUs. The authors also compared the different solutions derived from employing these criteria.

- The authors also address the issue of computing a right time slice to enable the tasks to be scheduled on the GPU.

- Finally a method to compute the task chain latencies is suggested. However the communication semantics are not spelled out clearly and

the latency seems to have some double accounting since the response times and the periods of the tasks along the chain are simply added.

8.2. Solution 2: Design Space Exploration using an Evolutionary Algorithm [28]

The authors in [28] propose a model based approach of design space exploration (DSE) employing an multi-objective evolutionary algorithm for the task-to-core mapping and an SMT solver for the schedule synthesis. Tasks follow an acquisition, execution and restitution (AER) model In addition to the given assumptions of the challenge, the authors make the following assumptions

- All memory transfers are performed asynchronously via the Direct Memory Access (DMA) engine.
- DMA prefetching has no interference effects on latency due to serializing the requests.
- Bandwidth of DMA access equals those of the GPU.
- Single rate tasks and no task periods provided in the challenge nor preemption is considered.

The authors highlight the fact that strategically prefetching data can help in reducing the makespan of tasks by overlapping computation and communication. With this is mind they jointly explore task mappings and data transfers strategies that are strongly coupled via common communication resources. The evolutionary algorithm based DSE is used to map tasks to CPU cores or the GPU, and and data transfers to the DMA unit. For each candidate mapping, the SMT scheduler and the temporal analysis are employed to estimate the resulting communication latencies. They also compare the response time of tasks with and without prefetching.

While this approach emphasizes on DMA transfers and the benefits of prefetching, the additional assumptions made by the authors weaken the usability of this approach. The scheduling on the GPU is not addressed, and the memory contention effects are overlooked. Furthermore with the assumption of non-preemptive single-rate tasks, the interference effects are also not considered in detail. Also offloading cases are not addressed in this solution.

8.3. Solution 3: Analytical approach with focus on end-to-end latencies [29]

The authors of [29] mainly focus on optimizing the end-to-end latencies, considering the LET and implicit communication semantics. They also apply standard response time analysis techniques for the fixed priority preemptive scheduling on the CPUs (including offsets) [31] as well as weighted round-robin scheduling on the GPUs [37].

- The CPU response time analysis basically uses the classic busy window analysis [31]. CPU tasks that offload work to the GPU essentially have a pre-processing, offloading (to the GPU) and post-processing phase: for such tasks, the response time in each phase is computed by considering an offset equal to the best (and worst-case) response time of the previous phase. The release jitter is also computed accordingly. The authors then apply the offset based analysis as proposed in [31]. With their approach, the authors therefore have a common analysis for synchronous and asynchronous offloading scenarios.

- Tasks scheduled on the GPU are analyzed using the busy window analysis for weighted round robin scheduling proposed in [37]. The delay accounted for the copy-in and copy-out operations is integrated in the analysis.

- The authors then focus on optimizing the end-to-end latencies, considering the different communication semantics [32]. Memory access latencies are computed according to the static analytic model presented in the paper. Given that there are no tuning knobs with the LET semantics since the end-to-end latency is dependent on the task periods which cannot be changed, the authors do not attempt further optimization. For implicit communication, the authors carry out a design space exploration (DSE) using genetic algorithms, by tuning task priority assignments, task allocations and GPU time-slices to arrive at different response times.

8.4. Solution 4:Applying the Self-Suspending Task Theory [30]

The authors of [30] provide a scheduling model description which considers the different specified task runtimes for different mapping targets, as well as synchronous offloading to the GPU. Each task is modeled as an interleaved sequence of execution region on the CPU and acceleration regions where the

Analysis Features	Solution 1	Solution 2	Solution 3	Solution 4
CPU Response Time Analysis	yes	partially	yes	yes
GPU Response Time Analysis	yes	no	yes	yes
Task chain latencies	partially	partially	yes	yes
Synchronous offloading	yes	no	yes	yes
Asynchronous offloading	yes	no	yes	no
Optimize Task Mapping	yes	partially	yes	yes
Optimize GPU Time Slices	yes	no	yes	partially

Table 2: Problems tackled by different approaches

task waits for completion of a GPU operation.The authors model a form of synchronous offloading scenario wherein on launching a GPU operation, the CPU suspends itself. The offloading of acceleration regions to the GPUs by the CPU tasks is thereby analyzed by means of the self suspending task theory [38]. Tasks on the GPU are assumed to be scheduled using a weighted round robin (WRR) mechanism. The response time analysis uses a supply bound function to model the service provided by the GPU to each task in an given time interval considering WRR. Response time analysis of tasks on the CPUs is carried out using the self suspending task theory [38], where acceleration regions(executed on the GPU) correspond to suspension regions and the timing effects of suspensions are accounted as release jitter.

In order to meet the different criterion stated in the challenge, the authors linearize the analysis and in include it in a Mixed-Integer Linear Programming (MILP) with the goal to optimize the following objectives:

- minimize task chain latencies.
- task to core mapping (considering the GPU load and acceleration of tasks).
- ensure schedulability.
- priority assignment.

Furthermore the solution proposes several potential extensions to their analysis and modeling approach as well as to the WATERS challenge.

8.5. Comparison of the Approaches

Summary. The purpose of the challenge presented in this paper was to explore the fit of state-of-the-art real-time modeling and analysis techniques for

an automated driving application executed on a heterogeneous HW platform where time critical cause-effect chains span across general purpose microprocessor cores and special accelerators like GPUs. It was expected that an overall analysis incorporating all aspects (CPU scheduling, GPU scheduling, task mapping, synchronous and asynchronous offloading, response time analysis, end-to-end latency analysis) would be challenging using the available toolbox of real-time research.

The results, however, show that this is only partially true. While each of the proposed solutions made some simplifying assumptions for parts of the overall problem as seen in Table 2, they nevertheless proposed accurate methods for other parts.

Each of the solutions interestingly used a different toolset from existing real-time research for solving the same problem. While authors [27] and [29] used genetic algorithms for design space exploration (with different objectives), [30] used a Mixed Integer Linear Programming ToolKit and [28] used an evolutionary algorithm with an SMT solver for schedule synthesis. Classical response time analysis using busy windows was used in [27] and [29], while [30] applied the self-suspending theory for the same.

In summary, it can thus be concluded that the toolbox of real-time research fits the proposed challenge. However, still work remains to be done to develop a holistic approach that covers all the above-mentioned aspects in detail and also covers future extensions of the challenge discussed in the following section.

9. Additional Modeling Extensions

The case study presented in this paper contains simplifying assumptions. Due to the interesting results obtained in the first round of solution proposals, it makes sense to introduce extensions to obtain a more realistic and complete model as basis of a new *Industrial Challenge* in the context of the ECRTS conference. Again, the main purpose of such an updated and far more realistic challenge would be to evaluate the extent to which methods from real-time research can answer various analytical questions.

9.1. GPU Modeling Extensions

GPU hardware. In the challenge model, the entire GPU hardware was abstracted as a single execution unit. However we can further extend it to model

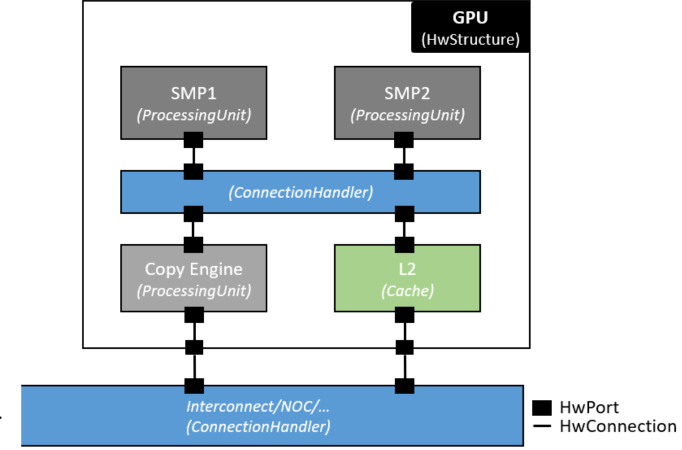


Figure 13: GPU modeling example

the GPU hardware in more detail as a hierarchical structure of streaming multiprocessors, grids, blocks and threads, including the shared cache.

As a first step, instead of representing the entire GPU by a single processing unit (PU), we can model more details as seen in Figure 13. Each streaming multiprocessor (2) is modeled by a PU and furthermore the copy engine can be modeled by another PU. This allows us the mapping of different tasks to the modeled PUs, e.g. a copy_in task can be executed on the copy engine parallel to the execution task of another stream on one of the streaming multiprocessors like it was shown in Figure 5.

In addition, the Level-2 cache resides within the GPU structure and is connected to the external interconnect via ports. In order to represent platforms like the C2050 [19] that support a separate copy engine for each direction between the host and the device, we can model each of them explicitly.

Streams. In the current model, in line with the assumption of each application having only a single stream, the Amalthea model can only handle single streams. By explicitly modeling streaming processors and the copy engine in the hardware model, we overcome this limitation and multiple streams can be represented. Accordingly we adapt the representation of streams in the software model. Instead of using one task with multiple runnables, we use tasks for every sub-operation(copy_in, execute and copy_out). Each GPU task in a given stream is therefore split into three different tasks representing the copy_in, kernel execution and copy_out task. The copy_in and copy_out

(sub)tasks are assigned to the copy engine(s) while the execution task is assigned to one of the SMs. We ensure a ordered, serialized execution of the copy_in, execute and copy_out tasks by using the interprocess trigger mechanism between them. In order to associate these three sub-tasks to the same task, it is recommended that tasks have the same prefix name or be tagged to a common application name.

Note: It is necessary to use tasks because tasks are the only scheduleable entities in Amalthea and therefore the only possibility to model parallel execution on multiple processing elements.

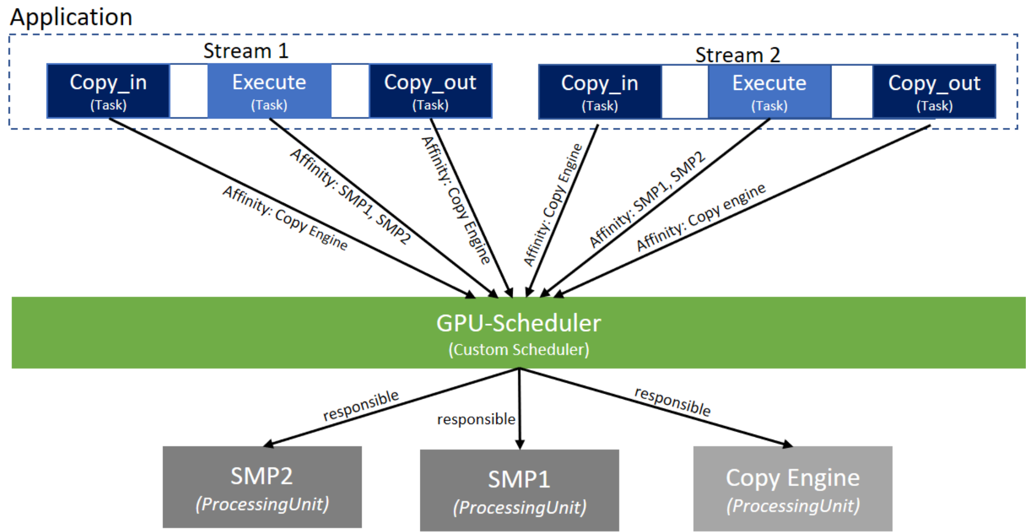


Figure 14: Scheduling the compute and copy tasks on the SMs and copy engines

A common GPU scheduler is responsible for scheduling all tasks on the multiple SMs and the copy engine(s) as seen in Figure 14. By mapping the different tasks to the responsible scheduler and using the affinity attribute of the mapping element, the target for each task can be specified - e.g. copy_in and copy_out tasks to the copy engine and the execute task with the affinity for both streaming multiprocessors. Note that since only one application context can run on the GPU at any given time, only streams of a given application may be executed concurrently on the SMs.

Generally, we do not intend to model DEFAULT streams. Given the non-determinism they bring in, it is unlikely that they will be considered for applications needing strict timing guarantees. Something that we consider interesting for future works is to consider the computing nature (i.e.,

memory intensive or compute intensive) of the streams. In [45], the authors highlighted that memory intensive and compute intensive streams interleave very well in the GPU (mainly because they do not interfere with each other when they execute concurrently), while streams of the same nature lead to stronger interference.

GPU memory hierarchy and types. Additionally, the current work does not consider the specific memory regions to which applications are mapped – these regions could be of type pinned, paged or unified, thereby impacting the memory access times. On accounting for these details, we plan to arrive at tighter estimates on the execution time of an application launched on the GPU. We can use custom properties in Amalthea to mark a memory as pinned or paged.

9.2. Memory Access Extensions

In the current model, only constant latencies are used for describing a memory access. However in reality, memory access latency is dependent on different components (like interconnects) in the access path to memory. Furthermore in cache-based systems usually a whole cache line is transferred and the access is defined by the latency, the bandwidth and the bit width in combination with the maximum burst size of the interconnect. In order to incorporate these parameters, the hardware model can be extended in terms of more detailed access paths instead of using fixed latencies for memory accesses.

In Amalthea, this can be realized by connecting the different ports of *ProcessingUnits, Caches, ConnectionHandlers* and *Memories* with *HwConnections*. Then, concrete *HwAccessPaths* through the system can be added to the *HwAccessElements* of each processing unit. In this case, the sum of annotated latencies of the different components of a specified *HwAccessPath* e.g. at the interconnect (specified at the connection handler definition) are summed up to the overall latency for an access to the specified target in the *HwAccessElement*. In addition to the latency, the bandwidth can also be specified. The used bandwidth of *HwAccessPaths* is the minimum annotated bandwidth of all elements in the specified path.

By using information about the cache line size, the bit width of the ports, burst size and information like latency and bandwidth of an *HwAccessPath*, more detailed simulation regarding memory accesses and a more detailed interference analysis compared to 6.2 can be done.

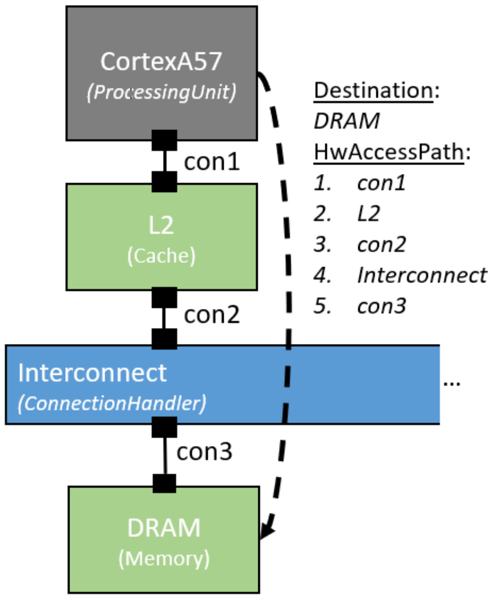


Figure 15: Access path example

9.3. QNX Adaptive Partitioning Scheduling

In this section, we show how we can further extend the modeling capabilities to include more practically deployed schedulers like the QNX Adaptive partitioned scheduler, in addition to the fixed priority schedulers which are already modeled in Amalthea.

The QNX Neutrino OS offers regular schedulers like the round robin scheduler and the sporadic scheduler. Additionally, it also offers the *Adaptive Partitioning Scheduler (APS)*, a global scheduler that has been designed with the objective to globally guarantee specified minimum shares of CPU time to groups of threads or processes residing within virtual containers called partitions. The percentage of the CPU time allotted to a partition is called budget. In effect, partitions are merely virtual containers for managing the budget. Partitions are not directly scheduled and executed, this is done at thread level as with other operating systems. The APS scheduler is invoked at each scheduling event (timer tick, thread termination, message arrival, etc.) and computes a priority metric for all the ready-to-run threads in the system.

This metric is a function, among other aspects, of the thread priority, the share of budget consumed by the parent partition, etc. The thread with the

highest priority metric is selected to execute. In general, without getting into the specifics of the priority metric functions, the following behavior is visible. Whenever partitions are below their budgets, the APS chooses the highest priority ready thread to execute. In other words, when under-loaded, the APS behaves like a priority-based real-time scheduler. In the case when all partitions are out of budget, the APS divides time between them by the ratios of their budgets. When a partition is not running and has budget (called idle-time mode), it is distributed to other partitions with ready threads either by thread priority or by the ratio of their budgets. The APS, thus, throttles thread execution by measuring the average CPU usage of each partition. The average is computed every 1ms (or scheduling event) over a configurable averaging window (typically 100 milliseconds).

A ready thread whose partition is out of budget, meaning that the consumption of CPU time over the averaging window exceeds the partition's budget, is not allowed to run, if there exist other ready threads from other partitions with available budget. Only once enough time has elapsed for the average consumed CPU time of its parent partition to fall below the assigned budget, the thread will be allowed to run again. However, the thread is guaranteed to eventually run. For more details on the behavior, please refer to the Adaptive Partitioning Scheduler Users Guide [44].

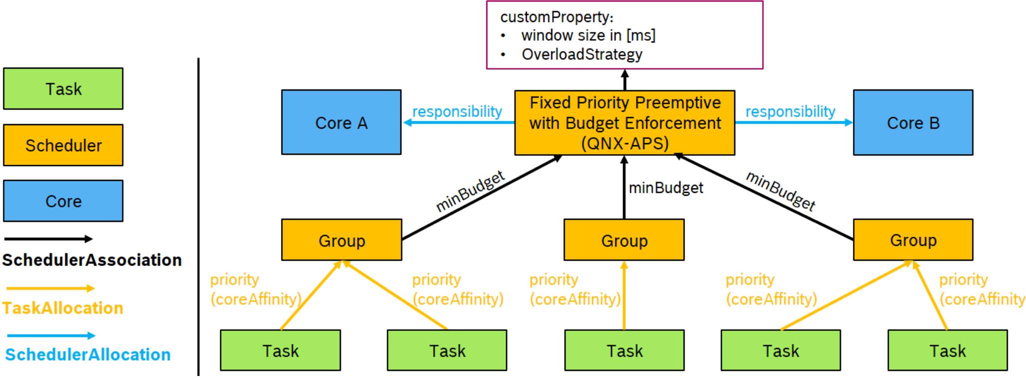


Figure 16: QNX Adaptive Partitioning Scheduler

In Amalthea the APS can be modeled as a hierarchical scheduler[5] as

[5]Information on how to model hierarchical schedulers with Amalthea can be found in the App4mc documentation [42]

depicted in Figure 16.

At the top-level, the APS is modeled as a fixed priority preemptive scheduler with budget enforcement. The size of the averaging window as well as the overload strategy are not directly supported by Amalthea and need to modeled as *custom properties*.

The nature of APS being a global scheduler is expressed by the *SchedulerAllocation* dependencies, assigning responsibility for all cores in the system. The partitions are modeled as logical groups. The processor capacity assigned to a partition is defined via the *minBudget* attached to the *SchedulerAssociation* edge between the APS and the partitions.

Tasks are allocated to the logical groups representing the APS partitions. Every task is assigned with a (globally unique) priority. The core affinities of tasks are optional, and can be used to restrict the task migrations performed by the global APS scheduler.

9.4. Extensions to Handle Publish-Subscribe Middleware Systems

Classically, real-time systems are built with precise knowledge about the activation patterns of concurrently executed applications. A large part of real-time theory is, for instance, based on the assumption that applications are activated *strictly periodically*. This periodic model can easily be extended with the notion of *activation jitter*, indicating that the individual points of execution may vary to a certain extent. For many applications (e.g. control) assumptions like periodic execution make perfectly sense, and most real-time operating systems come along with built-in support for realizing such behavior (e.g. through *periodic tasks*).

However, when looking at middleware systems such as ROS2 [39] or MQTT [40] which heavily use the *publish-subscribe messaging pattern*, controlling and understanding the activation patterns of applications is far more involved.

Publish-subscribe is a messaging pattern implemented by most popular middleware systems. Messages are sent asynchronously by so-called publishers without the knowledge how many (if any) so-called subscribers receive the messages. This decoupling of senders and receivers provides great flexibility since dependencies are not explicitly programmed, and can even be added dynamically during runtime. Messages are usually “delivered” by the execution of callback-functions giving the subscriber(s) the possibility to react and process the received data. In addition to this data-driven activation, most

middleware system also support time-driven activation of callback-function through timers.

When it comes to building real-time systems, the publish-subscribe abstraction can cause problems. For instance, since activated callback-functions can publish messages themselves, complex behavior can emerge making it potentially very hard to understand the dynamic behavior of an application.

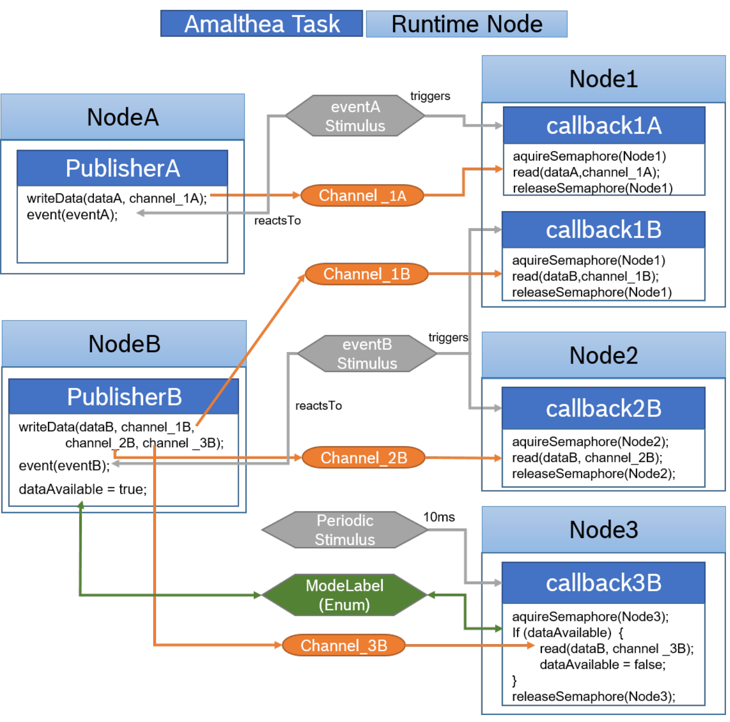


Figure 17: Publisher, Subscriber modeling

In the following, we describe how the publish-subscriber messaging pattern can be modeled in Amalthea. Thereby, the focus lies on a steady-state model of the system. Dynamic changes can be modeled but are out of scope in this work. The approach is illustrated in Figure 17. Please note, that we use the term *node* to denote a process publishing and subscribing to topics.

Between every publish/subscriber pair, a data *Channel* is modeled in Amalthea. In order to simplify the modeling effort, we model every *node* that subscribes to multiple topics into several *Tasks* in Amalthea. The *node* itself is not modeled in Amalthea. This split into individual tasks allows us to model different behavior (e.g. publishing new data) and execution times

for each callback.

Since we are modeling each callback as a separate task, we need to ensure that a *node* subscribing to multiple callbacks is executing them in FIFO manner, which is assumed to be the default behavior in this paper. To that end, we use a common *Semaphore* for each *node* to sequentialize the execution order.

The activation of callbacks can be either data or time-triggered. In Figure 17 *callback3B* is triggered every 10ms by a *PeriodicStimulus*. In Amalthea we model an additional *ModeLabel* as a boolean flag to ensure, that the callback functionality is only executed when new data is available. The callbacks of *Node1* and *Node2* are triggered on data availability. An *Event* sent out by the publisher activates the corresponding *EventStimulus*, which, in turn, triggers the corresponding callbacks of all subscribers.

Please note, that in the above proposed modeling approach, all callbacks are assumed to be executed in FIFO manner. In reality, existing middleware systems often exhibit more complex semantics that are implementation defined and not well documented. ROS2, for instance, repeatedly fetches all messages that have arrived in the underlying communication system (i.e. DDS) and executes the corresponding callbacks until completion. Only then newly arrived messages are fetched. During execution, callbacks triggered by timers are prioritized, afterwards callbacks triggered by messages are processed in the order they have been registered in the source code [41]. For now these sources of non-determinism and complex delays are not modeled in this work, hoping that future ROS2 releases exhibit a more well-defined callback execution behavior.

10. Conclusion

System designers and practitioners across the industry and academia are grappling with the problem of mastering the complexity of adopting heterogeneous systems in emerging automotive applications. In this work, we highlighted the existing gaps in available modeling tools and also presented different factors that must be considered while efficiently deploying applications on heterogeneous hardware considering a modern autonomous driving use case. We demonstrated how we can model these systems (the software and hardware) using the Amalthea modeling tool. We then presented the WATERS industrial challenge and the solutions presented by the different participants. Extensions to handle newer scheduling paradigms (like the

QNX APS), newer communication paradigms (publish-subscribe) and more detailed hardware modeling are also proposed. Different approaches in tackling the response time analysis and mapping problem were evaluated. We believe that since performance effects for modern systems, such as for the presented automated driving application, are much more difficult to predict and master compared to classical micro-controller based systems, performance modeling and analysis, as presented in this work will play in future an increasingly important role in automotive systems engineering.

Acknowledgment

This work was partially funded by the *PANORAMA* project of the German Federal Ministry for Education and Research with the funding IDs 01IS18057A.

This work has been supported by the European Union's Horizon 2020 research and innovation program under grant agreement No 871669.

The responsibility for the content remains with the authors.

References

[1] Eclipse APP4MC, `https://www.eclipse.org/app4mc/`

[2] Industrial Challenge, WATERS 2019, `https://www.ecrts.org/waters/`

[3] F. Wurst et al., "System Performance Modelling of Heterogeneous HW Platforms: An Automated Driving Case Study," 2019 22nd Euromicro Conference on Digital System Design (DSD)

[4] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein and M. Wolf, "Future Automotive Systems Design: Research Challenges and Opportunities", International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Turin, Italy, 2018.

[5] Rafik Henia and Laurent Rioux. The 2015 fmtv challenge. see `https://waters2015.inria.fr/files/2014/11/FMTV-2015-Challenge.pdf`, 2015

[6] A. Hamann, D. Ziegenbein, S. Kramer and M. Lukasiewycz, "Demo Abstract: Demonstration of the FMTV 2016 Timing Verification Challenge," 2016 IEEE Real-Time and Embedded Technology and

Applications Symposium (RTAS), 2016, pp. 1-1, doi: 10.1109/RTAS.2016.7461330.

[7] Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F., and Ziegenbein, D. (2017, June). Waters industrial challenge 2017. In International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS).

[8] OMG MARTE Group. A UML profile for MARTE:Modeling and analysis of real-time embedded systems,beta 2 (convenience document with change bars). InOMG MARTE documentation, pages 1–676, 2008.

[9] Timing Architects Tool Suite, https://www.vector.com/int/en/products/products-a-z/software/ta-tool-suite/

[10] Capella, Open Source Solution for Model-Based Systems Engineering, https://www.eclipse.org/capella/

[11] M. Gonzalez Harbour, J. J. Gutierrez Garcia, J. C. Palencia Gutierrez and J. M. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," Proceedings 13th Euromicro Conference on Real-Time Systems, 2001, pp. 125-134, doi: 10.1109/EMRTS.2001.934015.

[12] ChronSIM, Unleash the Power of Timing Simulation https://www.inchron.com/chronsim/

[13] Jetson TX2 Module, https://developer.nvidia.com/embedded/buy/jetson-tx2

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

[15] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst and D. Ziegenbein, "WATERS industrial challenge 2017", 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), https://waters2017.inria.fr/challenge

[16] How to Optimize Data Transfers in CUDA C/C++, https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc

[17] CUDA C++: Streams and Concurrency. Available at: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[18] N. Capodieci, R. Cavicchioli, M. Bertogna and A. Paramakuru, "Deadline-Based Scheduling for GPU with Preemption Support", IEEE Real-Time Systems Symposium (RTSS), Nashville, 2018.

[19] How to Overlap Data Transfers in CUDA C/C++ Available at: https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

[20] Euromicro Conference on Real-time Systems 2019, https://www.ecrts.org/

[21] Description of the WATERS Industrial Challenge 2019, https://www.ecrts.org/forum/viewforum.php?f=43

[22] AUTOSAR: Specification of RTE Software, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_RTE.pdf

[23] M. Bechtel et al. "Deeppicar: A low-cost deep neural network-based autonomous car", 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2018.

[24] N. Otterness et al. "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017.

[25] Kato, Shinpei, et al. "Autoware on board: Enabling autonomous vehicles with embedded systems", ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS) 2018.

[26] J. Redmon and F. Ali, "YOLOv3: An Incremental Improvement", CoRR Journal abs/1804.02767 (2018)

[27] R. Höttger, J. Ki, T. B. Bui, B. Igel, and O. Spinczyk. "CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems", WATERS Industrial Challenge, ECRTS 2019, https://www.ecrts.org/forum/viewtopic.php?f=43&t=134

[28] A. Diewald, S. Barner, and S. Saidi. "Combined Data Transfer Response Time and Mapping Exploration in MPSoCs", WATERS Industrial Challenge, ECRTS 2019, https://www.ecrts.org/forum/viewtopic.php?f=43&t=135

[29] L. Krawczyk, M. Bazzal, R. Govindarajan, and C. Wolff. "An analytical approach for calculating end-to-end response times in autonomous driving applications", WATERS Industrial, Challenge, ECRTS 2019, https://www.ecrts.org/forum/viewtopic.php?f=43&t=136

[30] D. Casini, P. Pazzaglia, A. Biondi, G. Buttazzo, and M. Natale. "Addressing Analysis and Partitioning issues for the WATERS 2019 Challenge", WATERS Industrial Challenge, ECRTS 2019, https://www.ecrts.org/forum/viewtopic.php?f=43&t=137

[31] J. C. Palencia and M. Gonzalez Harbour. "Schedulability analysis for tasks with static and dynamic offsets", Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), Madrid, Spain, 1998, pp. 26-37.

[32] A. Hamann, D. Dasari, S. Kramer, M. Pressler and F. Wurst. "Communication centric design in complex automotive embedded systems", 29th Euromicro Conference on Real-Time Systems (ECRTS), Dubrovnik, 2017.

[33] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter and R. Ernst, "System level performance analysis - The SymTA/S approach", IEE Proceedings - Computers and Digital Techniques, vol. 152, no. 2, pp. 148-166, March 2005.

[34] R. Cavicchioli, N. Capodieci and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms", 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, 2017.

[35] N. Capodieci et al. "Detailed characterization of platforms", High-Performance Real-time Architectures for Low-Power Embedded Systems (HERCULES), http://hercules2020.eu/wp-content/uploads/2017/03/D2.2_Detailed_Characterization_of_Platforms.pdf

[36] I. Sanudo, P. Burgio, and Marko Bertogna. "Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System", International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, WATERS, 2016.

[37] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst, "Improved response time analysis of tasks scheduled under preemptive Round-Robin", Proceedings of the 5th IEEE/ACM International Conference on Hardware Software Codesign and System Synthesis (CODES+ISSS '07). Association for Computing Machinery, New York, NY, USA, 179–184

[38] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems" Real-Time Systems, Sep 2018.

[39] ROS2 - Robotic Operating System 2, https://index.ros.org/doc/ros2/

[40] MQTT - Message Queuing Telemetry Transport, http://mqtt.org/

[41] D. Casini, T. Blaß, I. Lütkebohle, and B.B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling", 31st Euromicro Conference on Real-Time Systems (ECRTS), 2019.

[42] Description of hierarchical scheduling concept of App4mc, https://www.eclipse.org/app4mc/help/app4mc-0.9.8/index.html#section2.2.6

[43] Sprunt, Sha, and Lehoczky "Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System", 1989

[44] QNX User Guide http://www.qnx.com/developers/docs/6.5.0/index.jsp

[45] I.Sanudo, N.Capodieci, J.Martinez, A. Marongiu and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective Ignacio Sañudo University of Modena and Reggio

Emilia", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2020.

[46] G Brilli and P. Burgio, "Interference analysis of shared last-level cache on embedded GP-GPUs with multiple CUDA streams", in Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS) 2019

[47] I. Sanudo, N. Capodieci and R. Cavicchioli, "A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS", IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society, 2018.

[48] M. Becker, S. Mubeen, D. Dasari, M. Behnam and T. Nolte, "A generic framework facilitating early analysis of data propagation delays in multi-rate systems (Invited paper)," 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017, pp. 1-11, doi: 10.1109/RTCSA.2017.8046323.

[49] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. 2018. Analyzing end-to-end delays in automotive systems at various levels of timing information. SIGBED Rev. 14, 4 (November 2017), 8–13. DOI:https://doi.org/10.1145/3177803.3177805