A scenario- and platform-aware design flow for image-based control systems

Sajid Mohamed[a,*], Dip Goswami[a], Vishak Nathan[b], Raghu Rajappa[b], Twan Basten[a,c]

[a]*Electronic Systems group, Eindhoven University of Technology, The Netherlands*
[b]*Sioux Logena B.V., The Netherlands*
[c]*ESI, TNO, The Netherlands*

Abstract

Image-based control (IBC) systems are increasingly being used in various domains including autonomous driving. The key challenge in IBC is to deal with high computation demand while guaranteeing performance and safety requirements such as stability. While modern industrial heterogeneous platforms, such as NVIDIA Drive, offer the necessary compute power, application development on these platforms with performance and safety guarantees is still challenging. Alternative time-predictable platforms are not yet in widespread use.

A typical design flow for IBC systems consists of three distinct elements: (i) *mapping* tasks onto platform resources; (ii) *timing analysis*, consisting of *task-level* worst-case execution time (WCET) analysis and *application-level* analysis to obtain worst-case performance bounds on aspects such as latency and throughput; (iii) *controller design* using the obtained performance bounds, ensuring performance and safety. While such a three-step design process is modular in nature, it usually leads to over-dimensioned systems with sub-optimal performance, because task- and/or application-level timing bounds are pessimistic.

We present a scenario- and platform-aware design flow for IBC systems that exploits *frequently occurring workload scenarios* to optimize performance. For industrial platforms, where tight task-level WCET bounds are difficult to obtain, we moreover propose to use *frequently occurring task execution times* instead of WCET estimates to obtain tight application-level temporal bounds. During controller design, we then optimize performance and guarantee stability by identifying appropriate *system scenarios* and by designing a *switched controller* that switches between those scenarios. We illustrate our method considering a predictable multiprocessor system-on-chip platform - CompSOC. We validate the proposed method using hardware-in-the-loop (HiL) experiments with an industrial heterogeneous multiprocessor platform - NVIDIA Drive PX2 - considering a lane keeping assist system (LKAS). We obtain an improved control performance compared to state-of-the-art IBC design.

Keywords: image-based control, switched linear control, scenario-based design, platform-aware design, multiprocessor implementation, hardware-in-the-loop validation

*Corresponding author
Email addresses: s.mohamed@tue.nl (Sajid Mohamed), d.goswami@tue.nl (Dip Goswami),

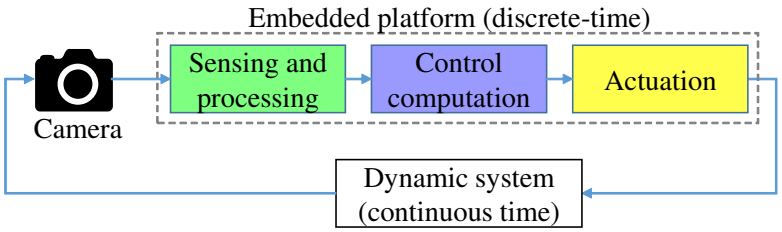


Figure 1: An image-based control (IBC) system: block diagram

1. Introduction

Image-Based Control (IBC) systems are a class of data-intensive feedback control systems whose feedback is provided by image-based sensing using a camera. Data-intensive feedback control systems are common nowadays due to advancements in cyber-physical systems (CPS) [1]. IBC systems have become popular with the advent of efficient image processing algorithms and low-cost CMOS cameras with high resolution [2]. The combination of the camera and the image processing algorithm gives necessary information on parameters such as relative position, geometry, relative distance, depth perception and tracking of the object-of-interest. Applications of IBC are found in robotics [2], autonomous vehicles [3, 4], advanced driver assistance systems (ADAS) [5], electron microscopes [6], visual navigation [7] and so on.

As illustrated in Fig. 1, a classical control implementation sequentially and periodically executes the sensing task, control compute task and actuating task. In an IBC system, the sensing task has a long, variable execution time and incurs a long sensing delay. Variability in execution time may occur due to variation in image-processing workload and/or in the platform load caused by other applications. The key challenge is to deal with this high dynamic computation demand while guaranteeing performance and meeting safety requirements such as stability.

IBC applications are nowadays usually implemented on some heterogeneous multiprocessor platform that may be shared with other applications. A typical design flow for IBC applications is composed of three distinct elements: (i) *mapping* tasks onto platform resources, which may be done manually or (semi-)automatically; (ii) *timing analysis*, consisting of *task-level* execution-time analysis and *application-level* analysis to obtain worst-case application-level latency and throughput bounds; (iii) *controller design* ensuring performance and safety guarantees taking into account task- and application-level temporal bounds. A typical flow abstracts variable task execution times through WCET estimates. These are often overly conservative, because of image-dependent workload variations and/or difficult to predict platform timing. This leads in turn to loose application-level timing bounds which hampers controller design. The resulting IBC system has sub-optimal control performance and is often over-dimensioned.

Fig. 2 illustrates a standard IBC implementation on a single processing core. A camera captures image frames with a period f_h, referred to as the frame rate. The frame rate determines the number of image frames that arrive per time unit, e.g., frames per second (fps). Typically, the cam-

Nathan.Vishak@sioux.eu (Vishak Nathan), Raghu.Rajappa@sioux.eu (Raghu Rajappa), a.a.basten@tue.nl (Twan Basten)

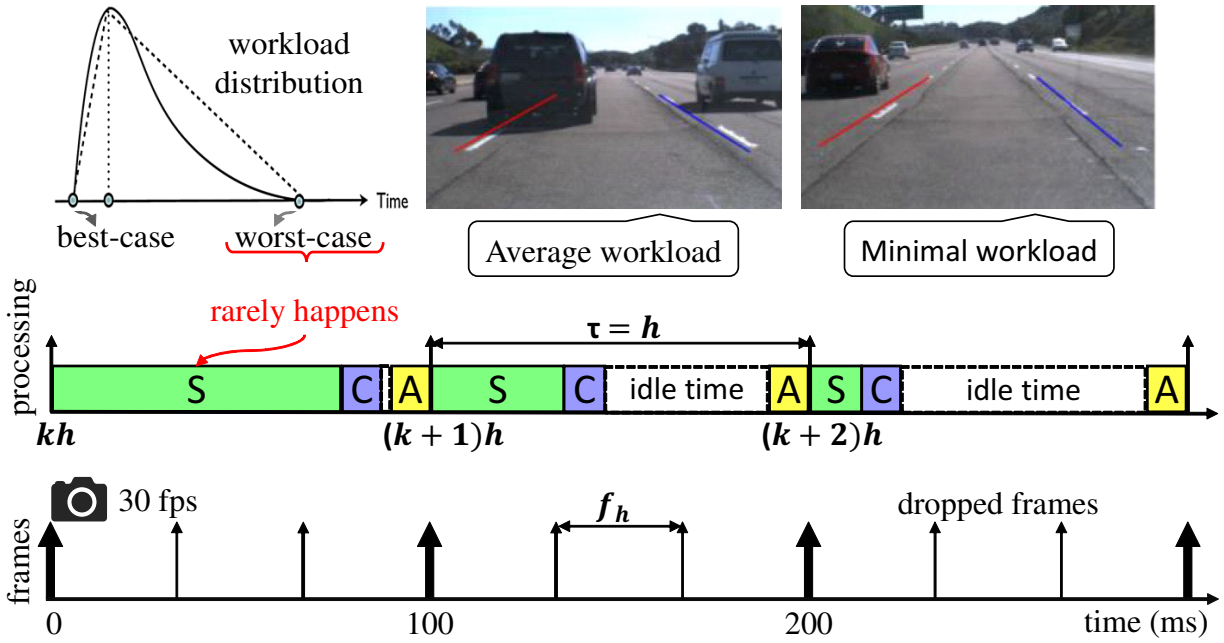


Figure 2: Illustration of classical IBC system implementation considering worst-case scenario. (S: sensing and image processing, C: control computation and A: actuation, see Fig. 1.)

era frame rate is much higher than the rate at which the frames can be processed on a single core. Pessimistic task-level WCET estimates and application-level analysis result in over-allocation of processing resources for the worst-case workload and idling for non-worst-case workloads (to keep the sampling period constant). This leads to a long sampling period h. With one processing core, we can only process every third image frame in this example. This results in sub-optimal control performance.

We present a model-based Scenario- and Platform-Aware Design flow (SPADe) for multi-processor IBC systems that exploits *parallelization of the sensing task* and *frequently occurring workload scenarios* to optimize performance. For industrial platforms, where tight task-level WCET bounds are difficult to obtain, we moreover propose to use *frequently occurring task execution times* instead of WCET estimates to obtain tight, though possibly no longer conservative, application-level temporal bounds for workload scenarios. For controller design, we identify appropriate *system scenarios* [8] that take into account platform mapping and controller performance for specific workload scenarios. Each system scenario corresponds to a specific sampling period. IBC performance is then optimized and stability is ensured by designing a *switched controller* that switches at run-time between system scenarios. Scenario-Aware Data Flow (SADF) [9] is used as a model of computation to capture parallelized (workload and system) scenarios and for timing analysis.

Predictable platforms, such as CompSOC [10] and PRET [11], provide predictable tight WCETs for individual tasks in an application. Further, the composability property of such platforms ensures that other applications sharing the platform do not interfere with the application under consideration. The WCET variations of a task execution on predictable platforms is mainly

due to the image workload variations. These properties make predictable MPSoC platforms suitable for model-based design. We develop and illustrate our SPADe approach for the predictable and composable interference-free CompSOC platform.

We further apply our method on the industrial NVIDIA Drive PX2 platform. Industrial heterogeneous platforms provide high compute power with support for extensive parallelization that is typically needed for data-intensive applications. However, such platforms are closed-source and use operating systems (OSs) that result in high variations in execution times of application tasks mapped to the platform. An ensuing challenge is that the application timing is difficult to predict. Derived task-level worst-case execution time (WCET) estimates are overly pessimistic. Model-based approaches using these pessimistic WCETs leed to pessismistic application-level performance bounds. This potentially compromises control performance and may lead to resource over-provisioning. However, task execution time distributions due to workload and platform-dependent variations can be statistically analysed from observed data, e.g. as a PERT distribution [12] (illustrated in Fig. 2). Such a distribution allows to classify the most frequently occurring task execution times. Using those execution times give tighter, though possibly no longer conservative application-level performance bounds. SPADe copes with possible timing analysis violations in the (switched) controller design. Using SPADe, we perform model-based design-space exploration (DSE) for an industrial setup over resource utilisation, quality of control and energy consumption to obtain Pareto-optimal system configurations at design time. We consider the concrete case study of a lane keeping assist system (LKAS) implemented on the NVIDIA Drive PX2 platform, sharing the platform with two other data-intensive applications - object detection and tracking and automatic emergency braking.

Contributions: This paper extends [13] and [14] that introduce the SPADe flow for predictable multiprocessor platforms and compare different controller design approaches. In the current paper, we present SPADe, and in addition to [13, 14],

1. we compare SPADe with a state-of-the-art pipelined control approach [15] through simulations for a predictable MPSoC platform - CompSOC. Pipelined control does not parallelize the sensing but uses multiple cores to pipeline multiple sensing instances. We provide a guideline when the SPADe approach is suitable with respect to the pipelined control approach.

2. we adapt SPADe targeting an industrial platform - NVIDIA Drive PX2. We show that we can leverage the principles of predictable model-based (co-)design for industrial platforms by carefully co-designing the image-processing implementation and the switched controller design, using a system-scenario-based approach [8].

3. we validate the SPADe approach in an industrial setting using a hardware-in-the-loop (HIL) experiment.

The rest of the paper is organised as follows. Section 2 presents related work. An introduction to the embedded IBC modelling, design, and implementation is given in Section 3 and the motivating lane keeping assist case study is briefly explained. The concepts related to the scenario-aware data flow model-of-computation used in our approach are explained in Section 4.

The problem statement addressed in our work is made precise in Section 5. The proposed scenario- and platform-aware design (SPADe) flow is presented in Section 6 with simulation results including a comparison with a state-of-the-art pipelined control design. The applicability of SPADe for an industrial IBC system is presented in Section 7 with a hardware-in-the-loop validation on the NVIDIA Drive PX2 platform. Conclusions and future work are summarised in Section 8.

2. Related Work

This work deals with an effective scenario- and platform-aware design flow (SPADe) for image-based control systems and its instances for a predictable MPSoC and an industrial platform. The key challenge in designing IBC systems is to deal with the long sensing delay due to compute-intensive image processing. Relevant literature deals with the questions: What are the relevant overall design approaches for such embedded control problems? What are the techniques to deal with long delays in a loop? What are the relevant modeling and analysis techniques?

Design paradigms: Embedded control applications are typically designed based on the *separation of concerns* principle between control theory and embedded systems disciplines [16, 17]. Here, the control engineer operates at the controller-level and designs the controller for a strictly periodic sampling interval with hard deadlines. Naturally, these assumptions impose hard requirements on the task-level and application-level timing bounds of the IBC application. The embedded systems engineer has to then guarantee these bounds at runtime by developing computational models and appropriate scheduling mechanisms. This design philosophy leads to a relatively straightforward design flow based on worst-case considerations that often restrict control performance and leads to significant resource over-dimensioning.

Alternately, platform-based design methods were proposed in literature that emphasize co-design of control and scheduling [18]. Here, the platform resource properties are taken into account while designing the controller. Literature on such platform-aware control design is numerous [16, 19]. Contract-based design [20] is another platform-based design paradigm for cyber-physical systems where the interactions between the control theory and embedded design are defined based on contracts.

From the embedded systems discipline, a system-scenario-based design approach [8] is proposed where different behaviours (scenarios) of an application are explicitly considered to avoid over-dimensioning or sub-optimal performance due to worst-case design. Identifying, characterising and modelling these scenarios and dealing with the run-time scenario transitions is specific for each application and generally not trivial.

Our SPADe approach combines the concepts of the system-scenario-based design and platform-based design methods for image-based control systems into a co-design approach that jointly develops and optimizes the image-processing implementation and the controller design. We classify the dynamic behaviours of the sensing application as scenarios. Though these scenarios occur in some arbitrary and unknown order, each scenario can be individually analysed and optimized using platform-aware design concepts taking into account controller performance for each of the scenarios. The scenarios are then integrated in a switched controller design.

Coping with long sensing delays: Strategies to cope with a long delay can be found in both control theory and embedded systems literature. Control engineers tackle a long delay using advanced state estimation [21], robust design [22], predictive control [23], observer-based [1], and multi-rate sampling [24] methods. These methods heavily rely on the system model and are vulnerable to modelling errors with longer delays. Further, these approaches do not consider platform constraints like resource availability and mapping, or workload variations in image processing [17].

Embedded systems engineers aim to reduce processing latency by parallel implementations of the algorithms using heterogeneous platforms having specialised hardware such as GPUs [25] and FPGAs [26]. Specialised hardware is used for accelerating compute-intensive tasks such as lane detection using a deep learning neural network, e.g. lanenet [27].

Pipelined control [15] is a co-design approach targeting homogeneous multiprocessor implementations. In pipelined control, the control loop is pipelined over multiple cores by creating multiple instances of the sensing algorithm to reduce the sampling period and improve control performance. It does not parallelize the sensing algorithm, it does not take into account workload variations, and it does not easily generalize to heterogeneous platforms.

SPADe is a co-design approach targeting both heterogeneous and homogeneous multiprocessor platforms that explicitly considers workload variations, application parallelism, and platform aspects to deal with long and variable delay.

System modelling: Model-based design [28, 29] approaches focus on designing applications based on abstract models of application and platform such that the implementation is guaranteed to behave with a predictable performance. Numerous models of computation (MoC) are available from literature [9, 30, 31, 32]. SPADe does not depend on a specific MoC. It needs a MoC that can capture the dynamic behaviours (scenarios) of the application, can analyse timing and has support for platform-aware mapping analysis. We choose scenario-aware data flow (SADF) [9] as our MoC as it inherently supports modeling scenarios and has tool support for timing analysis and platform-aware mapping.

3. Embedded image-based control

We consider a setting for an IBC system as shown in Fig. 1. Our sensor is the camera module that captures the image stream. The image stream is then fed to an embedded multiprocessor platform at a fixed frame rate per second (fps), e.g. 30 fps. The tasks in our application - compute-intensive image sensing and processing (S), control computation (C) and actuation (A) - are then mapped to run on this multiprocessor.

3.1. LTI feedback control systems

We consider a linear time-invariant (LTI) feedback control system given by:

$$\begin{aligned} \dot{x}_c(t) &= A_c x_c(t) + B_c u(t), \\ y_c(t) &= C_c x_c(t), \end{aligned} \tag{3.1}$$

where $x_c(t) \in \mathbb{R}^n$ represents the *state*, $y_c(t) \in \mathbb{R}$ represents the *output* to be regulated and $u(t) \in \mathbb{R}$ represents the control *input* of the system at any time $t \in \mathbb{R}_{\geq 0}$. A_c, B_c and C_c represent the system, input and output matrices of appropriate dimensions.

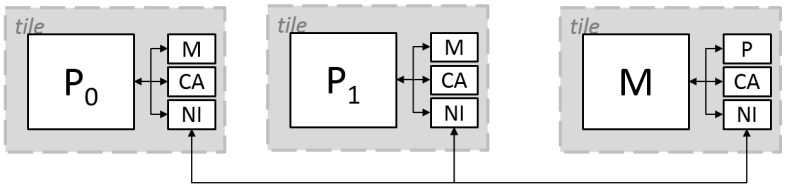


Figure 3: An MPSoC platform with two processor tiles and a memory tile connected through a NoC.

We illustrate our work using the motivating case study of a vision-based lateral control system model, commonly referred to as a lane keeping assist system (LKAS). Our camera sensor captures the image stream at a fixed frame rate per second, e.g., 30 fps. The tasks in our LKAS - compute-intensive image sensing and processing (S), control computation (C), and actuation (A) - need to be mapped to run onto a multiprocessor platform. Quality of Control (QoC) needs to be optimized. The state-space matrices of our LKAS derived from [33] are as follows,

$$A_c = \begin{bmatrix} -10.06 & -12.99 & 0 & 0 & 0 \\ 1.096 & -11.27 & 0 & 0 & 0 \\ -1.000 & -15.00 & 0 & 15 & 0 \\ 0 & -1.000 & 0 & 0 & 15 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \; B_c = \begin{bmatrix} 75.47 \\ 50.14 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \; C_c = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

The five system states are - lateral velocity, yaw rate of the vehicle, lateral deviation from the desired centerline point at look-ahead distance y_L, the angle between the tangent to the road and vehicle orientation, and the curvature of the road at the look-ahead distance. The control input $u(t)$ is the front wheel steering angle δ_f and the output $y_c(t)$ is the look-ahead distance y_L.

3.2. Platforms under consideration

We consider predictable and composable MPSoC platform CompSOC [10] for illustrating the proposed design flow. CompSOC offers a tile-based architecture [34] template (see Fig. 3). Each tile has a processor P_i, memory M, communication assist CA and network interface NI (see Fig. 3). The CompSOC platform offers a configuration with multiprocessors (processor tiles), interconnections through a network-on-chip (NoC), and memories (memory tiles). In the considered setup, each processor tile has a microblaze processor. The memory tile contains an external memory interface, e.g. DDRAM. The NoC provides interconnection between the tiles. The platform is predictable with tight bounds on WCETs of task, and composable so that applications sharing the platform do not interfere with each other. These properties makes the platform suitable for the proposed model-based design flow. A scheduler performs (re)configuration and time-triggered task execution.

The mentioned predictability and composability are usually not offered in an industrial platform. We adapt the SPADe approach for the industrial platform NVIDIA Drive PX2 [35] to demonstrate its applicability in an industrial context. It consists of 2 Tegra Systems-on-Chip (SoCs) that communicate to each other via ethernet. Each Tegra SoC has 2 CPU clusters (see Fig. 4). One cluster contains 4 ARM Cortex A57 cores and the other contains 2 NVIDIA Denver2 cores. The clusters are connected through a high-performance network interconnect. Each of the Tegra SoCs also has 2 Graphical Processing Units (GPUs) - an integrated Pascal GPU (iGPU)

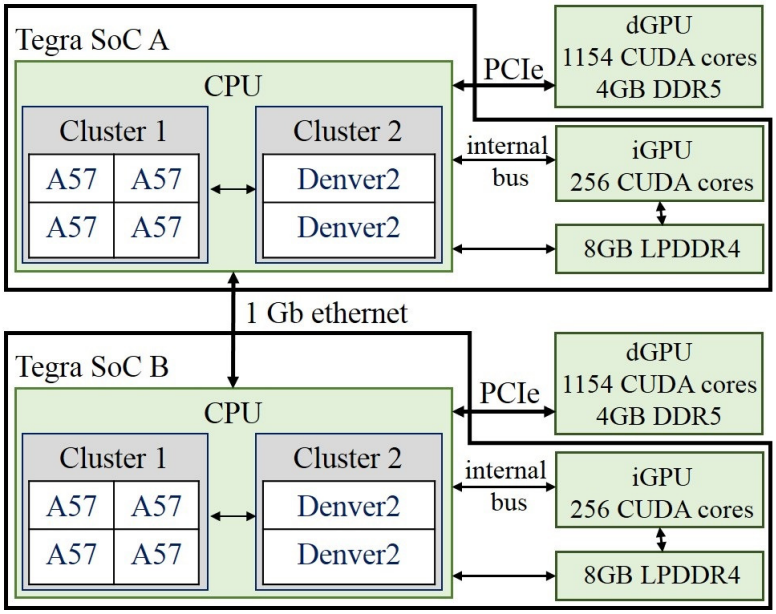


Figure 4: NVIDIA Drive PX2 platform graph structure. LPDDR4 and DDR5 are the memory blocks. Each CPU cluster also has internal instruction and data memory (not shown in the graph).

and a discrete GPU (dGPU) with maximum clock rates of 1.27 and 1.29 GHz respectively. The iGPU has 256 CUDA cores and the dGPU has 1154 CUDA cores. The GPUs are accessed via the respective CPUs in the SoC. The Ubuntu 16.04 LTS OS runs on the CPU platform.

A platform is modelled as a platform graph as shown in Fig. 3 and Fig. 4. A platform allocation determines the resources that are allocated to a task or to an application. Resources that are allocated include: i) number of processors (or part of a processor, e.g. slots in a time-division multiplexing (TDM) frame); ii) type of processors, e.g. GPU, ARM cortex A57, Denver2, microblaze and so on; iii) memory size - for local memory in a tile and/or shared memory, e.g. DDRAM, LPDDR4; and iv) communication bandwidth.

3.3. Embedded implementation

Implementation of an IBC system involves the execution of three sequential tasks: *sensing and processing* (S), *control computation* (C) and *actuation* (A). These tasks repeat; let the start and finish times of the k-th instance be given by $t_s(.)$ and $t_f(.)$, respectively. The execution times of S^k, C^k and A^k (the k-th instance) are given by,

$$e_T^k = t_f(T^k) - t_s(T^k),$$

where $T \in \{S, C, A\}$. The interval between two consecutive executions of sensing tasks S^k and S^{k+1} is then the *sampling period* h^k for the k-th instance. The time interval between the starting time of S^k and finishing time of A^k is the *sensor-to-actuator delay* τ^k for the k-th instance.

$$h^k = t_s(S^{k+1}) - t_s(S^k), \qquad \tau^k = t_f(A^k) - t_s(S^k). \tag{3.2}$$

We consider a time-triggered implementation of tasks $S,\ C$ and A. A camera captures the images at discrete intervals, e.g. 30 fps, and the image frame arrival period f_h is given by,

$$f_h = \frac{1}{\text{frame rate}};\ \text{for 30 fps, } f_h = \frac{1}{30}s = 33.33 \text{ ms}.$$

This means that the sampling period h needs to be an integer multiple of f_h. Sensor processing is followed by control computation and actuation operations which generally take a short and nearly constant time for execution. A sensing operation takes much longer time, i.e.,

$$e_S \gg e_C + e_A$$

where $e_S,\ e_C$ and e_A are the worst-case execution times of sensing and processing, control computation and actuation, introduced above. Moreover, $t_s(C^k) = t_s(S^k) + e_S$ and $t_s(A^k) = t_s(C^k) + e_C$. The total (worst-case) execution time of the control loop is then given by $\tau_t = e_S + e_C + e_A$.

The effective sensor-to-actuator delay τ and sampling period h are then given by,

$$\tau = \tau_t,\ h = \lceil \frac{\tau_t}{f_h} \rceil f_h. \tag{3.3}$$

We assume that the start of sensor data processing is aligned with the camera frame arrival and the actuation is delayed to guarantee constant sensor-to-actuator delay. With sensor-to-actuator delay τ and a zero-order-hold mechanism with sampling period $h \in \mathbb{R}$, $u(t)$ becomes piecewise constant in the intervals $t \in [kh + \tau,\ (k+1)h + \tau]$ for $k \in \mathbb{Z}_{\geq 0}$.

Image-processing workloads may vary, e.g., depending on image content. In Fig. 2, for example, the number of features in an image determines the workload. Each workload scenario s_k is annotated with a pair (h_k, τ_k) that models the sampling period and delay associated with it. A zero-order sample-and-hold approach can then be used to discretize the system based on the workload scenario s_k. Eq. 3.1 can be reformulated as follows:

$$\begin{aligned} x[k+1] &= A_{s_k} x[k] + B_{0,s_k} u[k] + B_{1,s_k} u[k-1], \\ y[k] &= C_c x[k] \end{aligned} \tag{3.4}$$

where,

$$\begin{aligned} A_{s_k} &= e^{A_c h_k}, \\ B_{0,s_k} &= \int_0^{h_k - \tau_k} e^{A_c s} ds \cdot B_c,\ B_{1,s_k} = \int_{h_k - \tau_k}^{h_k} e^{A_c s} ds \cdot B_c \end{aligned} \tag{3.5}$$

In Eq. 3.4, we assume that $u[-1] = 0$ for $k = 0$. We define new system states $z[k] = \begin{bmatrix} x[k] & u[k-1] \end{bmatrix}^T$ with $z[0] = \begin{bmatrix} x[0] & 0 \end{bmatrix}^T$ to obtain a higher-order augmented system as follows to obtain a delay-free state space:

$$z[k+1] = A_{aug,s_k} z[k] + B_{aug,s_k} u[k],\ y[k] = C_{aug} z[k] \tag{3.6}$$

where,

$$A_{aug,s_k} = \begin{bmatrix} A_{s_k} & B_{1,s_k} \\ 0 & 0 \end{bmatrix},\ B_{aug,s_k} = \begin{bmatrix} B_{0,s_k} \\ I \end{bmatrix},\ C_{aug} = \begin{bmatrix} C_c & 0 \end{bmatrix}. \tag{3.7}$$

0 and I represent the zero and identity matrices of appropriate dimensions. A check for controllability [36] is done for this augmented system. If the system is not controllable, controllability decomposition is done to obtain a controllable subsystem. We can apply standard control design techniques for the delay-free state-space model shown in Eq. 3.6.

3.4. Control law and control configurations

In view of the augmented system of Eq. 3.6, we use a *state feedback* controller $u[k]$ of the following form,

$$u[k] = F_{s_k} z[k] + F_{f,s_k} r_{ref} \tag{3.8}$$

where F_{s_k} is the state feedback gain and F_{f,s_k} is the feedforward gain both designed for the workload scenario s_k. r_{ref} is the reference value for the controller. The design of gains can be done with state-of-the-art control design techniques such as linear quadratic regulator (LQR) or pole-placement [36]. A detailed explanation of how we design the gains for our setting is explained in [14]. Note that any other state-of-the-art control design technique can also be used for designing these gains.

For each workload scenario s_k, we then define a *control configuration* χ_{s_k} as a tuple $\chi_{s_k} = (h_{s_k}, \tau_{s_k}, F_{s_k}, F_{f,s_k})$.

3.5. Controller stability

At runtime, the workload scenarios are switching based on the image workload variations and/or platform load. This switching behaviour can lead to system instability. Therefore, we must *guarantee* stability of the overall system while improving Quality of Control (QoC).

Theorem 3.1. (Stability criterion [37]) *Consider A_{aug,s_k} to be discrete-time LTI systems. $V(z) = z^T P z$ is the Common Quadratic Lyapunov Function (CQLF) of the systems A_{aug,s_k} if there exist $P = P^T > 0$, $Q = Q^T > 0$ and P is the simultaneous solution of the discrete-time Lyapunov equations,*

$$A_{aug,s_k}^T P A_{aug,s_k} - P = -Q < 0. \tag{3.9}$$

The existence of a CQLF is a sufficient condition for the stability of a system with switching subsystems.

We transform the stability condition (Eq. 3.9) into Linear Matrix Inequalities (LMIs) to analyse for the existence of a CQLF. The analysis equation, Eq. 3.10, is obtained by performing the following operations: i) substitute A_{aug,s_k} in Eq. 3.9 with $A_{aug,s_k} = A_{aug,s_k} + B_{aug,s_k} * F_{s_k}$, ii) apply Schur complement, and iii) left- and right- multiplication by diag(P^{-1}, I) and set $Q = P^{-1}$.

$$\begin{bmatrix} -Q & QA_*^T + QF_{s_k}^T B_*^T \\ A_*Q + B_*F_{s_k}Q & -Q \end{bmatrix} < 0,\ Q > 0 \tag{3.10}$$

where $A_* = A_{aug,s_k}$, $B_* = B_{aug,s_k}$ for each scenario s_k. If a solution exists, then the switching subsystems are stable. The choice of scenarios need to be modifies if a solution does not exist. A less aggressive mode with poorer performance is usually more likely to meet the stability condition. Failure to guarantee switching stability would result in a classical worst-case based design.

An alternate controller synthesis method has been proposed for this setting using a Markovian jump linear system formulation in [14].

3.6. Control performance: Mean square error (MSE)

The MSE is the mean of the cumulative sum of the squared errors, i.e.:

$$MSE \;=\; \frac{1}{n}\sum_{k=1}^{n}(x[k]-r_{ref})^2$$

where n is the number of observations, $x[k]$ is the value of the k^{th} observation and r_{ref} is the reference value. The MSE quantifies, in essence, how fast the output $y(t)$ reaches the reference r_{ref}. A lower MSE implies a better QoC.

4. Model of Computation for IBC

Our application is modelled using a model of computation (MoC) or a programming model that allows timing analysis. Our model should capture dynamic behaviour and scenario-awareness. This enables us to model and analyse execution time variations that happen at run-time due to either image workload variations and/or platform load. We assume that WCET estimates of task workloads are given for a platform or can be computed.

A MoC is required to compute the parameters relevant for the control design - the sampling period h and the sensor-to-actuator delay τ. However, the challenge now is: How to accurately determine h and τ at design time for a multiprocessor/heterogeneous platform implementation? The choice of binding and scheduling of tasks on the platform determines h and τ.

4.1. Scenario-aware data flow (SADF)

We choose the scenario-aware data flow [9] as the formal MoC for our application as it enables us to: i) model dynamic behaviour, analyse timing, and optimally map application tasks to the platform for maximising the effective utilisation of allocated resources, ii) relate throughput of the data flow graph to the sampling period, and thus combine data flow analysis and mapping with control design parameters and QoC, and iii) to efficiently implement a run-time mechanism that manages necessary dynamic reconfiguration between system scenarios.

A Scenario-aware data flow graph (SADFG) (see Fig. 5) is a tuple SADFG = ($\Sigma,\ \mathcal{F}$), where:

- $\Sigma \;=\; \{s_i \;\mid\; s_i \;=\; (w_i, \mathcal{G}_i),\ w_i \in W\}$ is a set of scenarios being a set of pairs of workload w_i and their corresponding synchronous data flow graphs (SDFGs) $\mathcal{G}_i$.
- The language $\mathcal{F}$ describes a set of infinite scenario sequences represented using ω-regular expressions of scenarios $s_i \in \Sigma$ [38].

An SDFG [39] is a tuple $\mathcal{G} = (\mathrm{A},\ \mathcal{C},\ e,\ r,\ i)$ where A is the set of actors, $\mathcal{C} \subseteq \mathrm{A}^2$ the set of channels, $e \;:\; \mathrm{A} \rightarrow \mathbb{R}_{\geq 0}$ returns for each actor its associated firing delay or execution time, $r \;:\; \mathrm{A} \times \mathcal{C} \rightarrow \mathbb{N}_{>0}$ returns for each actor port its associated rate and $i \;:\; \mathcal{C} \rightarrow \mathbb{N}_0$ returns for each channel its number of initial tokens. Actors of an SDFG may fire, consuming and producing tokens according to the specified rates.

The SADFG for our LKAS IBC application is shown in Fig. 5. The sensing and processing algorithm receives the camera image frames and detects the regions-of-interest (RoID) in the frames.

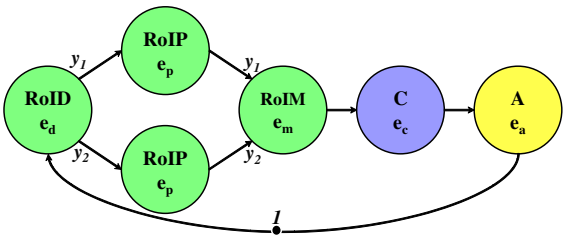


Figure 5: LKAS SADF graph, assuming two allocated processors and hence two RoIP actors.

The detected regions-of-interest (RoI) can be processed in parallel on a multiprocessor platform. The number of allocated processors for our application determines the number of RoI processing (RoIP) actors in our model. In this case, we have two allocated processors and hence two RoIP actors. The total number of RoI detected by RoID determines the workload w_i, i.e., $w_i = y_1 + y_2$. The parameters y_1 and y_2 determine how many RoI need to be allocated to the individual processors and are the rates for the corresponding scenario s_i. Note that the sensor-to-actuator delay and sampling period vary based on the value of y_1 and y_2. After processing the RoI, the data is merged and the controller state (the lateral deviation y_L in our LKAS case study) is computed by the RoI merging (RoIM) task. The control algorithm (C) then computes the controller input $u[k]$ (steering angle δ_f in our LKAS case study) and feeds it to the actuation (A) task.

Each workload w_i is associated with an SDFG $\mathcal{G}_i$. An SDFG instance of Fig. 5 is obtained by assigning values to parameters e_i (the actor execution times) and y_i. E.g., assigning $y_1 = 2,\ y_2 = 3,\ e_d = 5,\ e_p = 10,\ e_m = 7 \times (y_1 + y_2) = 35,\ e_c = 2,\ e_a = 2$ gives the SDFG for a workload of 5 RoI for mapping to two processors. The actors of $\mathcal{G}_i$ are $\mathrm{A}_i = \{$RoID, RoIP, RoIM, C, A$\}$. The channels of $\mathcal{G}_i$, $\mathcal{C}_i$, are shown as dependencies in the figure, and there is one (labelled) initial token in the channel from actor A to RoID. The scenarios are defined based on w_i and the parameters that change for the corresponding $\mathcal{G}_i$ are $y_1,\ y_2$, and $e_m = (y_1 + y_2) \times 7$. All other aspects of the $\mathcal{G}_i$ are the same for all scenarios.

As rates in an SDFG are constant for each firing, it is possible to construct a finite schedule (if it exists) that can be periodically repeated [39]. We call such minimal sequence of firings an *iteration* of the SDFG. This is a sequence of firings that has no net effect on the token distribution in the graph. The numbers of firings of each actor within an iteration constitute the *repetition vector* ρ of an SDFG. For the SADFG graph in Fig. 5, for each scenario w_i, the corresponding SDFG $\mathcal{G}_i$ has repetition vector $\rho_i = \begin{bmatrix} 1 & y_1 & y_2 & 1 & 1 & 1 \end{bmatrix}$, where y_1 and y_2 represent the firing rates of the two actors RoIP shown in Fig. 5. A word from the SADF language $\mathcal{F}$ now specifies a sequence of iterations of the corresponding scenario SDFGs.

The state-of-the-art SADF analysis uses (max, +) algebra [40]. The definitions needed for our analysis are summarised in the following paragraphs. For detailed explanations and analysis methods, the reader is referred to [38].

A time-stamp vector $\boldsymbol{\gamma_0}$ captures the availability times of (labelled) initial tokens. The production times of the labelled final tokens for a scenario s are then captured by Eq. 4.1.

$$\boldsymbol{\gamma_1} = \boldsymbol{G}_s \boldsymbol{\gamma_0} \tag{4.1}$$

We assume that initial and final tokens are the same. $\boldsymbol{G}_s$ is the scenario (or state) matrix of s. For

the scenario SDFG corresponding to 5 RoI, introduced above, $\boldsymbol{\gamma_0} = [0]$ since there is only one labelled initial token. $\boldsymbol{G}_s = [e_d + \max(y_1, y_2) \times e_p + e_m + e_c + e_a] = [74]$ and $\boldsymbol{\gamma_1} = [74]\,[0] = [(74 + 0)] = [74]$.

$\boldsymbol{G}_s$ is used to determine the evolution of any scenario sequence. Labelled final tokens of one scenario are the initial tokens of the next scenario execution. E.g., if s^ω is the infinite repetition of scenario s, then the production times of the labelled tokens after the execution of the k^{th} scenario in the sequence is given by:

$$\boldsymbol{\gamma_k} = \boldsymbol{G}_s \boldsymbol{\gamma_{k-1}} = \boldsymbol{G}_s^k \boldsymbol{\gamma_0}$$

For all $s \in \Sigma$, we can construct $\boldsymbol{G}_s \in \mathbb{R}_{-\infty}^{i(s)\times i(s)}$ [41]. Here, $i(s)$ is the total number of (labelled) initial tokens (in all channels) for scenario s.

Further, we need to analyse the production times of *outputs*, i.e., the relevant information produced, during an execution of a scenario sequence. Let the function $m : \Sigma \to \mathbb{N} \cup \{0\}$ map each scenario to the number of outputs produced in that scenario. The output production times of the scenario sequence s^ω can be computed as,

$$\boldsymbol{p}_k = \boldsymbol{H}_s \boldsymbol{\gamma_k} = \boldsymbol{H}_s \boldsymbol{G}_s^k \boldsymbol{\gamma_0} \tag{4.2}$$

where $\boldsymbol{p}_k$ is the time instance at which the k^{th} output is produced and $\boldsymbol{H}_s \in \mathbb{R}_{-\infty}^{m(s)\times i(s)}$ is the output matrix of the scenario s that captures the relation between the state vector and the production times of $m(s)$ outputs. Note that the first output is $\boldsymbol{p}_0$. The $\boldsymbol{H}_s$ matrices can be computed in a similar way as the state matrices. For the LKAS scenarios, the output is produced by the actor A, meaning that the output production time is equal to the production time of the token on the channel from A to RoID. This means that $\boldsymbol{H}_s = [74]$ and the production time of the first output $\boldsymbol{p}_0 = [74]\,[0] = [74]$ (note that $\boldsymbol{G}_s^0 = I$, the identity matrix).

We quantify the throughput ν of a given scenario sequence of an SADFG by the average number of outputs produced per time unit during the execution of that sequence. The throughput of the SADFG for the scenario sequence $\bar{s}$ is defined as follows,

$$\nu(\bar{s}) = \lim_{n\to\infty} \sup \frac{\sum_{i=1}^{n} m(s_i)}{\|\boldsymbol{\gamma_n}\|} \tag{4.3}$$

where $\|\boldsymbol{\gamma_n}\|$ is equal to the maximum entry in the vector $\boldsymbol{\gamma_n}$. For the infinite execution of the 5 RoI scenario SDFG, the throughput is $\frac{1}{74}$.

For the SADFG models in our SPADe flow we assume the following.

- Throughput is monotonic for our SADFG for different workloads. This is guaranteed by the following:
  1. The set of actors A is the same for all scenarios, i.e. $\mathrm{A}_i = \mathrm{A}_j$, where A_i and A_j are the sets of actors of $\mathcal{G}_i$ and $\mathcal{G}_j$ respectively.
  2. $w_i \leq w_j \implies \forall a \in \mathrm{A}_i,\ e(\mathrm{A}_i) \leq e(\mathrm{A}_j)$.
- The sensing task is not pipelined, i.e., the control is sequential. This is guaranteed by adding a channel with only one initial token from the actuation task to the start of the sensing task in our SADFG (as in the example).

4.2. System mapping and mapping configurations

System mapping refers to the binding of the application (modelled as an SADFG) to the given platform (modelled as a platform graph) allocation. Note that for each workload scenario, we can have multiple binding options on the given platform. The throughput of each of these binding options would be different. We then need to find maximum throughput for a workload scenario, given the platform allocation. The concrete problem is then to find the optimal mapping of a workload scenario to the platform that maximises throughput. Any design flow that does optimal mapping of an application to platform while maximising throughput can be used. We use the SDF3 design flow [42] as it optimises the resource usage, memory load and communication load for mapping, and embeds state-of-the-art throughput analysis techniques.

Optimal mapping of each workload scenario s_i (modelled as an SDFG $\mathcal{G}_i$) to a platform graph generates a binding-aware SDFG $\mathcal{G}_i^b$ with the task execution schedule encoded in it. A *mapping configuration* refers to the binding of a workload scenario on the platform and its execution schedule represented as a binding-aware SDFG.

5. Problem statement

We can now make our problem statement precise. For a given application and a platform allocation, design

1. mapping configurations,
2. controller configurations, and
3. a run-time reconfiguration mechanism,

such that we optimise

- quality-of-control (QoC) and
- resource utilisation.

6. Scenario- and platform-aware design (SPADe)

The SPADe flow comprises the following steps as shown in Fig. 6:

1. identify, model and characterise the frequently occurring workload scenarios that characterise the dynamic behaviour of the image processing in the control loop;
2. find optimal mappings for these scenarios for the given platform allocation;
3. identify optimal system scenarios combining workload and mapping information and taking into account constraints from the control domain, e.g. stability, and from the embedded domain, e.g. camera frame rate;
4. design a controller with high overall QoC and guaranteed stability for the chosen system scenarios; and

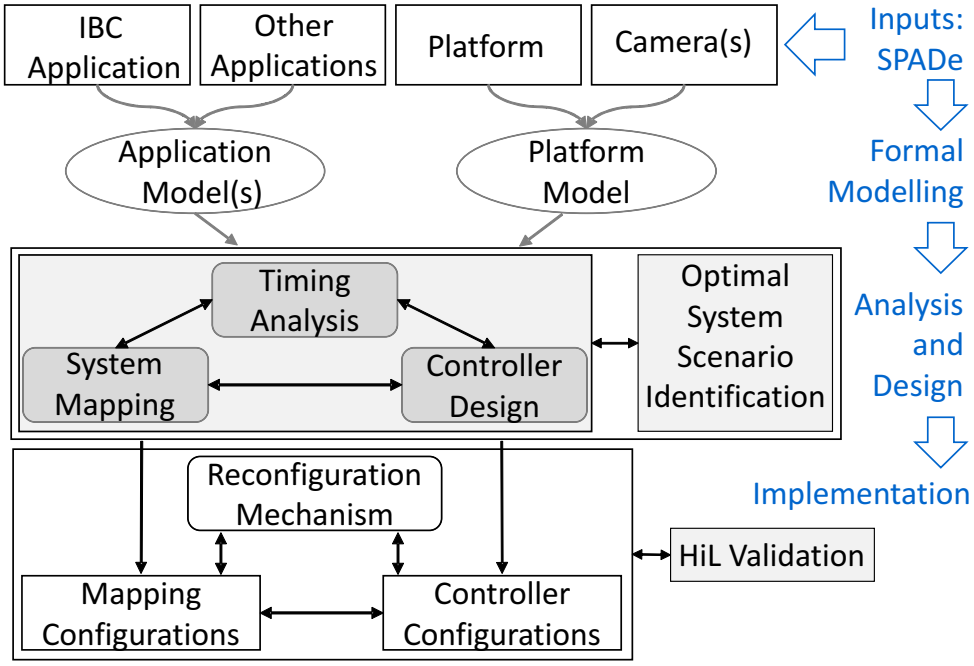


Figure 6: Overview of steps in SPADe flow

5. a runtime reconfiguration mechanism for implementation.

As already stated, we illustrate the SPADe design flow considering the predictability and composibility properties of the CompSOC platform. In the following, we detail the steps in the SPADe design flow.

6.1. SPADe inputs

The inputs to our design flow are details of the IBC application, other applications sharing the platform, given platform allocation for the IBC application and camera characteristics, e.g. fps. These should be compliant with the application and platform models. Note that the details of the other applications sharing the platform are not relevant for a composable platform such as CompSOC.

6.2. Formal modelling: application and platform models

A typical IBC application model of an LKAS derived from [13] is shown in Fig. 5. The details of this model have already been explained in Section 4. Task-level WCET profiling is required to compute the WCETs on the CompSOC platform. The platform is modelled as a platform graph as described in Section 3.2.

6.3. Analysis and Design

6.3.1. System mapping

We first describe the system mapping, i.e., binding and scheduling, of our IBC application model to the platform. Fig. 7 illustrates three workload scenarios and their possible platform

mapping. Fig. 7(a), (c), and (e) model the data flow graphs for different workloads and Fig. 7(b), (d) and (f) show their corresponding mappings on two or three processor tiles P_i. Having more processor tiles means that we can reduce h and τ by parallel execution of the sensing tasks.

System mapping refers to the mapping of application tasks (modelled as an SADF graph) to the platform. An application can have multiple mapping options for a given platform allocation. For example, in Fig. 7(c) and (e), the given platform allocation is two and three processor tiles respectively (visible in the number of RoIP actors) for the same workload (5 RoI).

6.3.2. Timing analysis: relation between data flow analysis and control design

The inverse throughput of the mapped binding-aware SDF graph $\mathcal{G}_i^b$ for scenario sequence s_i^ω gives the sensor-to-actuator delay τ_{s_i}, i.e.

$$\tau_{s_i} = \frac{1}{\nu(s_i^\omega)},\ h_{s_i} = \lceil \frac{\tau_{s_i}}{f_h} \rceil f_h, \tag{6.1}$$

where f_h is the camera frame arrival period.

The timing parameters for the three mapped workload scenarios in Fig. 7 are obtained as follows:

$$\tau_i = e_d + (\max_i^p y_i) \times e_p + e_m + e_c + e_a,\ h_i = \lceil \frac{\tau_i}{f_h} \rceil \times f_h.$$

Assume $f_h = \frac{1}{30}$ s for a camera with 30 fps and $e_m = 7 \times (\sum_i^p y_i)$ where p represents the number of allocated (or given) processors. Cost of communicating data between processors is assumed to be part of the actor execution times e_i; if meaningful, such cost could be made explicit, but for simplicity, we do not do so. For our example shown in Fig. 7:

$$\begin{aligned}
\tau_1 &= 5 + 1 \times 10 + 7 \times (1 + 1) + 2 + 2 = 33\text{ms}, \\
\tau_2 &= 5 + 3 \times 10 + 7 \times (2 + 3) + 2 + 2 = 74\text{ms}, \\
\tau_3 &= 5 + 2 \times 10 + 7 \times (2 + 1 + 2) + 2 + 2 = 64\text{ms},
\end{aligned}$$

and $h_1 = f_h,\ h_2 = 3f_h,\ h_3 = 2f_h$.

6.3.3. Control design

Once we obtain τ_{s_i} and h_{s_i} for mapped workload scenario s_i, they are then used for the discrete-time controller implementation as described in Section 3 and for designing the controller gains. Further, the timing parameters are a part of the control configuration as defined in Section 3.4. Any state-of-the-art control design method can be used for this design.

6.3.4. Optimal system-scenario identification

It is possible for multiple workload scenarios to have the same sampling period due to implementation constraints like platform allocation and camera frame rate. For example, for the workload scenario represented in Fig 7 (a) with (h_1, τ_1), the number of RoI, #RoI $= 2$. However, even for the workload scenario with #RoI $= 1$ mapped to two processors, we would have the same

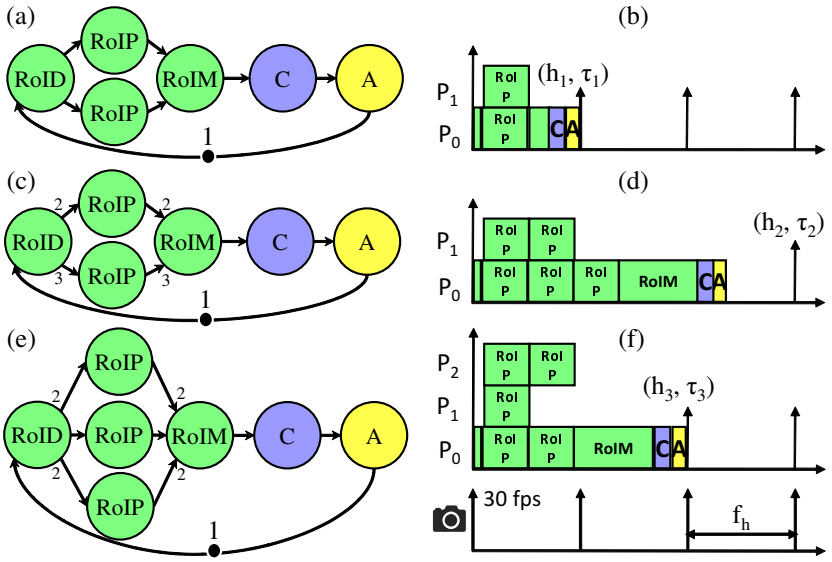


Figure 7: Illustration of workload variations and platform mapping.

timing parameters (h_1, τ_1) since the tasks would have to execute sequentially on one processor. Similarly, for the workload scenario in Fig 7 (c), we would have the same timing parameters for #RoI 5 and 6.

A system scenario s_k abstracts multiple workload scenarios s_i such that for $h_k = n \times f_h$ for some $n > 0$, $(h_k - f_h) < h_i \leq h_k$ and $\tau_i \leq \tau_k$, where $f_h = \frac{1}{30}$ s for a camera frame rate of 30 fps. Only system scenarios are then considered for defining the control configuration and for platform implementation.

6.4. Implementation and runtime reconfiguration mechanism

The optimal system scenarios are identified and their corresponding control and mapping configurations are stored as a look-up table (LUT) in platform memory for runtime implementation. During run-time, for every arriving input image frame, we compute the workload (e.g. through an image pre-processing step) and choose the correct system scenario associated with this workload from the LUT. Controller and mapping configurations of the corresponding system scenario are loaded from the LUT. A scheduler then reconfigures the mapping, the time-triggering of the actuation task and the controller gain parameters based on the chosen system scenario. The overhead cost for this reconfiguration has already been considered in our analysis model as a time cost in the start of sensing task (e.g. along with the actor RoID in Fig. 5).

6.5. Simulation results

The QoC provided by an IBC system depends on the nature of workload variation encountered by the application resulting in different switching sequences. We simulate the LKAS controller performance for various system scenario switching sequences with 2, 4, and 5 RoIs and sampling periods $h_1 = 0.033$s, $h_2 = 0.066$s, and $h_{wc} = 0.100$s for the corresponding system scenarios

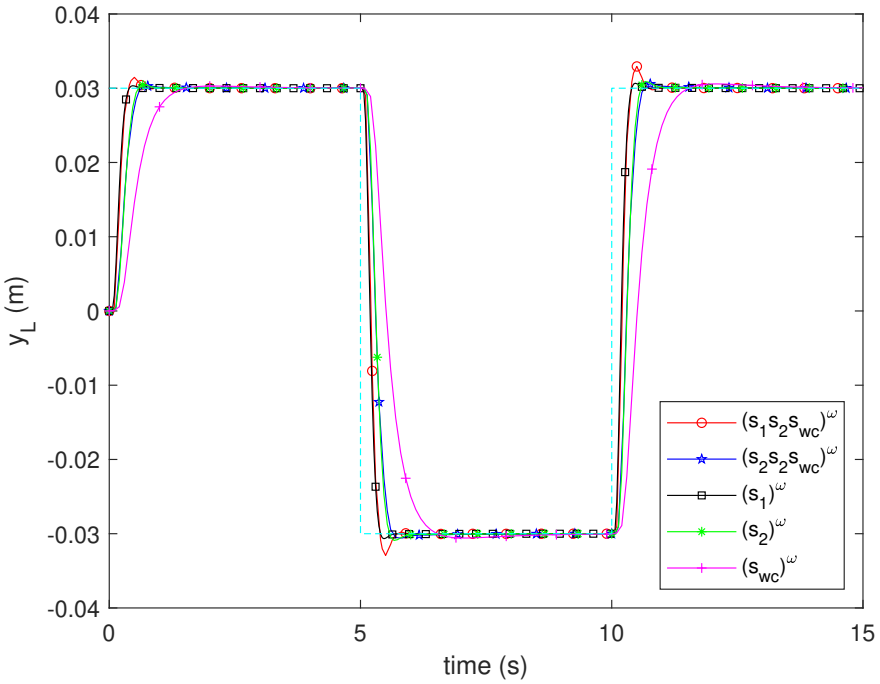


Figure 8: Controller performance: comparison of switching subsystems with worst-case s_{wc}

s_1, s_2, and s_{wc}, respectively, as shown in Fig. 8. We assume that there are 6 RoIs in the worst-case. We observe that our switching designs of SPADe (plot $(s_1 s_2 s_{wc})^\omega$ and $(s_2 s_2 s_{wc})^\omega$) have better QoC (low MSE) than the worst-case sampling period based design (see plot $(s_{wc})^\omega$) in Fig. 8. An example switching sequence is illustrated in Fig. 9(a). We see that the effective resource utilisation for each sampling period is improved (with less idling) with respect to the worst-case based design in Fig. 9(b).

6.6. Comparison with state-of-the-art pipelined control

We compare our SPADe approach with a state-of-the-art pipelined control approach [15]. For fairness in the comparison, we use the same control design technique - LQR with integral action - explained in [15] for SPADe. Further, we consider the same given platform allocation of two processors.

Pipelined control design: We discretize the continuous-time system model in Eq. 3.1 with sensor-to-actuator delay τ and sampling period h to obtain a delayed input system,

$$x((k+1)h) = A_d x(kh) + B_d u(kh - h), \tag{6.2}$$

where A_d, B_d are the discretized state and input matrices respectively. Here, $A_d = e^{A_c h}$ and $B_d = \int_0^h e^{A_c s} B_c ds$. The control input $u(t)$ applied at $t = kh$ uses h time units old sensing information in any sampling interval kh to $(k+1)h$ due to the sensor-to-actuator delay τ. This is reflected in Eq. 6.2 as the delayed input $u(kh - h)$.

For brevity, the pipelined control delay and period is represented as τ and h in this subsection. $\tau = \lceil \frac{\tau_t}{f_h} \rceil f_h$ and $h = \frac{\tau}{\gamma}$, where γ is the number of processing cores. Note that in [15], there is a strict criterion that the sampling period should be an integral multiple of f_h and strictly periodic.

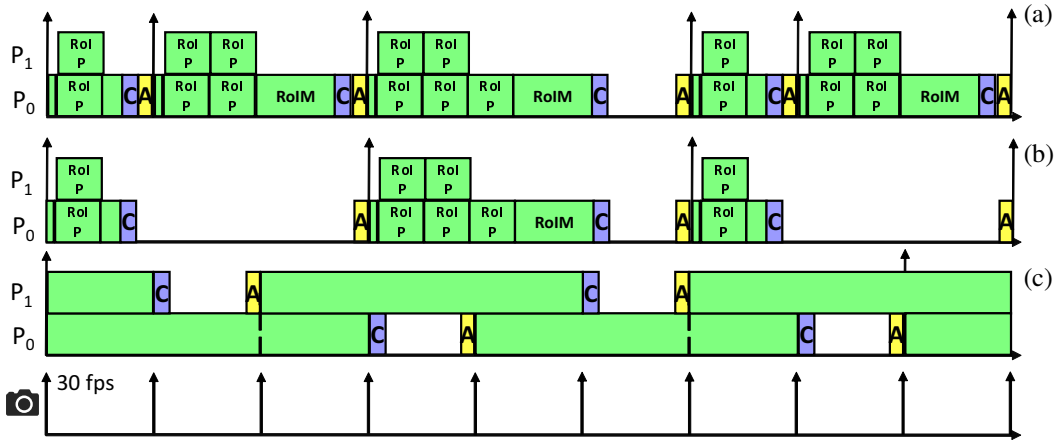


Figure 9: Gantt charts for (a) switching sequence $(s_1 s_2 s_{wc})^\omega$, (b) corresponding worst-case design $(s_{wc})^\omega$, and (c) pipelined control design used for comparison.

As such the τ should be relaxed based on γ. E.g., in our LKAS case, if $\tau_t = 0.084$s, we get $\tau = 0.100$s and $h = 0.050$s for $\gamma = 2$. However, $h = 0.050$s is not an integral multiple of f_h and as such we have to relax τ so that $\tau = 0.100 + f_h = 0.133$s and $h = 0.067$s which is an integral multiple of f_h.

For designing the delayed control input $u(kh - h)$, one design option is to transform the system in Eq. 6.2 into standard non-delayed form and apply any standard control design technique. Towards this, we define a new system state vector $\hat{z}(kh) = \begin{bmatrix} x^T(kh) & u(kh-h) \end{bmatrix}^T$ to obtain a higher-order augmented system in the non-delayed form as follows:

$$\begin{aligned} \hat{z}(kh+h) &= \Phi_d \hat{z}(kh) + \Gamma_d u(kh) \\ y(kh) &= C_d \hat{z}(kh), \end{aligned} \tag{6.3}$$

where $\Phi_d,\ \Gamma_d,\ C_d$ are the augmented discretized matrices such that,

$$\Phi_d = \begin{bmatrix} A_d & B_d & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix},\ \Gamma_d = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ 1 \end{bmatrix}. \tag{6.4}$$

$\mathbf{0}$ and $\boldsymbol{I}$ represent the zero and identity matrices of appropriate dimensions. A check for controllability [36] is done for this augmented system. If the system is not controllable, controllability decomposition is done to obtain a controllable subsystem.

System in Eq. 6.3 is in standard discrete-time form for which standard discrete-time control design technique such as LQR [36] can be used. We use an LQR-based optimal state feedback controller, with integral action for reference tracking, referred to in literature as linear quadratic integral (LQI) control [43, 44]. The state feedback controller is of the form,

$$u(kh) = F^{lqr} \begin{bmatrix} \hat{z}(kh) \\ x_i(kh) \end{bmatrix},\ \text{where } x_i(kh+h) = x_i(kh) + y(kh) - r(kh). \tag{6.5}$$

F^{lqr} is the LQR state feedback gain designed for the state space considering the integral action as given below,

$$\begin{bmatrix} \hat{z}(kh+h) \\ x_i(kh+h) \end{bmatrix} = \begin{bmatrix} \Phi_d & \mathbf{0} \\ C_d & \mathbf{1} \end{bmatrix} \begin{bmatrix} \hat{z}(kh) \\ x_i(kh) \end{bmatrix} + \begin{bmatrix} \Gamma_d \\ \mathbf{0} \end{bmatrix} u(kh). \tag{6.6}$$

This control design replaces the earlier presented LQR control design for each scenario in SPADe. We do so to have a fair comparison between different implementation strategies since the control theory for pipelined control systems considers that τ and h are integral multiples of f_h. However, the controller design approach of SPADe can have any value for τ and hence is more flexible than pipelined control. We adapt the pipelined control design applicable for $\tau \geq h$ to SPADe approach applicable for any $\tau < h$ by modifying Eq. 6.6 with parameters from Eq. 3.6:

$$\begin{bmatrix} \hat{z}(kh+h) \\ x_i(kh+h) \end{bmatrix} = \begin{bmatrix} A_{aug,s_k} & \mathbf{0} \\ C_{aug} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \hat{z}(kh) \\ x_i(kh) \end{bmatrix} + \begin{bmatrix} B_{aug,s_k} \\ \mathbf{0} \end{bmatrix} u(kh). \tag{6.7}$$

Then we design the gain F^{lqr,s_k} for each SPADe scenario s_k using the Eq. 6.5.

Comparison: For pipelined control, the total sensor-to-actuator delay $\tau_t = 5 + 6 \times 10 + 6 \times 7 + 2 + 2 = 111$ms (since each pipe executes sequentially), the effective sensor-to-actuator delay = $\tau = \lceil \frac{\tau_t}{f_h} \rceil f_h = 133.33$ms, and the sampling period $h = \frac{\tau}{2} = 66.67$ms for the given two processing cores. The Gantt chart of the pipelined execution is shown in Fig. 9(c). As mentioned, for SPADe, we use the same pipelined control design approach. For scenario s_{wc}, $\tau_t = 5{+}3{\times}10{+}6{\times}7{+}2{+}2 = 81$ms so that $\tau_{s_{wc}} = 0.100 = h_{wc}$. Similarly, for scenario s_1, $\tau_1 = 0.033 = h_1$ and for scenario s_2, $\tau_2 = 0.067 = h_2$.

The results of the comparison between the pipelined controller with respect to the SPADe approach are shown in Fig. 10. Note that SPADe allows for parallelisation that reduces both sampling period and sensor-to-actuator delay. However, pipelining only reduces the sampling period.

The key observations are:

- The performance of the LQI controllers highly depends on the quality of controller tuning [15]. We observe that the QoC of the pipelined controller is always in the range of QoC between the worst-case design and the SPADe approach. Fig. 10 shows two different tunings of the pipelined controller: plot pipelined$_{\text{bc}}$ is tuned with the same control parameters as scenario s_1 and pipelined$_{\text{wc}}$ is tuned with the same control parameters as scenario s_{wc}.

  If we execute in a frequently occurring scenario, e.g., s_1 (see plot $(s_1^{10} s_{wc})^\omega$ in Fig. 10), then we see that the control performance is better than the pipelined control. In this particular case, arbitrary switching between s_1, s_2, and s_{wc} is unstable. To meaningfully apply the SPADe approach, we should have a frequently occurring scenario during run time, and switching from this frequently occurring scenario to the worst-case should be stable, e.g., based on a dwell time criterion [37] (as shown for plots $(s_1^{10} s_{wc})^\omega$ and $(s_2^{10} s_{wc})^\omega$ in Fig. 10).

- SPADe performs better with a shorter τ when $\tau < h$ and other control tuning parameters are kept the same. When $\tau_1 < \tau_2 < h$, the case with τ_1 will have better performance than τ_2 for the same h. The actual performance improvement further depends on the system dynamics.

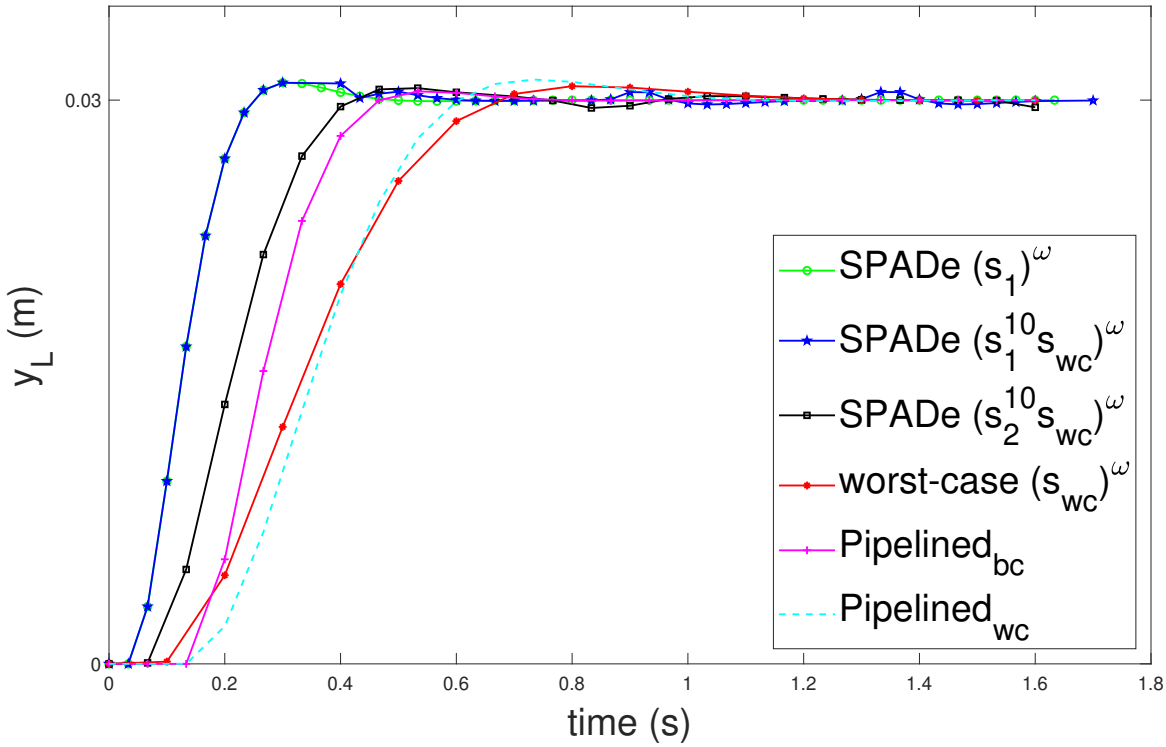


Figure 10: Comparison between pipelined and SPADe approach

SPADe gives prominent advantages when the algorithm structure is known, i.e., the application is a white/gray box and when there is scope for parallelisation. Pipelining works also when the application is a black box and is not dependent on the parallelisation of the algorithm. However, pipelining cannot be used when there are inter-frame dependencies for the algorithm, whereas SPADe is not affected by inter-frame dependencies. Further, SPADe gives better results when the application is executing in its frequently occurring scenario. Pipelining is better suited if the application is frequently executing closer to its worst case. A brief comparison between SPADe and pipelined approaches is illustrated in Table 1.

Table 1: SPADe vs pipelined: applicability criteria and comparison

Criteria	SPADe	Pipelining [15]
Algorithm	should be white/gray box	white/gray/black box
Degree of parallelisation	should be high for better QoC	independent (no parallelism)
Inter-frame dependencies	independent (no pipelining)	should not exist
Workload variations	considered in design	not considered
Platform	independent (applicable for all)	suitable mainly for homogeneous
Restrictions on h	any multiple of f_h	multiple of f_h; strictly periodic
Restrictions on τ	none	multiple of h and $(\gamma \times f_h)$

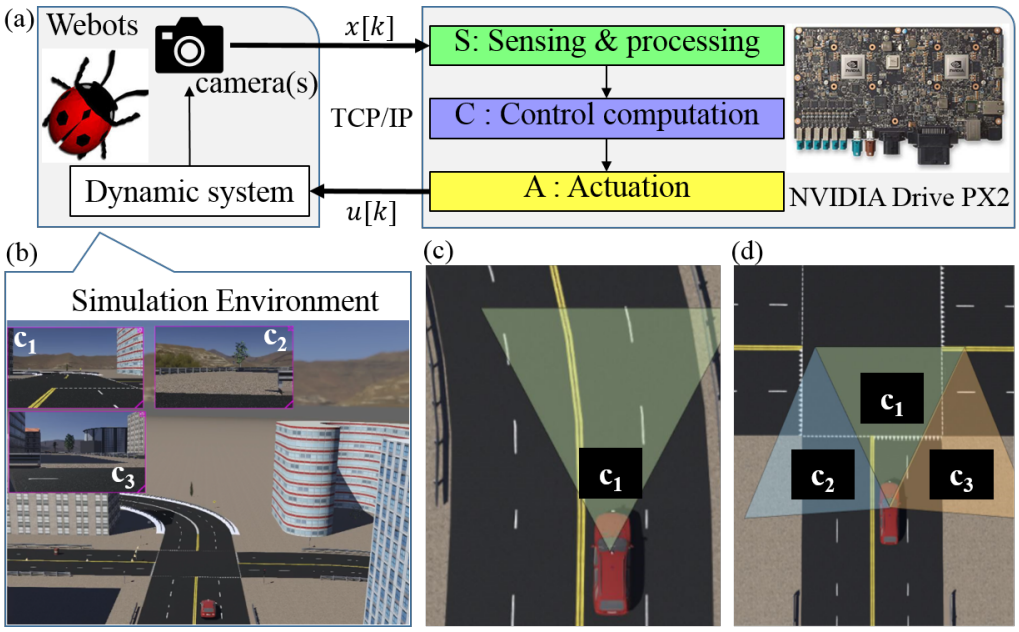


Figure 11: (a) IBC system block diagram and the HiL simulator. (b) a snapshot of the HiL simulation environment in webots. (c) LKAS using single camera. (d) multi-camera LKAS; c_1, c_2, c_3 are the cameras.

7. SPADe for an industrial platform

7.1. Case study: multi-camera LKAS sharing the platform with other applications

We consider a concrete case study of a multi-camera lane keeping assist system (LKAS). The goal of the LKAS is to steer the vehicle autonomously to follow the center line of a lane. Multiple cameras are used since the field-of-view of a single camera is not sufficient to detect the lanes when the vehicle has to make sharp turns, e.g., at a T-junction. Fig. 11(c) and (d) show the two different scenarios in the LKAS system. The first scenario s_1 (see Fig. 11(c) occurs when the vehicle is navigating on a road with no sharp turns. In scenario s_1, only one camera c_1 needs to be active. The second scenario s_2 (see Fig. 11(d) happens when the vehicle needs to take a sharp turn. In this case, all three cameras c_1, c_2 and c_3 need to be active. During runtime the scenarios are detected based on the following: i) when there is a lane detected by camera c_1 and there is no request to make a turn, the LKAS executes in scenario s_1; ii) when there is no lane detected by camera c_1 or there is a request to make a turn, the LKAS executes in scenario s_2. Our multi-camera LKAS is sharing the NVIDIA Drive PX2 platform with two other data-intensive applications - object detection and tracking (ODT) and automatic emergency braking (AEB).

7.2. SPADe input: IBC application

7.2.1. Image sensing and processing (S)

The main stages in the compute-intensive image sensing and processing of an automotive IBC system are the image signal processing (ISP) pipeline, environment perception and application-specific rendering (if required) (shown in Fig. 12(a)). The ISP pipeline is generic for all IBC

applications. Environment perception involves application-specific preprocessing, feature extraction and inference. Rendering refers to the display of relevant information on the dashboard or screen of the vehicle and is application-specific. Below, we explain these stages in detail for our LKAS system case study.

Image signal processing (ISP) pipeline The NVIDIA Drive PX2 comes with a Tegra configurable ISP hardware and supports different image types - CUDA, OpenGL, NvMedia - and different pixel formats - RAW, grayscale, RGB, Red Clear Clear Blue (RCCB), RGB alpha (RGBA), YUV. NvMedia is an NVIDIA proprietary framework which uses dedicated hardware blocks on the Tegra SoCs for faster image processing. Algorithmic analysis of a closed-source proprietary ISP pipeline is not possible. The stages common to generic ISP pipelines are explained in [45]. For our LKAS, the GMSL camera [35] captures the image frame at a fixed frame rate, 30 fps. Each frame then goes through the closed-source ISP pipeline to obtain an image in ≪NvMedia, YUV≫ format.

Perception The perception stage performs a set of application-specific preprocessing, feature extraction and control state computation steps on the image obtained from the ISP.

The preprocessing step in LKAS (shown in Fig. 12(a)) involves converting the image in ≪NvMedia, YUV≫ format to the ≪CUDA, RGBA≫ and ≪OpenGL, RGBA≫ image type and pixel formats. Closed-source functions 'image streamer' and 'format conversion' from NVIDIA perform the image type conversions and pixel format conversions respectively. The ≪CUDA, RGBA≫ format is used for applications that use GPUs and ≪OpenGL, RGBA≫ for rendering.

The features to extract are application-specific. The LKAS extracts the lanes from the image using the NVIDIA proprietary (pre-trained) high-precision DNN *lanenet* [27] that enables pixel-level lane detection. Lanenet executes on the GPU and its input is a ≪CUDA, RGBA≫ image. The output of Lanenet is the position values of all the lane containing pixels, i.e., a set of polyline values in the pixel domain.

Finally, the lateral deviation of the vehicle from the center of the lane is derived. A homography transformation matrix [46] is computed at design time. This matrix is stored in the platform memory and is used at runtime to compute the position values of the detected polylines from Lanenet. The left and right lane polylines are then fit to a second degree polynomial. For a given look-ahead distance, the center of the lane is derived using these polynomials while the center of the image gives the vehicle's current position. Using these, the lateral deviation is calculated at the look-ahead distance. The homography transformation at runtime needs to be done only for the identified lane pixels.

Rendering For LKAS, the rendered image consists of the pre-processed image captured by the camera in ≪OpenGL, RGBA≫ format superimposed with the polylines detected by Lanenet. The rendering step is not important for the correct functioning of LKAS. Rendering is used for debugging and often provided as an add-on for automotive customers for visual pleasure.

7.2.2. Control computation (C) and actuation (A) tasks

The default **scenario $\mathbf{s_1}$** persists when there is always a lane detected in the image captured by the camera c_1 and when there is no request to take a turn, e.g. at a junction. For this scenario, the LKAS controller explained in Section 3 is used. **scenario $\mathbf{s_2}$** occurs when there is no lane detected by camera c_1 or when there is a request to take a sharp turn. Here the control computation is a

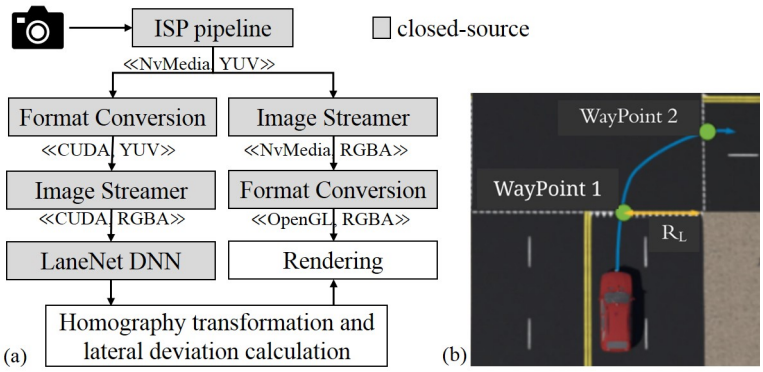
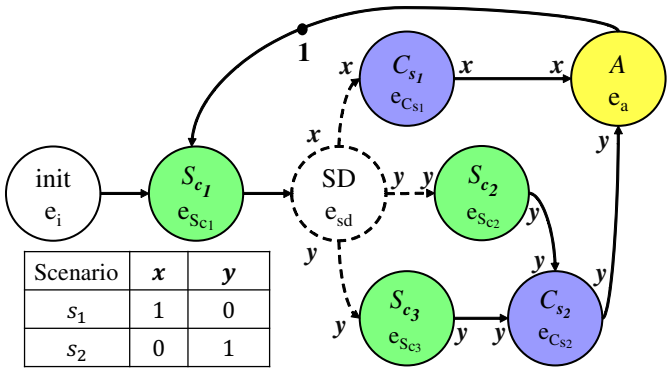


Figure 12: (a) The block diagram of image sensing and processing task S. (b) Path planning for scenario s_2.



Scenario	x	y
s_1	1	0
s_2	0	1

Figure 13: Multi-camera LKAS SADF graph.

standard path planning algorithm. The direction of the turn is user input or determined arbitrarily if lanes are detected on both c_2 and c_3. If no lanes are detected in any of the cameras, AEB is activated. Actuation task A actuates the vehicle steering to the desired value communicated to it by the control computation task.

7.3. Formal modelling

The application is modelled as a data flow graph (see Fig. 13 and Section 4). The applications sharing the platform act as load and the platform is modelled as a platform graph (see Fig. 4) using the available information [35] (see Section 3).

The LKAS has two application scenarios. The *init* actor models platform initialisation. S_{c_1}, S_{c_2} and S_{c_3} actors model the image sensing and processing tasks for cameras c_1, c_2 and c_3. Each sensing task has the internal structure shown in Fig. 12(a). C_{s_1} and C_{s_2} model the control computations for scenarios s_1 and s_2. The actuation task is modelled by actor A. The scenario detector actor SD determines which scenario the application runs in. e_i, $e_{S_{c_1}}$, e_{sd}, $e_{C_{s_1}}$, $e_{S_{c_2}}$, $e_{S_{c_3}}$, $e_{C_{s_2}}$ and e_a are the execution times of the corresponding actors.

Note that the workload for this case is defined by the combination of application scenarios, non-predictable timing behaviour of the closed-source industrial platform and the platform load. Each platform load condition is abstracted as a variant (see Table 2) for a systemic analysis. The

parameters which change based on the workload are x, y shown in Fig. 13 (due to the application scenarios) and the execution times of all actors (due to the platform load).

7.4. Analysis and Design

7.4.1. System mapping and mapping configurations

The tasks/actors need to be mapped to the Tegra SoC resources - CPU, iGPU, dGPU and memory, where CPU refers to the A57 and denver2 cores. Note that there are two Tegra SoCs in the NVIDIA Drive PX2 platform. The options available for the developer when mapping to the PX2 platform are limited. Priorities for tasks can only be assigned on CPU resources. The tasks mapped to the GPUs are executed by the proprietary NVIDIA scheduler and the execution times for such tasks vary the most due to the (unpredictable) scheduler [47].

The mapping configuration does not include the schedule of tasks mapped to GPUs as it is not controllable. Note that init, the homography computation step of S, SD, C and A are always mapped to CPUs and the other steps of S are always mapped to GPUs. The GPUs can only be accessed through the CPUs in the same SoC by a blocking call.

7.4.2. Profiling and timing analysis

Platform-aware profiling is a crucial step in this instance of the SPADe flow. Since there are closed-source functions in the application and a non-real-time Ubuntu OS, the WCETs of tasks in the application are difficult to predict. The WCET of tasks depends on three factors: scenario of the IBC application, choice of mapping, and the load on the platform due to the shared applications. For our case study, we consider two other applications - object detection and tracking (ODT) and automatic emergency braking (AEB) sharing the platform. Both applications take camera images as input.

We define variants $v.i$ to characterise and abstract multiple workload scenarios. The variants we consider based on our mapping choice and platform load are defined in Table 2. The mapping is characterised based on mapping to iGPU and dGPU as preliminary experiments show that compute-intensive imaging tasks perform better on GPUs. Tasks mapped to CPUs take less than 5% of the overall WCET and are not explicitly considered. The platform load AEB and ODT denote the mapping of these applications to the same GPU as the LKAS. The platform load ODTs denote the mapping of ODT and LKAS to multiple GPUs (of the same type) so that there is task sharing between GPUs. This happens in NVIDIA by assigning just the type of GPU to be mapped for the applications and the proprietary scheduler allocates tasks between multiple GPUs. This can be observed by analysing the GPU utilisation (explained in Section 7.7).

Note that due to the closed-source GPU scheduler of NVIDIA, the workload due to the application scenarios and the platform load conditions at runtime cannot be distinguished. Thus, the abstraction as variants is a means to enable the optimal system-scenario identification and runtime reconfiguration(explained in Section 7.5).

For profiling, a database of around 200 images (captured by the GMSL camera) are identified with varying image workload. Considering image workload variations is important since they affect the WCET analysis. The image for the minimal workload has no lane markings and no other vehicles on the road; for the maximal workload it contains three lane markings and other vehicles. For each variant, each image from the database is run on the PX2 for 10000 iterations.

Table 2: Characteristics of variants based on mapping choice and platform load conditions

variants	v.1	v.2	v.3	v.4	v.5	v.6	v.7	v.8	v.9	v.10
mapping	dGPU	iGPU	dGPU	iGPU	dGPU	iGPU	dGPU	iGPU	dGPU	iGPU
load	-	-	AEB	AEB	ODT	ODT	AEB, ODT	AEB, ODT	ODTs	ODTs

ODTs: Object detection and tracking with task sharing between GPUs

The worst-case sensor-to-actuator delay τ_{wc} and sampling period h_{wc} is computed as explained in Section 6.3.2. The execution times of each task are profiled over all the variants and the maximum value is taken as the estimated WCET of the corresponding task. This WCET estimate is used in the application SADF model. τ_{wc} and h_{wc} are then computed. Note that though this worst-case rarely happens, it is needed to guarantee stability of the IBC system. Similarly, for each variant $v.i$, the third quartile values of the measured execution times of each task (profiled for the corresponding variant) is used to compute τ_i and h_i. We thus avoid the measured WCETs for the majority of the analyzed workload scenarios to avoid overly pessimistic model predictions.

7.4.3. Controller design

The controller for scenario s_1 is designed as explained in Section 3. The standard linear quadratic regulator control is used to design the state feedback gain F_i and the feed forward gain $F_{f,i}$ for each variant v_i. The control configuration χ_i is then defined as a tuple $\chi_i = (\tau_i, h_i, F_i, F_{f,i})$. For each version, only χ_i needs to be stored in the memory during implementation. The stability of this switched system is analysed by deriving linear matrix inequalities (LMIs) that check for the existence of a common quadratic Lyapunov function (CQLF) (see Section 7.4.3).

For scenario s_2, the path planning algorithm identifies two waypoints once the direction to turn is determined (illustrated in Fig. 12(b) for a 90 degree turn). Waypoint 1 is the centre of the lane from where the vehicle has to start turning and Waypoint 2 is the centre of the lane after the turn, from where we expect scenario s_1 (see Fig. 12(b)). This can be predicted based on the turning radius R_L. The steering angle $\delta_f = atan\left(\frac{L_{wb}}{R_L}\right)$, where L_{wb} is the wheelbase of the vehicle. This steering angle is constantly applied from Waypoint 1 until the vehicle reaches Waypoint 2 and then task S repeats. Only L_{wb} needs to be stored in memory for scenario s_2.

7.5. System-scenario identification, implementation and runtime reconfiguration mechanism

A system scenario abstracts multiple variants with the same sampling period and optimal system scenarios are identified as explained in Section 6.3.4. The control and mapping configurations of the variants and their relation to system scenarios are stored as a LUT in platform memory for runtime implementation. During runtime, we keep track of the start and finishing time of S, i.e. the sensing delay, to check for which system scenario we need to execute from the LUT. After identifying the system scenario, we load the corresponding control configuration χ_i and execute C. The mapping configuration is then loaded for the subsequent arriving frame. Note that even though control configurations are loaded every frame, mapping configurations cannot be loaded until after the system scenario identification is completed and as such there is a delay in loading mapping configuration by one frame. The classification as variants is thus essential in the identification of system scenario at runtime as the scenario identification at runtime is dependent on

Table 3: Bounded τ_i and h_i of Pareto-optimal system configurations

variants	v.1	v.2	v.3	v.4	v.5	v.6	v.7	v.8	v.9	v.10
τ_i in s	0.028	0.042	0.034	0.066	0.040	0.069	0.051	0.100	0.038	0.045
h_i in s	1/30	2/30	2/30	2/30	2/30	3/30	2/30	3/30	2/30	2/30

the current mapping as well. An LQR controller designed for the worst-case (τ_{wc}, h_{wc}) and its corresponding control configuration χ_{wc} is also stored in the memory as the worst-case system scenario. At runtime, system scenarios are switching (as explained in 7.4.3) based on the load and/or mapping choices.

7.6. Design-space exploration at design time: QoC vs utilisation trade-offs

The QoC is defined as the inverse of MSE (defined in Section 3.6) so that a lower MSE means a better QoC. The utilisation at design-time is defined based on the estimated time spent by the application in GPU kernel calls. We use this definition for the utilisation metric as this could be computed in an actual implementation as well. A DSE to obtain different system configurations is performed for each of the variants defined in Table 2 by choosing different mapping options for S, C and A tasks. Note that the task mappings are allowed to span over two Tegra SoCs and the subtasks of S (shown in Fig. 12(a)) can also be mapped to separate GPUs. Pareto-optimal system configurations are then identified for each variant through Pareto optimisation.

The τ_i and h_i from these Pareto-optimal configurations for each variant (see Table 3) are then considered for system-scenario identification. The $v.i$ in Fig. 14 correspond to the predicted design points using our design flow. In Fig. 14, the MSE for $v.8$ with the largest τ_i among different variants is the poorest. The MSE for different variants tends to aggregate based on the h_i. System scenarios can then be identified based on the requirements, e.g. if QoC is the only criterion, we can select variants $v.1,\ v.7$, and the worst case as system scenarios.

The pessimistic τ_{wc} and h_{wc} are estimated as explained in Section 6.3.2 to be 0.150 s and 5/30 s respectively. Note that h_7 is only 2/30 s and the identified worst-case variant $v.8$ has $h_8 = 0.100$ s. This means that a control design for h_{wc} would see a much worse MSE than any of the variants.

7.7. Hardware-in-the-loop validation using NVIDIA Drive PX2

A design-time analysis alone is insufficient as the runtime behaviour of an industrial platform cannot be predicted. We implement the 10 different variants mentioned in Table 2 using a hardware-in-the-loop (HiL) simulator for LKAS (shown in Fig. 11) and compare its performance with the design-time analysis. Our HiL simulator uses webots [48] as the physics simulation engine and interacts with NVIDIA Drive PX2 using TCP/IP. An initial simulator for a single camera LKAS using V-REP was introduced in [49, 50]. We extend this framework using webots for multi-camera LKAS with support for turning at a junction (or at user input).

The performance metrics we consider are MSE (explained in Section 3.11) and GPU utilisation. *GPU utilisation* is measured by the proprietary NVIDIA Nsight software [51]. GPU utilisation gives the measure of the time spent by the application in GPU kernel calls. For compute-intensive image-based applications sharing the platform, minimising the utilisation is better.

Design-time analysis vs HiL implementation: The $v.i$ in Fig. 14 correspond to the predicted design points using our design flow and the $v.i'$ correspond to the design points obtained from

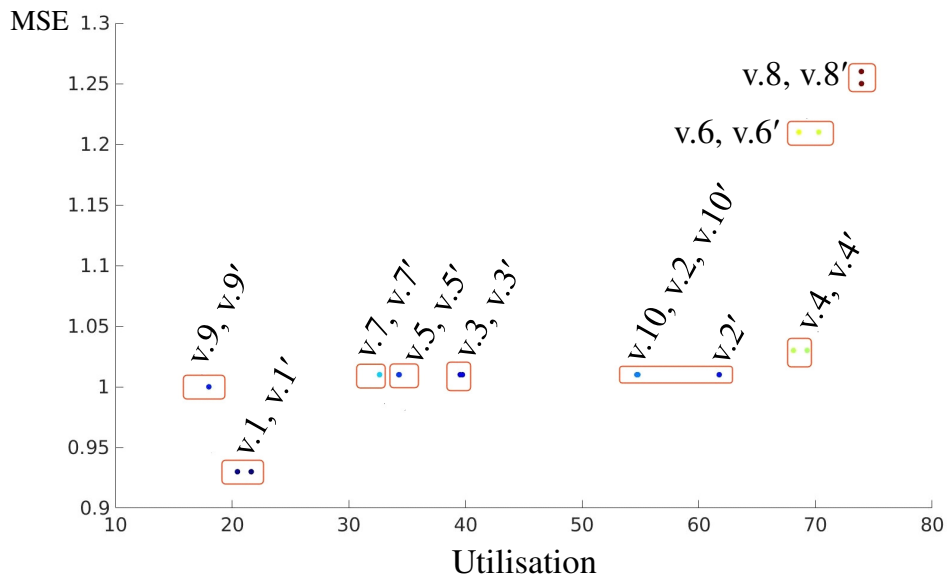


Figure 14: Validating the design time Pareto-optimal system configurations for each variant $v.i$ with the corresponding HiL implementation $v.i'$.

actual implementation using the HiL simulator. Even though the numbers vary between our design flow prediction and actual implementation, the trends we observe for the different variants are the same. Recall that we used measured third quartile execution times instead of WCET in our models. At run time, when we encounter the WCET or any violation of the (τ_i, h_i) for $v.i$, we execute the worst-case controller designed for (τ_{wc}, h_{wc}). At runtime, a switched controller considering the different variants has a much better MSE than the worst-case as there is no aggressive switching, i.e. once we are running in a particular variant, the runtime situation persists for some time. Notice that the QoC improves at runtime since the controller executes in the frequently occurring system scenario.

8. Conclusion

We presented a structured IBC (co-)design flow that considers sensing-application parallelism, workload variations, platform settings, and control parameters for an efficient design and implementation of an IBC system. Our scenario- and platform-aware design (SPADe) approach optimises control quality and maximises the effective resource utilisation for a given platform allocation. We demonstrate the applicability of SPADe for both a predictable multiprocessor platform and for an industrial platform. Though application timing is difficult to predict in industrial platforms, we show that we can leverage existing predictable dataflow model-based design methods by carefully co-designing the sensing implementation and the (switched) controller design using system scenarios. Future work involves a joint Pareto optimisation of multiple (IBC) applications sharing a platform.

Acknowledgment

This work has partially received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under grant agreement no 674875 (oCPS) and the Electronic Component Systems for European Leadership (ECSEL) Joint Undertaking under grant agreement no 783162 (FitOptiVis).

References

[1] E. van Horssen, Data-intensive feedback control: switched systems analysis and design, Ph.D. thesis, Eindhoven University of Technology (2018).

[2] P. Corke, Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised, Vol. 118, Springer, 2017.

[3] J. Elfring, R. Appeldoorn, S. van den Dries, M. Kwakkernaat, Effective world modeling: Multisensor data fusion methodology for automated driving, Sensors 16 (10) (2016) 1668.

[4] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, M. H. Ang, Perception, planning, control, and coordination for autonomous vehicles, Machines 5 (1) (2017) 6.

[5] K. Bengler, K. Dietmayer, B. Farber, M. Maurer, C. Stiller, H. Winner, Three decades of driver assistance systems: Review and future perspectives, IEEE Intelligent Transportation Systems Magazine 6 (4) (2014) 6–22.

[6] FEI, An introduction to electron microscopy, ISBN:978-0-578-06276-1.

[7] I. Chakraborty, S. S. Mehta, J. W. Curtis, W. E. Dixon, Compensating for time-varying input and state delays inherent to image-based control systems, in: American Control Conference (ACC), 2016, pp. 78–83.

[8] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, et al., System-scenario-based design of dynamic embedded systems, ACM Transactions on Design Automation of Electronic Systems (TODAES) 14 (1) (2009) 3.

[9] B. D. Theelen, M. C. Geilen, T. Basten, J. P. Voeten, S. V. Gheorghita, S. Stuijk, A scenario-aware data flow model for combined long-run average and worst-case performance analysis, in: Formal Methods and Models for Co-Design (MEMOCODE), 2006, pp. 185–194.

[10] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, CoMPSoC: A template for composable and predictable multi-processor system on chips, TODAES 14 (1) (2009) 2.

[11] S. A. Edwards, E. A. Lee, The case for the precision timed (PRET) machine, in: Design Automation Conference (DAC), IEEE, 2007, pp. 264–265.

[12] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, R. Schiffelers, Robustness analysis of multiprocessor schedules, in: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), IEEE, 2014, pp. 9–17.

[13] S. Mohamed, D. Zhu, D. Goswami, T. Basten, Optimising quality-of-control for data-intensive multiprocessor image-based control systems considering workload variations, in: 21st Euromicro Conference on Digital System Design (DSD), 2018, pp. 320–327.

[14] S. Mohamed, A. U. Awan, D. Goswami, T. Basten, Designing image-based control systems considering workload variations, in: 58th IEEE Conference on Decision and Control (CDC), 2019.

[15] R. M. Sánchez, J. Valencia, S. Stuijk, D. Goswami, T. Basten, Designing a controller with image-based pipelined sensing and additive uncertainties, TCPS 3 (3) (2019) 33:1–33:26.

[16] K.-E. Arzén, A. Cervin, J. Eker, L. Sha, An introduction to control and scheduling co-design, in: Proceedings of the 39th IEEE Conference on Decision and Control (CDC), Vol. 5, IEEE, 2000, pp. 4865–4870.

[17] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, M. Wolf, Future automotive systems design: research challenges and opportunities: special session, in: CODES+ISSS, 2018.

[18] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, K.-E. Arzen, How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime, IEEE control systems magazine 23 (3) (2003) 16–30.

[19] D. Goswami, A. Masrur, R. Schneider, C. J. Xue, S. Chakraborty, Multirate controller design for resource- and schedule-constrained automotive ecus, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2013, pp. 1123–1126.
[20] A. Sangiovanni-Vincentelli, W. Damm, R. Passerone, Taming dr. frankenstein: Contract-based design for cyber-physical systems, European journal of control 18 (3) (2012) 217–238.
[21] Y. Wang, B. M. Nguyen, H. Fujimoto, Y. Hori, Multirate estimation and control of body slip angle for electric vehicles based on onboard vision system, IEEE Transactions on Industrial Electronics 61 (2) (2014) 1133–1143.
[22] A. Kawamura, K. Tahara, R. Kurazume, T. Hasegawa, Robust visual servoing for object manipulation with large time-delays of visual information, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2012, pp. 4797–4803.
[23] G. Macesanu, V. Comnac, F. Moldoveanu, S. M. Grigorescu, A time-delay control approach for a stereo vision based human-machine interaction system, Journal of Intelligent & Robotic Systems 76 (2) (2014) 297–313.
[24] H. Fujimoto, Visual servoing of 6 dof manipulator by multirate control with depth identification, in: 42nd IEEE International Conference on Decision and Control, Vol. 5, IEEE, 2003, pp. 5408–5413.
[25] R. Agrawal, S. Gupta, J. Mukherjee, R. K. Layek, A gpu based real-time cuda implementation for obtaining visual saliency, in: Proceedings of the 2014 Indian Conference on Computer Vision Graphics and Image Processing, ACM, 2014, p. 1.
[26] S. Kestur, M. S. Park, J. Sabarad, D. Dantara, V. Narayanan, Y. Chen, D. Khosla, Emulating mammalian vision on reconfigurable hardware, in: IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2012, pp. 141–148.
[27] Z. Wang, W. Ren, Q. Qiu, LaneNet: Real-Time Lane Detection Networks for Autonomous Driving, arXiv preprint arXiv:1807.01726.
[28] E. A. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on computer-aided design of integrated circuits and systems 17 (12) (1998) 1217–1229.
[29] L. Thiele, S. Chakraborty, M. Naedele, Real-time calculus for scheduling hard real-time systems, in: IEEE International Symposium on Circuits and Systems, Vol. 4, IEEE, 2000, pp. 101–104.
[30] S. Stuijk, M. Geilen, T. Basten, A predictable multiprocessor design flow for streaming applications with dynamic behaviour, in: DSD, IEEE, 2010, pp. 548–555.
[31] S. Chakraborty, S. Künzli, L. Thiele, A general framework for analysing system properties in platform-based embedded system designs, in: Design, Automation and Test in Europe Conference and Exposition (DATE), 3-7 March, Munich, Germany, 2003, pp. 10190–10195.
[32] R. I. Davis, A. Burns, R. J. Bril, J. J. Lukkien, Controller area network (CAN) schedulability analysis: Refuted, revisited and revised, Real-Time Systems 35 (3) (2007) 239–272.
[33] J. Kosecka, R. Blasi, C. Taylor, J. Malik, Vision-based lateral control of vehicles, in: Proceedings of Conference on Intelligent Transportation Systems, IEEE, 1997, pp. 900–905.
[34] S. Stuijk, Predictable mapping of streaming applications on multiprocessors, Ph.D. thesis, Eindhoven University of Technology (2007).
[35] NVIDIA DRIVE: Scalable AI Platform for Autonomous Driving, NVIDIA, 2017.
URL https://www.nvidia.com/en-us/self-driving-cars/drive-platform/
[36] R. C. Dorf, R. H. Bishop, Modern control systems, Pearson, 2011.
[37] Z. Sun, S. S. Ge, Stability theory of switched dynamical systems, Springer Science & Business Media, 2011.
[38] H. Alizadeh Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, T. Basten, Scalable analysis for multi-scale dataflow models, ACM Transactions on Embedded Computing Systems (TECS) 17 (4) (2018) 80.
[39] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.
[40] F. Baccelli, G. Cohen, G. J. Olsder, J.-P. Quadrat, Synchronization and linearity: an algebra for discrete event systems, John Wiley & Sons Ltd.
[41] F. Siyoum, M. Geilen, H. Corporaal, Symbolic analysis of dataflow applications mapped onto shared heterogeneous resources, in: Proceedings of the 51st Annual Design Automation Conference, ACM, 2014, pp. 1–6.
[42] S. Stuijk, M. Geilen, T. Basten, SDFˆ3: SDF for free, in: ACSD, IEEE, 2006, pp. 276–278.
[43] K. Zhou, J. C. Doyle, K. Glover, et al., Robust and optimal control, Vol. 40, Prentice hall New Jersey, 1996.
[44] P. C. Young, J. Willems, An approach to the linear multivariable servomechanism problem, International journal

of control 15 (5) (1972) 961–979.
[45] M. Buckler, S. Jayasuriya, A. Sampson, Reconfiguring the imaging pipeline for computer vision, in: ICCV, 2017.
[46] R. Hartley, A. Zisserman, Multiple view geometry in computer vision, Cambridge university press, 2003.
[47] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, F. D. Smith, Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems, in: ECRTS 2018, pp. 20:1–20:21. doi:10.4230/LIPIcs.ECRTS.2018.20.
[48] O. Michel, Cyberbotics ltd. webots™: professional mobile robot simulation, IJARS 1 (1) (2004) 5.
[49] S. Mohamed, S. De, K. Bimpisidis, V. Nathan, D. Goswami, H. Corporaal, T. Basten, IMACS: a framework for performance evaluation of image approximation in a closed-loop system, in: 8th Mediterranean Conference on Embedded Computing (MECO), IEEE, 2019, pp. 1–4.
[50] S. De, S. Mohamed, K. Bimpisidis, D. Goswami, H. Corporaal, T. Basten, Approximation trade offs in an image-based control system, in: Design, Automation and Test in Europe (DATE), 2020.
[51] NVIDIA Nsight, V. S. Edition, 3.0 user guide, NVIDIA Corporation.