Optimizing multiprocessor image-based control through pipelining and parallelism

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 3, 2021, accepted July 27, 2021, date of publication August 6, 2021, date of current version August 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3103051

Optimising Multiprocessor Image-Based Control Through Pipelining and Parallelism

SAJID MOHAMED, (Member, IEEE), DIP GOSWAMI, (Member, IEEE), SAYANDIP DE, (Graduate Student Member, IEEE), AND TWAN BASTEN, (Senior Member, IEEE)
Electronic Systems Group, Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands

Corresponding author: Sajid Mohamed (s.mohamed@tue.nl)

This work was supported in part by the FitOptiVis Project and by the COMP4DRONES Project, both funded by the ECSEL Joint Undertaking under Grant numbers H2020-ECSEL-2017-2-783162 and H2020-ECSEL-2018-826610, respectively.

ABSTRACT Image-based control (IBC) systems have a long sensing delay due to compute-intensive image processing. Modern multiprocessor IBC implementations consider either parallelisation of the sensing task or pipelining of the control loop to cope with this long delay. However, the impact of both parallelisation and pipelining together on the quality-of-control (QoC) of IBC systems is not explored in the literature. We present a model-based design method for multiprocessor IBC implementation, considering both parallelisation and pipelining together. In particular, we address the following problem: For a given platform allocation, what is the optimal degree of pipelining and degree of parallelisation required to maximise the QoC? The proposed method takes into account image-workload variations, inter-frame dependencies and platform constraints. The application is efficiently modelled and analysed using a scenario-aware dataflow graph, and an implementation-aware switched controller is designed that optimises QoC and guarantees stability. We validate the proposed method using simulations and hardware-in-the-loop experiments, considering a lane-keeping assist system.

INDEX TERMS Image-based control, switched linear control, scenario-based design, platform-aware design, multiprocessor implementation, hardware-in-the-loop validation.

I. INTRODUCTION

Image-Based Control (IBC) systems are feedback control systems whose feedback is provided by camera(s) as the sensor(s) (illustrated in Fig. 1 (a)). A camera captures image frames at a pre-defined constant frame rate per second (fps) from the dynamic system environment. A compute-intensive image processing algorithm processes the image frames to detect features in the image such as objects, traffic signs and lanes. These features are then used to compute the states of the system, such as relative position and distance [1]. A controller computes the control input for actuation using the computed states. The actuation task applies the computed control input to the IBC system.

A typical periodic implementation of such an IBC system is illustrated in Fig. 1 (b). The main challenge here is to deal with the inherent long (worst-case) sensing delay due to compute-intensive image-processing algorithms. A long processing delay results in dropping some camera frames from processing. Moreover, the sensing delay is variable due to image workload variations [2]. These variations can be captured statistically using a probability distribution [3] (illustrated in Fig. 1 (c)). A long worst-case sensing delay leads to a long sensor-to-actuator delay τ (the time between the start of a sensing task and the end of the corresponding actuation task) and thus results in degraded control performance [4], [5]. The question is: *How to cope with the long variable sensing delay in an IBC system?*

The advent of multiprocessor platforms enables **coping with the long sensing delay** by either *parallelising the sensing task* [6], [7] or *pipelining the control loop* [8], [9]. Parallelisation refers to executing sensing subtasks in parallel and thereby reduces the delay compared to the worst-case (illustrated in Fig. 2 (a)). It is, however, limited by the degree

The associate editor coordinating the review of this manuscript and approving it for publication was Wen-Sheng Zhao.

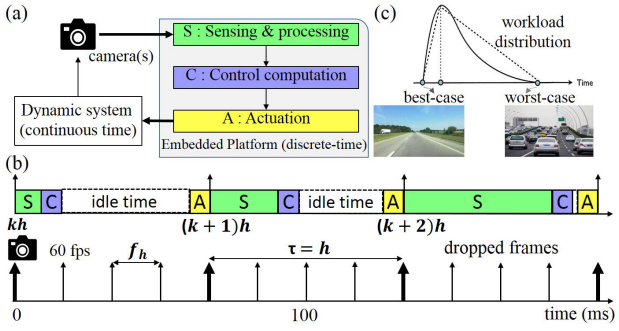

FIGURE 1. **An image-based control (IBC) system: (a) block diagram; (b) Gantt chart for a typical IBC implementation; (c) workload variations captured as a distribution.**

of parallelism of the sensing algorithm. Pipelining refers to the pipelined execution of the control loop over multiple processing cores. Pipelining helps to reduce the number of camera frames being skipped. It reduces the sampling period h (the time between the start of two successive sensing tasks) by processing frames on available cores (illustrated in Fig. 2 (b)). Pipelining is, however, limited by the presence of inter-frame dependencies, i.e. the data or algorithmic dependencies between consecutive frame processing, e.g. due to video coding [10] or visual tracking [11]. In literature, the controller implemented for the pipelining is for $\tau > h$ [8], [9] and for parallelisation is for $\tau \leq h$ [6], [7].

Why should we consider pipelining and parallelism together? Pipelining does not reduce the delay τ compared to the worst-case. By executing the frames in a pipeline, only h is reduced, whereas parallelising the sensing tasks reduces τ. However, h is still at least τ for a parallel implementation. Considering both pipelining and parallelism together helps to reduce both τ and h and thereby improves the quality-of-control (QoC) of our IBC system. Further, the inherent limitations of pipelining - due to inter-frame dependencies - and parallelism - due to a limited degree of parallelism of the algorithm - can be mitigated significantly by considering them both together. The challenge then is to identify the optimal implementation choice considering both the degrees of pipelining and parallelism that improves the system performance. The degree of pipelining is quantified by the maximum number of active pipes in the pipeline, and the degree of application parallelism is quantified by the maximum amount of parallel execution within one sensing task. Both are limited by the available processing resources.

Challenges: The current literature does not explore the impact of both pipelining and parallelism together on the QoC of IBC implementation. Existing pipelined IBC implementations [8], [9] assume that the mapping is given, and that each pipe is mapped to a unique resource *without any resource sharing between pipes*. This is a restrictive implementation choice. Inter-frame dependencies, which are crucial for practical implementation, are also not considered. There are two main challenges that are not explicitly explored in the literature. First, how to model a multiprocessor IBC system considering both pipelining and parallelism together? The challenge in modelling is to explicitly consider workload variations, inter-frame dependencies and constant (often periodic) control timing parameters τ and h. Second, how to identify the optimal mapping of the sensing task on shared processing resources that considers both pipelining and parallelism together and provides a tight analytical bound on control timing parameters τ and h so as to optimise QoC.

In literature, pipelining and parallelism are captured by a model-of-computation (MoC) such as synchronous dataflow graphs (SDFG) [12]. Our approach 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 dataflow graphs (SADFG) [13] as our MoC as it inherently supports modelling scenarios and has tool support for timing analysis and platform-aware mapping. Existing mapping analysis tools [14], [15] typically assume that each node (subtask or actor) in the graph is bound to one processing resource. Pipelining involves (possibly) concurrent executions of subtasks on multiple resources, with inter-frame dependencies between actor instances. Moreover, control assumes careful time-triggered execution of sensing and actuation tasks. All these aspects can only be analysed after non-trivial graph transformations (as e.g. exemplified in [16]).

Fig. 2 (c) illustrates an implementation of two pipes on a shared platform allocation of two processors, with each pipe having a parallelised sensing subtask. With parallelised pipes but without resource sharing between pipes, we would need four processors to achieve the same delay and period as obtained in Fig. 2 (c). To integrally consider pipelining and parallelisation on a shared multiprocessor platform, we need an efficient analysis to identify the optimal mapping. The mapping should guarantee the required (often constant) worst-case delay and period for the controller design.

The contributions of the current paper are as follows:

1) We extend the scenario- and platform-aware design (SPADe) approach of [7] by considering pipelining of the control loop and formalising the IBC system modelling. The extended SPADe approach (as explained in Sec. IV) integrally considers pipelining and parallelism for a multiprocessor IBC implementation. The state-of-the-art approaches focus on either pipelining or parallelism. The exact problem addressed is the following: *For a given multiprocessor platform allocation, identify the optimal design choice for an IBC system considering both pipelining and parallelism and explicitly considering image-workload variations, inter-frame dependencies, resource sharing between pipes and platform constraints.* The optimal design choice identifies the degree of pipelining and degree of parallelism required for maximising the QoC and is translated into *system configurations* that guarantee control timing parameters.
2) We propose model transformations for modelling, analysing, and mapping the IBC system. The model

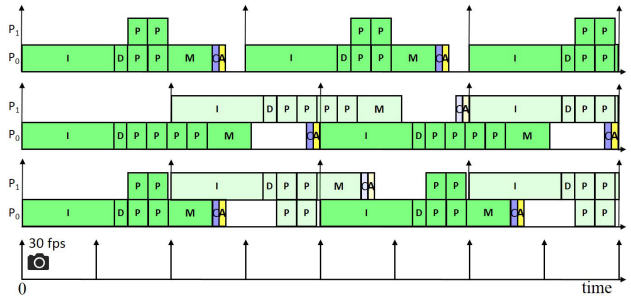

FIGURE 2. IBC implementations for worst-case image workload: (a) Parallelisation of sensing; (b) Pipelining without resource sharing; (c) Pipelining and parallelism together with resource sharing. Note: 1) Sensing task S is composed of image signal (pre-)processing (I), regions-of-interest (RoI) detection D, RoI processing P, and RoI merging M explained in Sec. IV-A1; 2) P_0 and P_1 are the two processing cores.

transformations for pipelined parallelism are the main contribution. These transformations consider both pipelining and parallelism together (as explained in Sec. V). The model transformations allow us to relate the dataflow timing (throughput and latency) analysis to the key control timing parameters (h and τ) and to optimise the mapping while integrally considering pipelining and parallelism along with workload variations, inter-frame dependencies and resource-sharing between pipes. Implementation-aware model transformations for model-based design of IBC systems are not considered in the existing literature.

3) We validate our approach using Matlab simulations considering a predictable multiprocessor platform - CompSOC [17] - and using hardware-in-the-loop (HiL) experiments with an industrial heterogeneous multiprocessor platform - NVIDIA AGX Xavier - considering a lane-keeping assist system (LKAS).

The rest of the paper is organised as follows. Section II explores the related work. Section III describes the multiprocessor IBC system implementation, the motivating case study and the QoC metrics. Section IV details the SPADe design flow. Section V introduces the model transformations required for the SPADe design flow to analyse pipelined parallelism. Section VI revisits the SPADe design flow and precisely describes an algorithm using the model transformations and other considerations for pipelined parallelism. Section VII explores the experimental results, the design-space exploration (DSE), and compares SPADe with the state-of-the-art multiprocessor IBC system implementations. Section VIII presents the SPADe adaptation for an industrial platform, the NVIDIA AGX Xavier, and validates the results of our approach in a hardware-in-the-loop (HiL) setting. Section IX concludes the work and suggests possible future directions.

II. RELATED WORK

This work deals with an effective model-based design flow for multiprocessor IBC implementation, considering both pipelining and parallelism together. Relevant literature deals with the questions: What are the relevant design approaches for such embedded control problems? What are the techniques for IBC system design to deal with long delays in a feedback control loop? What are the relevant IBC system modelling and analysis techniques?

A. DESIGN APPROACHES

An IBC system is often designed based on the *separation-of-concerns* principle between the control theory and embedded systems disciplines [18], [19]. Co-design of control and scheduling is another design paradigm explored in the literature [20]. The emphasis is on platform-based design methods that take into account platform resource constraints while designing the controller [18], [21]. Contract-based design [22] is a platform-based design paradigm for cyber-physical systems where the interactions between control theory and embedded design are defined based on contracts.

From the embedded-systems discipline, a system-scenario-based design approach [23] 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 runtime scenario transitions are specific for each application and generally not trivial.

The SPADe scenario- and platform-aware design approach for non-pipelined IBC systems is proposed in [2], [7] where the concepts of the system-scenario-based design and platform-based design methods for IBC are combined into a co-design approach that jointly develops and optimises the image-processing implementation and the controller design. In this work, we extend the SPADe approach [7] by considering pipelining of the control loop along with parallelism and formalising the IBC system modelling.

B. IBC SYSTEM DESIGN

The main challenge in designing an IBC system is to cope with the long sensing delay. Control engineers tackle a long delay using advanced state estimation [24], robust design [25], predictive control [26], observer-based [27], and multi-rate sampling [28] methods. These methods rely heavily on the system model and are vulnerable to modelling errors with longer delays. Embedded systems engineers aim to reduce processing delay and period by parallel implementations of the algorithms using heterogeneous multiprocessor platforms having specialised hardware such as GPUs [7], [29] and FPGAs [30]. Pipelined control [8], [9] is another approach targeting homogeneous multiprocessor implementations that reduce the effective sampling period without changing the processing delay. However, both pipelining and parallelism together for IBC systems implementation is not explored in the current literature.

C. IBC SYSTEM MODELLING

Model-based design [31], [32] approaches focus on designing applications based on abstract models of application and platform such that the implementation is guaranteed to behave with predictable performance. Numerous models-of-computation (MoC) are available in literature [13], [33]–[35]. Our approach 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 dataflow graphs (SADFG) [13] as our MoC as it inherently supports modelling scenarios and has tool support for timing analysis and platform-aware mapping. The graph transformations we propose in this work are, however, specific for SADFG.

Existing literature does not model or explore the impact of both pipelining and parallelism together on the QoC of an IBC system. Inter-frame dependencies, e.g. due to video coding [10] or visual tracking [11], and resource sharing between pipes, which are crucial aspects for a practical IBC system implementation, are also not explicitly considered in the literature.

III. MULTIPROCESSOR IBC IMPLEMENTATION

We consider a typical setting for an IBC system as shown in Fig. 1 (a) having the workload distribution as illustrated in Fig. 1 (c). The main sensor is a 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. 60 fps and image arrival period f_h given by $f_h = 1/60 = 16.67$ ms. The tasks include compute-intensive image sensing and processing (S), control computation (C), and actuation (A), which are then mapped to run on a multiprocessor platform.

A. PLATFORMS UNDER CONSIDERATION

We consider a predictable and composable multiprocessor system-on-chip (MPSoC) platform - CompSOC [17] for illustrating the SPADe approach. CompSOC offers a tile-based architecture [36] (see Fig. 3). Each tile has a processor P_i, memory M, communication assist CA and network interface NI. Each processor tile has a microblaze processor, the memory tile contains an external memory interface, e.g., DDRAM, and the NoC provides interconnection between the tiles. The platform is predictable with tight bounds on WCETs of tasks, and composable so that applications sharing the platform do not interfere with each other. A scheduler performs (re)configuration and time-triggered task execution.

Predictability and composability are usually not offered in an industrial platform. We adapt our approach for the industrial platform NVIDIA AGX Xavier [37] (illustrated in Fig. 4) to demonstrate its applicability in an industrial context [7]. It consists of a Xavier system-on-chip (SoC) and other components explained in [37]. The CPU complex consists of four heterogeneous dual-core NVIDIA Carmel

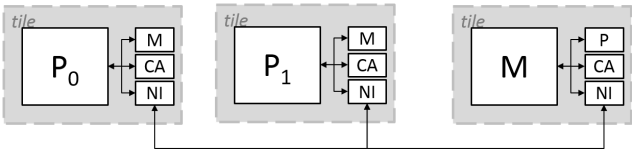

FIGURE 3. A CompSOC MPSoC platform with two processor tiles and a memory tile connected through a NoC.

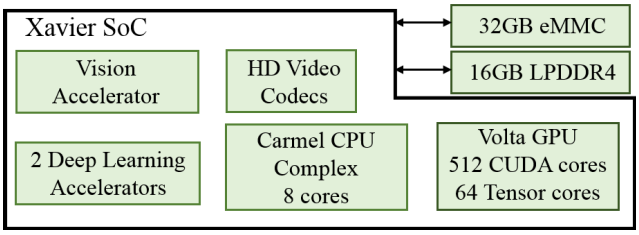

FIGURE 4. NVIDIA AGX Xavier platform block diagram. LPDDR4 and eMMC are the memory blocks. Each CPU cluster also has internal instruction and data memory (not shown in the graph).

CPU clusters based on ARMv8.2 with a maximum clock frequency of 2.26GHz. The GPU with a maximum clock frequency of 1.37GHz is accessed via the CPUs in the SoC. The Ubuntu 18.04 LTS OS runs on the CPU platform.

B. CONTROL SYSTEM AND EMBEDDED IMPLEMENTATION

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

$$\dot{x}(t) = A_c x(t) + B_c u(t),\ y(t) = C_c x(t) + D_c u(t), \quad (1)$$

where $x(t) \in \mathbb{R}^n$ represents the *state* vector, $y(t) \in \mathbb{R}$ contains the measured *output* 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, C_c and D_c represent the system, input, output and feedforward matrices of appropriate dimensions.

A typical 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 of these tasks, S^k, C^k, and A^k, be given by $t_s(.)$ and $t_f(.)$, respectively. The execution times of S^k, C^k, and A^k are then given by $e_T^k = t_f(T^k) - t_s(T^k)$, where $T \in \{S, C, A\}$. The interval between the starts of two consecutive executions of sensing tasks S^k and S^{k+1} is the *sampling period* h^k for the k-th instance. In addition, the time interval between the start time of S^k and finish time of A^k is the *sensor-to-actuator delay* τ^k for the k-th instance.

$$h^k = t_s(S^{k+1}) - t_s(S^k), \quad \tau^k = t_f(A^k) - t_s(S^k). \quad (2)$$

A sensing operation typically takes a much longer time than the other two operations, i.e., $e_S \gg e_C + e_A$, where e_S, e_C and e_A are the worst-case execution times of tasks S, C, and A, respectively. Due to platform constraints or image-workload variations, h^k and τ^k might vary. However, for a typical controller implementation, we need to guarantee a

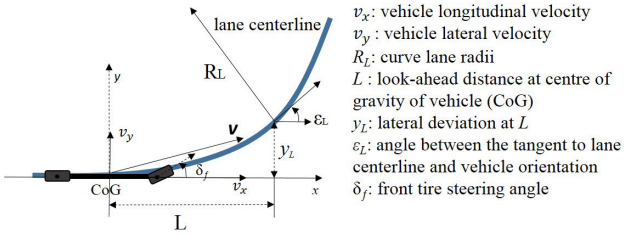

FIGURE 5. LKAS dynamics model derived from [38].

constant (worst-case) delay τ and sampling period h. As such, we consider a time-triggered implementation of tasks S, C and A. We assume that the start of sensor-data processing is aligned with the camera frame arrival, i.e., h is an integer multiple of f_h. Also, the control computation task and actuation task are delayed, if needed, to guarantee a constant τ such that $t_s(C^k) = t_s(S^k) + \tau - e_C - e_A$, and $t_s(A^k) = t_s(C^k) + e_C$.

For a non-pipelined implementation (see Fig. 2 (a)), $\tau \leq h$, i.e., $t_s(S^{k+1}) \geq t_s(S^k) + \tau$. For a pipelined implementation (see Fig. 2 (b), (c)), $\tau > h$, i.e. $t_s(S^{k+1}) < t_s(S^k) + \tau$. 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 $k \in \mathbb{Z}_{\geq 0}$.

The main challenge here is to compute tight τ and h for a multiprocessor IBC implementation. Identifying the optimal mapping that guarantees a constant τ and h, considering both pipelining and parallelism together, is non-trivial.

C. MOTIVATING CASE STUDY: LKAS

We illustrate our work using the motivating case study of a lane keeping assist system (LKAS). We consider the bicycle model derived from [38] (illustrated in Fig. 5) for simulating the LKAS of a vehicle[1] on a straight road and it is described as follows,

$$A_c = \begin{bmatrix} -\frac{c_f + c_r}{m v_x} & \frac{-m v_x^2 + c_r l_r - c_f l_f}{m v_x} & 0 & 0 \\ \frac{-l_f c_f + l_r c_r}{I_\psi v_x} & -\frac{l_f^2 c_f + l_r^2 c_r}{I_\psi v_x} & 0 & 0 \\ -1 & -L & 0 & v_x \\ 0 & -1 & 0 & 0 \end{bmatrix},$$

$$B_c = \begin{bmatrix} \frac{c_f}{m} & \frac{l_f c_f}{I_\psi} & 0 & 0 \end{bmatrix}^\top,$$

$$C_c = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix},$$

$$D_c = 0,$$

where, referring to Fig. 5, we define the state vector $x(t) = [v_y, \dot{\psi}, y_L, \varepsilon_L]$, the measured output $y(t)$ as y_L, and the control input $u(t)$ as the steering angle δ_f, where $\dot{\psi}$ is the vehicle's yaw rate in rad/s, where the velocity components v_x and v_y are in m/s, where l_f, l_r (= 1.22 and 1.62 m respectively) denote distance of the front and rear axles from the center of gravity (CoG), where I_ψ (= 2920 kg·m^2) is the total inertia of the vehicle around its CoG, where c_f, c_r (= 1.2×10^5 N/rad) denote cornering stiffness of the front and rear tires, and where the total mass of the vehicle is m (= 1590 kg).

[1]The vehicle parameters are those specified in [38] for Honda Accord.

D. QUALITY-OF-CONTROL (QoC) METRICS FOR CONTROL STABILITY AND PERFORMANCE

We evaluate the QoC of our IBC system design choices by considering stability and performance. Stability margins - gain and phase margins [39] - quantify the control stability and give an analytical basis to compare two different IBC system design choices and thus allow us to identify the optimal degree of pipelining and application parallelism. Once we make a choice, we can further optimise the controller with respect to performance, considering mean square error (MSE) and settling time (ST).

The gain margin and phase margin quantify the additional gain and phase lag that makes the system marginally stable. Systems with greater stability margins can withstand greater changes in system parameters before becoming unstable. Gain margin and phase margin are computed analytically from the system model. On the other hand, MSE and ST can be analysed only through simulations.

1) GAIN MARGIN (GM)

The GM is defined as the change in open-loop gain expressed in decibels (dB), required at 180 degrees of phase shift to make the system unstable. The GM is the difference between the magnitude curve and 0dB at the point corresponding to the frequency that gives us a phase of -180 degrees (the phase cross-over frequency).

2) PHASE MARGIN (PM)

The PM is the change in open-loop phase shift required at unity gain to make a closed-loop system unstable. The PM is the difference in phase between the phase curve and -180 degrees at the point corresponding to the frequency that gives us a gain of 0dB (the gain cross-over frequency).

The control performance quantifies, in essence, how fast the output $y(t)$ reaches the reference r_{ref}. The control performance can be tuned in the cost function for the control gains' design using the state and input weights [2].

3) MEAN SQUARE ERROR (MSE)

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

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

where n is the number of observations, $y[k]$ is the value of the k^{th} observation and r_{ref} is the reference value. A lower MSE implies a better QoC.

4) SETTLING TIME (ST)

The settling time is defined as the time required for the output $y(t)$ to reach and stay within a range of a certain percentage (usually 5% or 2%) of the final (reference) value r_{ref} forever without external disturbances.

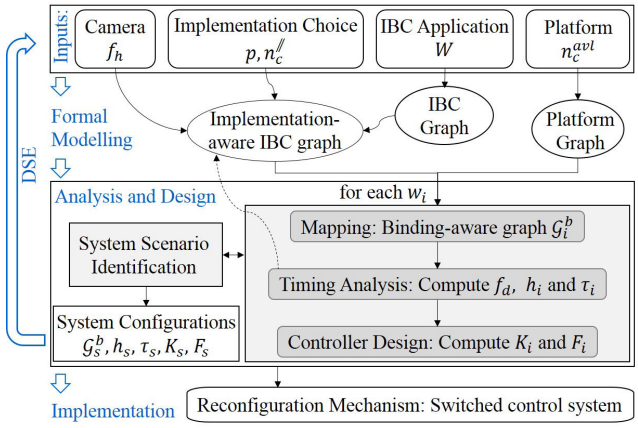

FIGURE 6. Overview of our SPADe design flow. W is the set of varying workloads and w_i, $\mathcal{G}_i^b$, τ_i, h_i, K_i and F_i are the workload, binding-aware graph, sensor-to-actuator delay, sampling period, feedback gain and feedforward gain for a workload scenario s_i (determined by $w_i \in W$); $\mathcal{G}_s^b$, τ_s, h_s, K_s and F_s are the corresponding parameters for an identified system scenario s_s (that abstract multiple workload scenarios). f_h is the camera frame arrival period, f_d is the inter-frame dependence time, p is the number of pipes for pipelining, $n_c^{//}$ is the number of cores allocated for parallelism per pipe, and n_c^{avl} is the total number of available cores.

IV. SPADe DESIGN FLOW

We present a scenario- and platform-aware design flow (SPADe) for IBC systems extending the approach presented in [7] by considering pipelining along with parallelism and formalising the IBC system modelling. An overview of our SPADe approach is illustrated in Fig. 6, summarised below and explained in detail in subsequent subsections.

1) Formal modelling of the IBC system: An IBC application is captured as an IBC SADFG considering workload variations W and the platform as a platform graph. Further, an *implementation-aware* IBC SADFG captures the given design parameters - camera frame arrival period f_h, maximum number of allowed pipes p, maximum allocated available cores n_c^{avl} and allocated processing cores for parallel execution per pipe $n_c^{//}$. The design parameters fully determine the implementation choice - non-pipelined without parallelism, non-pipelined with parallelism, pipelined without parallelism and pipelined with parallelism. The parallelism here refers to the parallel execution of sensing subtasks limited by the degree of parallelism of the IBC application. Graph transformations are proposed to obtain the implementation-aware SADFG.
2) Analysis and design: We map the implementation-aware IBC graph for each workload $w_i \in W$ to the platform graph to obtain the *binding-aware graph* $\mathcal{G}_i^b$ for that specific workload using the SDF3 mapping flow [36]. $\mathcal{G}_i^b$ is an SDFG that models the mapping of the implementation-aware graph to the platform graph. The mapping binds each actor in the SDFG to a processing core in the platform graph. For the ordering of execution of actors bound to the same core, a static-order schedule is encoded in the SDFG. A throughput and latency analysis of $\mathcal{G}_i^b$ yields the sensor-to-actuator delay τ_i, and sampling period h_i. For a pipelined implementation, the throughput analysis of the worst-case image-workload scenario allows to compute the inter-frame dependence time f_d (as explained later in Section VI-D1). If $f_d > h_i$, the implementation-aware graph is updated with the realisable period and τ_i and h_i are recomputed. The controllers are then designed for the resulting (τ_i, h_i) to obtain the controller feedback and feedforward gains (K_i, F_i). Trying to cater to the designed workload scenarios at runtime means that we have a switching system. A switching system with too many switching states is challenging for controller stability and may result in poor performance. Hence, we aggregate multiple workload scenarios with similar control timing parameters as a *system scenario*. A system scenario s_s abstracts multiple workload scenarios and has a constant (τ_s, h_s) during implementation. A *system configuration* is defined as the combination of mapping and controller configurations, i.e. $\mathcal{G}_s^b$, τ_s, h_s, K_s, and F_s (as explained later in Section IV-B4). Typically, there are a few identified system scenarios, and the idea is that switching between the system scenarios at runtime guarantees stability and improved performance. For pipelined parallelism, a design-space exploration (DSE) using the SPADe flow needs to be performed by varying the design parameters to identify the best implementation choice (parameters p, $n_c^{//}$, further explained in Section VII-A).
3) Runtime implementation: The system configurations for the implementation choice are stored in a look-up table (LUT) in platform memory for the runtime implementation. Dynamic runtime reconfiguration may be needed since there can be a switching behaviour between system configurations due to image-workload variations.

A. FORMAL MODELLING

An IBC application is captured as an IBC graph considering workload variations and the platform as a platform graph. Further, an *implementation-aware* IBC graph is created considering the design parameters (the number of pipes p, allocated processing cores for parallel execution per pipe $n_c^{//}$ and camera frame arrival period f_h).

1) IBC GRAPH AND IMPLEMENTATION-AWARE GRAPH

The IBC graph and implementation-aware graph are modelled using an SADFG. Graph transformations to obtain an implementation-aware graph from the IBC graph are different for the different implementation choices and, as such, are explained in later sections. We choose SADFG [13] as the formal MoC for our application as it enables us to: i) model dynamic behaviour and dependencies, analyse timing, and optimally map application (sub)tasks to the platform for maximising the effective utilisation of allocated resources;

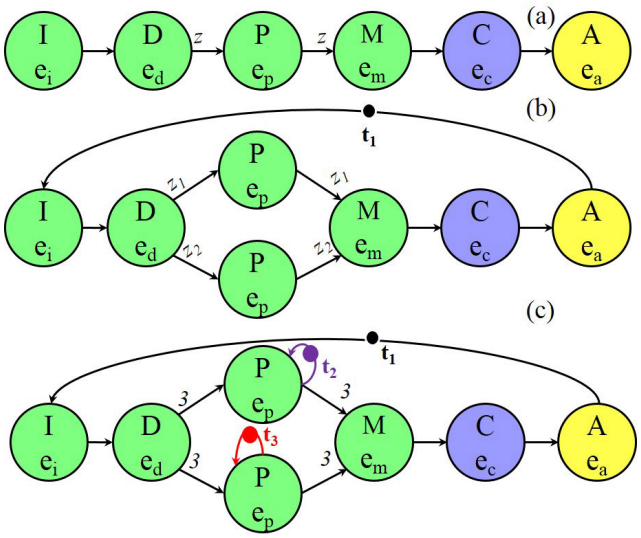

FIGURE 7. IBC SDFG: (a) graph structure. The rates z indicate the workload w. (b) Implementation-aware graph for non-pipelined implementation on two cores (given platform allocation). (c) A (simplified) binding-aware graph for non-pipelined implementation on two cores for a workload of 6 RoI.

ii) relate latency and throughput of the data flow graph to the control timing parameters τ and h, and thus combine data flow analysis and mapping with control design parameters and QoC; iii) analyse inter-frame dependencies (captured as inter-frame dependence time f_d) through graph transformations (as explained in Sec. V); and iv) to efficiently implement a runtime mechanism that manages necessary dynamic reconfiguration.

Following the formalisation of [40], an SADFG (see Fig. 7 (a)) is a tuple $(\Sigma,\ \mathcal{F})$, 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 workloads w_i and their corresponding synchronous data flow graphs (SDFGs) $\mathcal{G}_i$;
- the (ω-)language $\mathcal{F}$ describes a set of infinite scenario sequences represented using ω-regular expressions of scenarios $s_i \in \Sigma$.

We assume that workloads are totally ordered, i.e., for any two workloads w_i and w_j, either $w_i \leq w_j$ or $w_j \leq w_i$. An SDFG [12] is a tuple $\mathcal{G} = (\mathcal{A},\ \mathcal{C},\ e,\ r_p,\ r_c,\ i)$ where $\mathcal{A}$ is a finite set of actors, $\mathcal{C} \subseteq \mathcal{A}^2$ the set of channels, $e : \mathcal{A} \to \mathbb{R}_{\geq 0}$ returns for each actor its associated firing delay or execution time, $r_p : \mathcal{C} \to \mathbb{N}_{>0}$ is a partial function that returns for each channel its production rate, $r_c : \mathcal{C} \to \mathbb{N}_{>0}$ is a partial function that returns for each channel its consumption rate, $i : \mathcal{C} \to \mathbb{N}_0$ returns for each channel its number of initial tokens. Actors of an SDFG may fire, consuming and producing tokens according to the specified consumption and production rates.

A *repetition vector* ρ of an SDFG $\mathcal{G}$ is a function $\rho : \mathcal{A} \longrightarrow \mathbb{N}_0$ such that for every channel $c = (a_m, a_n) \in \mathcal{C}$, $r_p(c) \times \rho(a_m) = r_c(c) \times \rho(a_n)$. A repetition vector ρ for an SDFG $\mathcal{G}$ is called non-trivial iff for all $a_m \in \mathcal{A}$, $\rho(a_m) > 0$. An SDFG is called consistent iff it has a non-trivial repetition vector. For a consistent SDFG, the unique smallest non-trivial repetition vector is designated as the repetition vector ρ of the SDFG. An SDFG *iteration* is a minimal non-empty set of actor firings that has no net effect on the token distribution in the graph. For a consistent SDFG, for all $a_m \in \mathcal{A}$, the set contains $\rho(a_m)$ firings of a_m. For the scope of this work, we assume that the IBC graph, which is an SADFG, can only have consistent SDFGs and the SDFGs are deadlock-free. These assumptions can be checked efficiently and are valid as any SDFG which is inconsistent or deadlocks is not useful in practice.

The SADFG for our example IBC system is visualised in Fig. 7 (a). The sensing and processing task receives the RAW camera image frames, which are processed in a sequence of steps to extract the state information required for the controller. The image-signal (pre-)processing (I) subtask converts the RAW image in the Bayer domain to pixels in the RGB domain. (Sub-)tasks translate to actors in the data flow graph, shown as circles in the figure. Data dependencies between (sub-)tasks translate to channels, shown as arrows. After the image processing, we detect the regions-of-interest (RoI) in the RGB image frames (D). RoI are processed (P), and, subsequently, the controller state (the lateral deviation y_L in our LKAS case study) is computed by the RoI merging (M) subtask. 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.

The total number of RoI detected by D determines the workload w_i, i.e., $w_i = z$ in Fig. 7 (a). Note that the workloads here are totally ordered as it is related to the number of RoI. z is the production rate of the channel from actor D to actor P, and, correspondingly, the consumption rate of the channel from actor P to actor M. Rates are annotated with the channels, where rates of 1 are not shown explicitly. The workloads translate to variable token production and consumption rates in the graphs.

Graph transformations are required to analyse the parallel and/or pipelined implementations. This is, among others, because the typical mapping analysis tools assume that one actor can be bound to only one processing core. Fig. 7 (b) shows an implementation-aware graph for a non-pipelined parallelised implementation on two processors. It has two actor instances of the P subtask. The workload $w_i = z_1 + z_2$ in this case.

Each workload w_i in an SADFG is associated with an SDFG $\mathcal{G}_i$. An SDFG instance of Fig. 7 (b) is obtained by assigning values to parameters e_j (the actor execution times) and z_k. E.g., assigning $z_1 = 3,\ z_2 = 3,\ e_i = 10,\ e_d = 5,\ e_p = 10,\ e_m = 3 \times (z_1 + z_2) = 18,\ e_c = 1,\ e_a = 1$ gives the SDFG for a workload of 6 RoI for mapping to two processors. There is one (labelled) initial token t_1 in the channel from actor A to I. This channel, with its single initial token, enforces a non-pipelined execution of the control loop. All actors in Fig. 7 (a) have repetition-vector entries of 1, except actor P, which as a repetition-vector entry z; and all

actors in Fig. 7 (b) have repetition-vector entries of 1, except the two P instances that have entries z_1 and z_2 respectively.

Fig. 7 does not show the language of allowed scenario sequences. In the LKAS case, all possible workload sequences are allowed.

2) PLATFORM GRAPH

A platform, e.g. the CompSOC MPSoC shown in Fig. 3, is modelled as a platform graph that captures processing resources, and other relevant aspects such as memories and connections, with their processing and access latencies, data rates, etc. The details needed for the model depend on the used mapping flow. For the sake of explaining SPADe, we assume the platform is simply abstracted as a set of tiles. A tile $\mathcal{T}_i$ abstracts a resource with the processor type pt_i that determines the execution time of actors bound to the tile. The CompSOC instance shown in Fig. 3 has three tiles. Two of these tiles have a microblaze processor type. The third tile is a memory tile that does not play a role in further explanations. Also, the connections are abstracted for the sake of simplicity. Hence, the platform is abstracted as a 2-node platform graph without any connections. Note that the used SDF3 mapping flow does support the modelling of memories and connections, including their timing, and takes these into account in the mapping optimisation.

A platform allocation determines the resources that are allocated to a task or to an application. Resources that are allocated may include the number of tiles or processors, or parts of processors (e.g. slots in a time-division multiplexing (TDM) frame in CompSOC), and types of processors, e.g. GPU, ARM, and microblaze. For our running LKAS example, an allocation consists only of the number of tiles of a specific processor type.

B. ANALYSIS AND DESIGN

We map the implementation-aware IBC graph for each workload $w_i \in W$ to the platform graph to obtain the *binding-aware graph* $\mathcal{G}_i^b$ (further explained below) using the SDF3 mapping flow [36]. A throughput and latency analysis of $\mathcal{G}_i^b$ yields the control timing parameters τ_i and h_i for the workload scenario s_i. Controllers are designed for each workload scenario s_i using the computed timing parameters $(\tau_i,\ h_i)$ to obtain the controller feedback and feedforward gains $(K_i,\ F_i)$. System-scenario identification is then performed to identify the set of system scenarios for runtime implementation. For a pipelined implementation, inter-frame dependence time f_d (as explained in Section VI-D1) is also computed using the throughput analysis.

1) SYSTEM MAPPING AND MAPPING CONFIGURATIONS

System mapping refers to the mapping of the IBC application (modelled as an SADFG) to the given platform (modelled as a platform graph). Note that for each workload scenario s_i, we can have multiple mapping options for the given platform allocation. The throughput and latency of each of these mapping options would be different. The concrete problem is to find the mapping of s_i to the given platform allocation that maximises throughput. Any design flow that does the (Pareto-)optimal mapping of an application to a platform while maximising throughput can be used.

We use the SDF3 mapping flow [14] as it optimises the resource usage, memory load and communication load for mappings (to the extent that these aspects are considered in the models), and embeds state-of-the-art throughput analysis techniques. Mapping an s_i (modelled as an SDFG $\mathcal{G}_i$) to a platform graph generates a binding-aware SDFG $\mathcal{G}_i^b$. $\mathcal{G}_i^b$ is an SDFG that models the mapping of the implementation-aware graph to the platform graph, where each actor in the SDFG is bound to a tile in the platform graph. For the ordering of execution of actors bound to the same tile, a static-order schedule is encoded in $\mathcal{G}_i^b$.

Fig. 7 (c) shows a simplified binding-aware graph for the 6-RoI workload scenario of the running example, bound to two tiles. It encodes two static-order schedules: IDP^{3}MCA for one iteration of the graph on one core and P^3 for one graph iteration on the second core. Self-loops with a single token need to be added to the two parallelised P actors to model the binding of the actor to a particular core and to enforce sequential execution of the P firings on each of the two cores. This suffices to encode the schedules. The graph is simplified in the sense that SDF3 encodes many more aspects in the binding-aware graph, such as memory accesses and interprocessor communication.

A *mapping configuration* $\chi_{s_i}^m$ refers to the binding of s_i to the platform and its execution schedule represented in a binding-aware SDFG. The SPADe flow tries to minimise the number of cores used even if a given number of cores is allocated. This happens naturally when we map our SDFGs to the platform using the SDF3 tool, as SDF3 gives a Pareto-optimal mapping that minimises utilisation.

2) TIMING ANALYSIS - COMPUTING f_d, τ_i AND h_i

The computation of inter-frame dependence time f_d is specific for pipelined implementation and is explained later in Section VI-D1. In this subsection, we explain how we compute the throughput and latency of the SADFG and relate it to the control timing parameters τ_i and h_i for a workload scenario s_i. Note that the state-of-the-art SADFG analysis uses (max, +) algebra [41] and the definitions needed for the computation of throughput have already been explained in [7]. In this subsection, we summarise the relevant definitions for our analysis. For detailed explanations, the reader is referred to [40].

A time-stamp vector $\boldsymbol{\gamma_0}$ captures the availability times the initial tokens. The production times of the final tokens resulting from the execution of a scenario s are then $\boldsymbol{\gamma_1} = \boldsymbol{G_s}\boldsymbol{\gamma_0}$, where $\boldsymbol{G_s}$ is the scenario (or state) matrix of s. For the binding-aware scenario SDFG corresponding to 6 RoI, introduced in Fig. 7 (c), $\boldsymbol{\gamma_0} = [0\,0\,0]^T$. That is, the three initial tokens are all available at time 0. Scenario matrix $\boldsymbol{G_s}$ captures the dependencies and corresponding delays between the initial and final tokens. For the running

example, $\boldsymbol{G}_s$ equals

$$\begin{bmatrix} e_i+e_d+3e_p+e_m+e_c+e_a & e_m+e_c+e_a & e_m+e_c+e_a \\ e_i+e_d+3e_p & 3e_p & -\infty \\ e_i+e_d+3e_p & -\infty & 3e_p \end{bmatrix}$$

Entry ij in this matrix contains the time delay from consuming token t_j to reproducing token t_i in one iteration of the graph. The top left entry thus indicates the delay to reproduce the final token t_1 on the A-I channel. The two $3e_p$ entries show that the three firings of the two P actors are sequentialized. The two $-\infty$ entries indicate that the two self-loop tokens of the two P actors, t_2 and t_3, are independent. The other entries capture the delay from t_1 to the self-loop tokens t_2 and t_3 and the delay from t_1 to t_2 and t_3.

With the concrete actor execution times given earlier, this results in the following concrete matrix:

$$\begin{bmatrix} 65 & 50 & 50 \\ 45 & 30 & -\infty \\ 45 & -\infty & 30 \end{bmatrix}$$

The production times after execution of scenario s are then obtained from $\boldsymbol{\gamma_1} = \boldsymbol{G}_s\,[0\,0\,0]^T = [\,\max(65, 50, 50)\ \max(45, 30, -\infty)\ \max(45, -\infty, 30)]^T = [65\,45\,45]^T$. Note that the matrix multiplication in this analysis is the (max, +) matrix multiplication. The analysis shows that the three tokens in the binding-aware graph of Fig. 7 (c) are reproduced after 65, 45, and 45 time units, respectively.

$\boldsymbol{G}_s$ is used to determine the evolution of any scenario sequence. The final tokens of one scenario execution 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 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 scenarios $s \in \Sigma$, we can construct $\boldsymbol{G}_s \in \mathbb{R}_{-\infty}^{i(s)\times i(s)}$ following the procedure of [42]. Here, $i(s)$ is the total number of initial tokens (in all channels) for scenario s and $\mathbb{R}_{-\infty} = \mathbb{R} \cup \{-\infty\}$ is the domain of (max, +) algebra.

Further, we need to analyse the production times of *outputs*, i.e., the relevant information produced, during the 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,

$$\boldsymbol{p}_k = \boldsymbol{H}_s\boldsymbol{\gamma_k} = \boldsymbol{H}_s\boldsymbol{G}_s^k\boldsymbol{\gamma_0} \tag{3}$$

where $\boldsymbol{p}_k$ are the times at which the outputs in the $(k+1)^{th}$ iteration are produced and where $\boldsymbol{H}_s \in \mathbb{R}_{-\infty}^{m(s)\times i(s)}$ is the output matrix of the scenario s that captures the relation between the state vector and the production times of the $m(s)$ outputs. Note that the first output production times are given by $\boldsymbol{p}_0$. 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 I. This means that $\boldsymbol{H}_s = [65\,50\,50]$, corresponding to the first row of $\boldsymbol{G}_s$, and the production time of the first output $\boldsymbol{p}_0 = [65\,50\,50]\,[0\,0\,0]^T = [65]$.

The throughput of an SADFG for an infinite scenario sequence $\bar{s}$ is defined as follows.

$$\nu(\bar{s}) = \lim_{n\to\infty} \sup \frac{\sum_{i=1}^{n} m(\bar{s}_i)}{\|\boldsymbol{\gamma_n}\|} \tag{4}$$

where $\bar{s}_i$ refers to the i^{th} symbol in sequence $\bar{s}$ (a scenario), and $\|\boldsymbol{\gamma_n}\|$ is equal to the maximum entry in the vector $\boldsymbol{\gamma_n}$.

Latency is the maximum (worst-case) time taken to complete one iteration. Given an initial state $\boldsymbol{\gamma_0}$, the latency of a scenario sequence $\bar{s}$ relative to a period μ is defined as

$$\mathcal{L}(\bar{s}, \boldsymbol{\gamma_0}, \mu) = \max_{k\geq 0} \boldsymbol{p}_k - \mu k \tag{5}$$

For the infinite execution of the 6-RoI scenario SDFG of Fig. 7 (c), the throughput is $\frac{1}{65}$ and the latency, relative to the period equalling the inverse of the throughput, is 65. We omit the details of the computation, referring the reader to [40]. But the results should not be surprising given the timing analysis of the scenario execution given earlier. Note that the inverse of throughput and latency are equal in this case due to the model with one initial token on the A-I channel that enforces the non-pipelined execution of the SDFG. We exploit such modelling tricks in our model transformations for mapping and pipelined implementation (as explained in later sections).

The sensor-to-actuator delay τ_i and the sampling period h_i for the workload scenario s_i that we need for controller design are computed from the binding-aware graph $\mathcal{G}_i^b$ that is obtained from mapping the implementation-aware graph onto the allocated resources (as explained earlier and elaborated in Section VI). The two values are computed as follows.

$$\tau_i = \mathcal{L}(s_i^\omega, \mathbf{0}, 1/\nu(\mathcal{G}_i^b)),\ h_i = \left\lceil \frac{\tau_i}{f_h \times p} \right\rceil f_h, \tag{6}$$

where $\mathbf{0}$ is the zero vector, f_h is the camera frame arrival period, and p is the number of pipes in the pipelined parallelism implementation. The delay τ_i of scenario s_i is the latency of executing that scenario repetitively after mapping it onto the platform, with respect to the throughput obtained from that mapping and assuming that initial tokens are available at time 0. For the computation of the effective frame processing period h_i, $\left\lceil \frac{\tau_i}{f_h} \right\rceil$ computes the number of frame periods within the time-interval τ_i. By dividing by the number of pipes p, rounding up, and multiplying with the frame period f_h, one obtains the effective sampling period for the particular scenario implementation. For the infinite execution of the 6-RoI scenario SDFG of Fig. 7 (c), assume $f_h = \frac{1}{60}$ s and $p = 1$. Then, $\tau_i = 65$ ms, in line with the earlier latency analysis, and $h_i = 66.7$ ms. Further details on how the SPADe flow uses τ_i and h_i are provided in Section VI.

3) CONTROLLER DESIGN AND CONTROL CONFIGURATIONS

The LKAS case study we consider is a single input single output (SISO) system. We discretize the IBC system model

in Eq. 1 using $(\tau_i,\ h_i)$, computed for the binding-aware SDFG $\mathcal{G}_i^b$ for the workload scenario s_i. Let p be the number of pipes used in the implementation (as reflected in the binding-aware graph), where non-pipelined implementation corresponds to $p = 1$. We assume that $u[-1] = 0$ and define new system states $z[k] = \begin{bmatrix} x[k]\ u[k-(p-1)] \cdots u[k-1] \end{bmatrix}^T$ with $z[0] = \begin{bmatrix} x[0]\ 0 \cdots 0 \end{bmatrix}^T$ to obtain a higher-order augmented system as follows:

$$\begin{aligned} z[k+1] &= A_{aug,s_i} z[k] + B_{aug,s_i} u[k], \\ y[k] &= C_{aug} z[k] + D_c u[k], \end{aligned} \tag{7}$$

where A_{aug,s_i}, B_{aug,s_i}, and C_{aug} are augmented system matrices. The computation of A_{aug,s_i}, B_{aug,s_i}, and C_{aug} varies for the non-pipelined and pipelined implementation choices and as such is explained in the later sections. A check for controllability [43] is done for the augmented system. If the system is not controllable, controllability decomposition is done to obtain a controllable subsystem.

We can then apply standard control-design techniques [43] for the augmented system models in Eq. 7. We use a *state-feedback* controller $u[k]$ of the following form:

$$u[k] = K_i z[k] + F_i r_{ref} \tag{8}$$

where K_i is the state-feedback gain and F_i is the feedforward gain both designed for the workload scenario s_i. r_{ref} is the constant reference value for the controller.

We design the gains using the optimal linear quadratic regulator (LQR) [43]. A detailed explanation of how we design the gains for our setting is given in [2]. Note that any other state-of-the-art control-design technique can also be used for designing these gains. For each workload scenario s_i, we then define a *control configuration* $\chi_{s_i}^c$ as a tuple $\chi_{s_i}^c = (h_i, \tau_i, K_i, F_i)$.

4) SYSTEM-SCENARIO IDENTIFICATION, SYSTEM CONFIGURATIONS AND STABILITY

System-scenario identification is done to limit the number of switching scenarios during runtime implementation. It is possible for multiple workload scenarios to have the same sensor-to-actuator delay and/or sampling period due to implementation constraints like platform allocation and camera frame rate [2].

For the non-pipelined implementation, a system scenario s_s abstracts multiple workload scenarios s_i such that for $h_s = n \times f_h$, for frame arrival period f_h and some $n > 0$, $(h_s - f_h) < h_i \leq h_s$. That is, we aggregate workload scenarios based on h_s. Then, for the aggregated workload scenarios s_i, we choose τ_s to be the maximum among the τ_i. $\mathcal{G}_s^b$, K_s and F_s are then re-designed for the $(\tau_s,\ h_s)$ identified for the system scenarios s_s. We design $\mathcal{G}_s^b$ by assigning $\tau = \tau_s$ and $h = h_s$ to the corresponding implementation-aware graph and verifying the existence of a mapping that satisfies τ_s and h_s. A control configuration $\chi_{s_s}^c = (h_s, \tau_s, K_s, F_s)$ is then derived following the approach outlined earlier for workload scenarios. Only system scenarios are then considered for defining the system configurations $\chi_{s_s}^s$, which is a combination of control configuration $\chi_{s_s}^c$ and mapping configuration $\chi_{s_s}^m$, i.e., $\chi_{s_s}^s = (\mathcal{G}_s^b,\ h_s,\ \tau_s,\ K_s,\ F_s)$. The system-scenario identification for pipelined implementation is explained in Section VI-D3.

At runtime, the system scenarios switch 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 1 (Stability Criterion [39]): Consider A_{aug,s_s} for all system scenarios s_s 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_s} if there exist $P = P^T > 0$, $Q = Q^T > 0$ and P is the simultaneous solution of the discrete-time Lyapunov equations,

$$A_{aug,s_s}^T P A_{aug,s_s} - P = -Q < 0. \tag{9}$$

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

We transform the stability condition of Eq. 9 into Linear Matrix Inequalities (LMIs) to verify the existence of a CQLF. The analysis equation, Eq. 10, is obtained by performing the following operations: i) substitute A_{aug,s_s} in Eq. 9 with $A_{aug,s_s} = A_{aug,s_s} + B_{aug,s_s} K_s$, ii) apply Schur complement, iii) left- and right- multiplication by $\mathrm{diag}(P^{-1}, I)$, and iv) set $Q = P^{-1}$.

$$\begin{bmatrix} -Q & QA_*^T + QK_s^T B_*^T \\ A_* Q + B_* K_s Q & -Q \end{bmatrix} < 0,\ Q > 0 \tag{10}$$

where $A_* = A_{aug,s_s}$, $B_* = B_{aug,s_s}$ for each scenario s_s. If a solution exists, then the switching subsystems are stable. The choice of system scenarios (particularly the τ_s and h_s) needs to be modified if a solution does not exist.

C. RUNTIME IMPLEMENTATION

At design time, the system configurations $\chi_{s_s}^s$ are stored in a look-up table (LUT) in platform memory. During runtime, for every arriving input image frame, we compute the workload w_i (e.g. through the RoI detection task D) and choose the correct system scenario s_s associated with this workload from the LUT. System configuration $\chi_{s_s}^s$ of the corresponding system scenario s_s is loaded from the LUT. Dynamic runtime reconfiguration is typically needed since there can be a switching behaviour between system configurations due to image-workload variations. For the non-pipelined implementation and the pipelined implementation without resource sharing between pipes, dynamic runtime reconfiguration means that, if needed, a scheduler reconfigures the mapping $\mathcal{G}_s^b$, the time-triggering of the actuation task (that determines τ_s) and the controller gain parameters (K_s and F_s) based on the system scenario s_s associated with the image workload from the LUT. The overhead cost for this reconfiguration needs to be considered in the analysis model, e.g., for the LKAS example, as an additional execution time cost in the actor D (see Fig. 7).

Arbitrary switching and reconfiguration in the pipelined implementation are challenging. Let p be the number of pipes, and h_s be the periods per system scenario. If we restrict h_s to be a multiple of f_h, the number of periods possible due to arbitrary switching considering image-workload variations only grows linearly with f_h and p. However, if we do not restrict h_s and allow it to take arbitrary values, the number of periods possible grows exponentially. E.g. assume that we have three periods h_1 ($= f_h$ for workload w_1), h_2 ($= 2f_h$ for workload w_2), and h_3 ($= 3f_h$ for workload w_3) due to image-workload variations. For simplicity, let us assume that $\tau_s = h_s$ and a given three-core platform allocation with three pipes. Consider the case shown in Fig. 8 where the image frames k, $k+1, \ldots, k+i$ have workload w_3, w_2, w_1, w_3, w_1, w_3, and so on. When multiple control computations complete at the same time, e.g., just before frame $k+3$ is captured, the actuation should be coordinated among the cores. Further, the controller for the image frame $k+4$ should ideally be designed using the discretized model considering $h_1 = f_h$ if the period is defined as the time between two consecutive starts of the sensing task. However, it should also take into account that there was no actuation just before this, i.e., the previous actuation was at the time $t - 2f_h$. So, if the period was defined as the time between two consecutive actuations, then the period for the controller for the frame at $k+4$ is $2f_h$. A similar situation exists for frame $k+5$. Such behaviours add to the complexity of the design space to be explored. The main challenge, however, is proving the stability of the ensuing switched system with these behaviours. Also, modelling these behaviours for the control design is far from trivial.

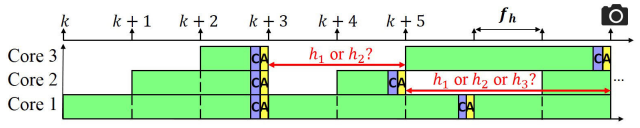

FIGURE 8. Challenges in pipelined implementation due to switching when h_i is a multiple of f_h.

For the scope of this work, we enforce a constant sampling period h_{eff} for the overall pipelined implementation. A constant sampling period helps to limit the design space to be explored and handle the dynamic reconfiguration with less runtime overhead. As explained, the sensor-to-actuator delay τ_s is constant per identified system scenario s_s. Consequently, two system scenarios s_1 and s_2 have the control timing parameters (h_{eff}, τ_1) and (h_{eff}, τ_2).

A similar challenge exists for a pipelined implementation with resource sharing between pipes, where reconfiguring the mapping dynamically is non-trivial. Resource sharing between pipes increases the design space to be explored for considering the possible reconfiguration options. For the scope of this work, dynamic reconfiguration for pipelined implementation with resource sharing between pipes comprises a static mapping where actors are switched on and off considering image workload variations, and choosing the controller gains dynamically from the LUT by the control-computation task based on the system scenario considering the latest state measurement available.

The SPADe flow is not restricted to the mentioned design and implementation choices. Its key feature is that pipelining and parallellism are integrally considered. As we will see, this provides benefits in the achievable QoC. Other controller design and implementation choices can be integrated as long as appropriate timing analysis and stability guarantees can be provided.

V. MODEL TRANSFORMATIONS

This section explains the model transformations required for modelling, analysing, and mapping the IBC system using SADFG. The model transformations are required to obtain the implementation-aware IBC graph from the IBC graph for the given design parameters, as illustrated in Fig. 6. Our model transformations consist of maximising parallelism, creating a pipe, replicating pipes to implement pipelining, introducing camera-awareness, introducing workload-awareness, modelling inter-frame dependencies, and re-timing of actor execution times. For each workload scenario, we assume that the sensing and processing task is modelled as an SDFG $\mathcal{G}_S$, the control computation task is modelled as an SDFG $\mathcal{G}_C$, and the actuation task is modelled as an SDFG $\mathcal{G}_A$. The graphs $\mathcal{G}_S$, $\mathcal{G}_C$, and $\mathcal{G}_A$ should have identifiable source and sink actors $a_{src,i}$ and $a_{snk,i}$, $i \in \{S, C, A\}$. A source is an actor without any incoming edges and a sink is an actor without any outgoing edges. We moreover enforce that $\rho(a_{src,i}) = \rho(a_{snk,i}) = 1$. Having identifiable source and sink actors with repetition-vector entries equal one ensures well-formedness for our model transformations. Note that an SDFG with a single actor satisfies the assumptions. The source and sink actors should be identical across all workload SDFGs in an application IBC SADFG.

To maximize opportunities to speed up the computations in the control loop, we want to maximize parallelism in graphs $\mathcal{G}_S$, $\mathcal{G}_C$, and $\mathcal{G}_A$. Automatically extracting task- and data parallelism in computations is challenging. So, in general, it is up to the designer to maximize parallelism in the three mentioned graphs. But given an SDFG of a workload scenario, it is possible to maximize data parallelism by transforming the SDFG to a homogeneous SDFG (HSDFG) [12], [44]. Essentially, this transformation replicates actors with a repetition-vector entry greater than one into multiple actors (as many as the repetition-vector entry of the actor for the SDFG) with each a repetition-vector entry one in the HSDFG. A platform-aware mapping such as implemented in the SDF3 tool [14] then clusters actors of the HSDFG per processor in the given platform allocation for maximising throughput. A disadvantage of this approach, however, is the scalability of the mapping and performance analysis that depends on the number of actors.

Another option is to replicate the parallelisable actors as many times as meaningful given the platform allocation. That is, we transform an SDFG $\mathcal{G}$ via a transformation $RepA(\mathcal{G}, \varphi)$ that preserves the number and timing of firings in a single

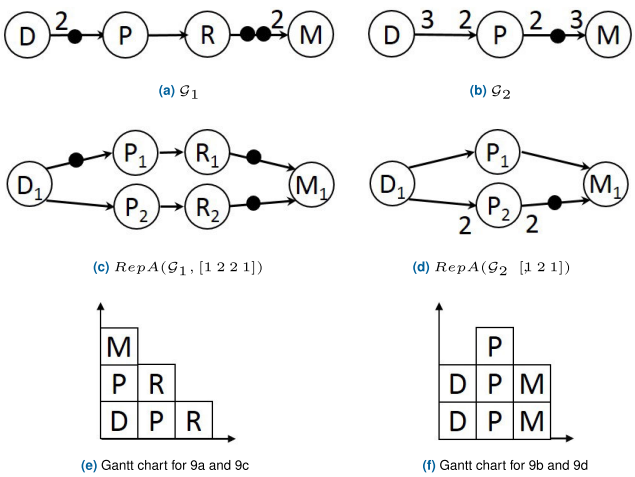

FIGURE 9. Examples of the replicate actors *RepA* transformation. The Gantt charts cover one iteration of the corresponding graph and assume actor execution times to be 1. Subscripts resulting from *RepA* transformations are omitted for brevity.

iteration of the original graph $\mathcal{G}$ in the transformed graph, where φ is the replication vector with size equal to the number of actors in $\mathcal{G}$ and where each element $\varphi(a)$ represents the number of times an actor a needs to be replicated. A straightforward replication vector can then be defined using the repetition vector ρ and the maximum number of processing cores allocated for parallel execution of tasks per pipe $n_c^{//}$, as $\varphi(a) = min(\rho(a), n_c^{//}),\ a \in \mathcal{A}$. Often, this transformation is relatively straightforward, but a definition that works in general is not obvious. The challenge when replicating actors is to accurately model the transformations of channels, production and consumption rates, and initial tokens such that the functional and timing behaviour of the original graph is preserved. Fig. 9 gives some example transformations, including Gantt charts that illustrate that actor firings and their timing are preserved. We leave a generic definition (and the proof that such a transformation exists in general and preserves functionality and timing) as future work. Note that the SDFG-to-HSDFG transformation of [12], [44] is an instance of *RepA* when the replication vector is chosen equal to the repetition vector.

For the remainder, assume that $\mathcal{G}_S$, $\mathcal{G}_C$, and $\mathcal{G}_A$ are the graphs obtained after maximizing parallelism. The **Create pipe** $Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A)$ transformation creates a model for a single pipe by adding a delay actor and channels between the sinks and sources of $\mathcal{G}_S$ and $\mathcal{G}_C$, $\mathcal{G}_C$ and $\mathcal{G}_A$, $\mathcal{G}_A$ and delay, and delay and $\mathcal{G}_S$. The execution time of the delay actor is set to zero, and one initial token is added to the channel between the delay actor and the source of $\mathcal{G}_S$ to enforce sequential implementation of the pipe (see Fig. 10). The latency of the resulting SDFG can be configured by an appropriate choice of the execution time of the delay actor (which can be set using the re-timing transformation given in Definition 6). The model transformation results in an SDFG whose latency is equal to the inverse of throughput.

Definition 1: **(Create pipe** $Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A)$**)** Transformation $Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A)$ creates a single pipe and sequentialises the graph execution (by restricting pipelining). $Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A) = (\mathcal{A}',\ \mathcal{C}',\ e',\ r_p',\ r_c',\ i')$ with

$$
\begin{aligned}
\mathcal{A}' &= \mathcal{A}_S \cup \mathcal{A}_C \cup \mathcal{A}_A \cup \{\text{delay}\},\\
\mathcal{C}' &= \mathcal{C}_S \cup \mathcal{C}_C \cup \mathcal{C}_A\\
&\quad \cup \{c_1 = (a_{snk,S}, a_{src,C}),\ c_2 = (a_{snk,C}, a_{src,A}),\\
&\quad c_3 = (a_{snk,A}, \text{delay}),\ c_4 = (\text{delay}, a_{src,S})\},\\
e' &= e_S \cup e_C \cup e_A \cup \{(\text{delay}, 0))\},\\
r_p' &= r_{p_S} \cup r_{p_C} \cup r_{p_A} \cup \{(c_1, 1), (c_2, 1), (c_3, 1), (c_4, 1)\},\\
r_c' &= r_{c_S} \cup r_{c_C} \cup r_{c_A} \cup \{(c_1, 1), (c_2, 1), (c_3, 1), (c_4, 1)\},\\
i' &= i_S \cup i_C \cup i_A \cup \{(c_1, 0), (c_2, 0), (c_3, 0), (c_4, 1)\}.
\end{aligned}
$$

The *Pipe* transformation is essential to compute sensor-to-actuator delay τ for our implementations. To compute τ_i for a workload scenario s_i: i) compute $Pipe(RepA(\mathcal{G}_{S_i}, \varphi_{S_i}), RepA(\mathcal{G}_{C_i}, \varphi_{C_i}), RepA(\mathcal{G}_{A_i}, \varphi_{A_i}))$; ii) map the transformed graph to the given platform allocation to obtain the binding-aware graph $\mathcal{G}_i^b$; and iii) compute the latency of $\mathcal{G}_i^b$. This latency value is equal to τ_i.

The replicate-pipe transformation is an intermediate step in the model transformation, where we replicate the entire pipe to enable pipelining (see Fig. 10, $Rep(g_1, 2)$). Since each actor can be mapped to only one processing core, implementing pipelining on multiple processing cores is challenging without replication of a single pipe. $Rep(\mathcal{G}, d)$ facilitate the modelling for a pipelined implementation with d pipes. Recall that a pipe is created from $\mathcal{G}_S$, $\mathcal{G}_C$ and $\mathcal{G}_A$ (see Definition 1). These subgraphs in a single pipe are all replicated d times by the replicate-pipe transformation. The *Rep* transformation does not use the specifics of the graph it is applied to. It generically replicates the entire SDFG the specified number of times.

Definition 2 (Replicate Pipe $Rep(\mathcal{G}, d)$*):* Let $\mathcal{G} = (\mathcal{A},\ \mathcal{C},\ e,\ r_p,\ r_c,\ i)$ be an SDFG. Transformation $Rep(\mathcal{G}, d)$ replicates the SDFG d times resulting in $Rep(\mathcal{G}, d) = (\mathcal{A}',\ \mathcal{C}',\ e',\ r_p',\ r_c',\ i')$ with

$$
\begin{aligned}
\mathcal{A}' &= \bigcup_{a \in \mathcal{A}} \{a_j \mid 1 \le j \le d\}^2,\\
e' &= \{(a_j, e(a)) \mid a \in \mathcal{A},\ 1 \le j \le d\},\\
\mathcal{C}' &= \bigcup_{c \in \mathcal{C}} \{c_j \mid 1 \le j \le d\},\\
i' &= \{(c_j, i(c)) \mid c \in \mathcal{C},\ 1 \le j \le d\},\\
r_p' &= \{(c_j, r_p(c)) \mid c \in \mathcal{C},\ 1 \le j \le d\},\\
r_c' &= \{(c_j, r_c(c)) \mid c \in \mathcal{C},\ 1 \le j \le d\}.
\end{aligned}
$$

The three following transformations - adding camera-awareness, workload-awareness and inter-frame dependencies - assume that the replicate-pipe transformation has been performed with replication factor d on a single pipe

[2]This union replicates every actor in the original graph d times. For every actor a in $\mathcal{A}$, we create $a_1, \ldots, a_d$ actors in the new graph.

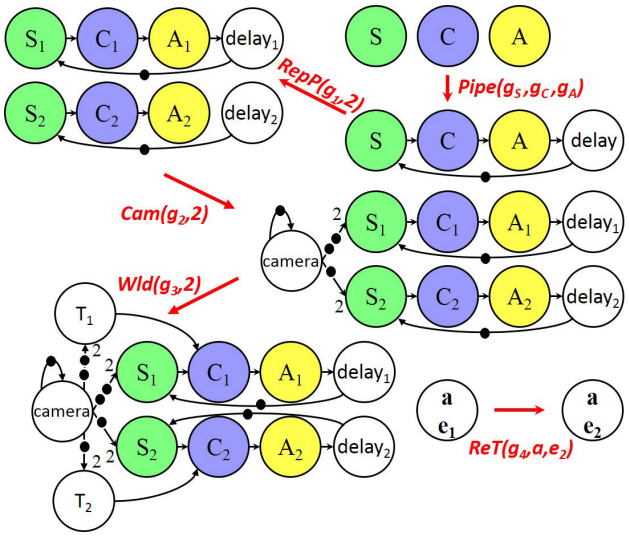

FIGURE 10. Illustration of model transformations.

created with transformation *Pipe*, optionally preceded by replicate-actor transformations (which do not affect these transformations).

The **Camera-awareness** transformation is an intermediate step in our model transformations. It adds a camera actor with execution time equal to the inverse of the given camera frame rate (the frame arrival period f_h), a self-edge with an initial token to model the frame arrival, and channels from the camera actor to the d replicated source actors a_{src,S_j} with the channel consumption rate equal to the number of replications d (see Fig. 10, $Cam(g_2, 2)$). The number of initial tokens in these channels are set to enforce an ordering of the pipes in the pipelined implementation.

Definition 3 (Camera-Awareness $Cam(\mathcal{G}, d)$): Let $\mathcal{G} = (\mathcal{A}, \mathcal{C}, e, r_p, r_c, i)$ be the SDFG $Rep(Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A), d)$. Transformation $Cam(\mathcal{G}, d) = (\mathcal{A}', \mathcal{C}', e', r_p', r_c', i')$ enforces a camera frame rate, with

$$\begin{aligned}
\mathcal{A}' &= \mathcal{A} \cup \{\text{camera}\},\ e' = e \cup \{(\text{camera}, f_h)\},\\
\mathcal{C}' &= \mathcal{C} \cup \{(\text{camera})^2\}\\
&\quad \cup \{c_j = (\text{camera}, a_{src,S_j}) \mid 1 \le j \le d\},\\
r_p' &= r_p \cup \{((\text{camera})^2, 1)\} \cup \{(c_j, 1) \mid 1 \le j \le d\},\\
r_c' &= r_c \cup \{((\text{camera})^2, 1)\} \cup \{(c_j, d) \mid 1 \le j \le d\},\\
i' &= i \cup \{((\text{camera})^2, 1)\} \cup \{(c_j, d - j + 1) \mid 1 \le j \le d\}.
\end{aligned}$$

The **Workload-awareness** transformation is a step in the model transformations performed on graph $Cam(Rep(Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A), d), d)$. Due to image-workload variations, the sensing task's runtime execution times are varying. Also, for the SPADe implementation, the system scenarios may abstract multiple workload scenarios with varying execution times for the sensing task. However, the SPADe controller design requires a constant sensor-to-actuator delay per system scenario for the implementation. The *Wld* transformation enforces a constant sensor-to-actuator delay for our implementation. The *Wld* transformation adds actors T_j with an incoming channel from the camera actor and an outgoing channel to the (replicated) source actor a_{src,C_j} of the computation SDFG $\mathcal{G}_C$. The consumption rate and initial tokens for the channels from camera actor to T_j are the same as for the channel from camera actor to source actors in the *Cam* transformation, again to enforce ordering in the pipelined execution (see Fig. 10, $Wld(g_3, 2)$). The T_j actors create a path in parallel to the $\mathcal{G}_S$ graph instances. By setting the execution time of these added T_j actors to an appropriately large value, a constant sensor-to-actuator delay can be enforced. The *Wld* transformation sets the execution time to 0. The execution-time value can be updated when needed in the SPADe flow by the re-timing transformation introduced below.

Definition 4 (Workload-Awareness $Wld(\mathcal{G}, d)$): Let $\mathcal{G} = (\mathcal{A}, \mathcal{C}, e, r_p, r_c, i)$ be the SDFG $Cam(Rep(Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A), d), d)$. Transformation $Wld(\mathcal{G}) = (\mathcal{A}', \mathcal{C}', e', r_p', r_c', i')$, with

$$\begin{aligned}
\mathcal{A}' &= \mathcal{A} \cup \{T_j \mid 1 \le j \le d\},\\
e' &= e \cup \{(T_j, 0) \mid 1 \le j \le d\},\\
\mathcal{C}' &= \mathcal{C} \cup \{cT_j = (\text{camera}, T_j) \mid 1 \le j \le d\}\\
&\quad \cup \{TC_j = (T_j, a_{src,C_j}) \mid 1 \le j \le d\},\\
r_p' &= r_p \cup \{(cT_j, 1) \mid 1 \le j \le d\}\\
&\quad \cup \{(TC_j, 1) \mid 1 \le j \le d\},\\
r_c' &= r_c \cup \{(cT_j, d) \mid 1 \le j \le d\}\\
&\quad \cup \{(TC_j, 1) \mid 1 \le j \le d\},\\
i' &= i \cup \{(cT_j, d - j + 1) \mid 1 \le j \le d\}\\
&\quad \cup \{(TC_j, 0) \mid 1 \le j \le d\}.
\end{aligned}$$

The **inter-frame-dependency** transformation adds channels to enforce the dependencies for actor firings between two consecutive pipes. E.g., an actor b_j that executes in the k-th pipe might depend on the completion of execution of an actor a_i that executes in the $(k-1)$-th pipe. This transformation is optionally done after *Cam* (and has no further effect on the definition of the earlier transformations). An example for this transformation is illustrated in Fig. 11. Ideally, our model transformations ensure that the inverse throughput of the implementation-aware graph is equal to the execution time of the camera actor. E.g. if $e(\text{camera}) = f_h$, then the throughput of the implementation-aware graph is equal to (or limited by) the camera frame rate $\frac{1}{f_h}$. Now, the *ifd* transformation allows the throughput to be limited also by the inter-frame dependencies. The inverse throughput of the implementation-aware graph will then be equal to the maximum of $e(\text{camera})$ and the inter-frame dependence time f_d (as explained later in Section VI-D1).

Definition 5 (Inter-Frame Dependency $ifd(\mathcal{G}, a, b, d)$): Let $\mathcal{G} = (\mathcal{A}, \mathcal{C}, e, r_p, r_c, i)$ be the SDFG $Cam(Rep(Pipe(\mathcal{G}_S, \mathcal{G}_C, \mathcal{G}_A), d), d)$. Transformation $ifd(\mathcal{G}, a, b, d) = (\mathcal{A}, \mathcal{C}', e, r_p', r_c', i')$ adds inter-frame dependencies between

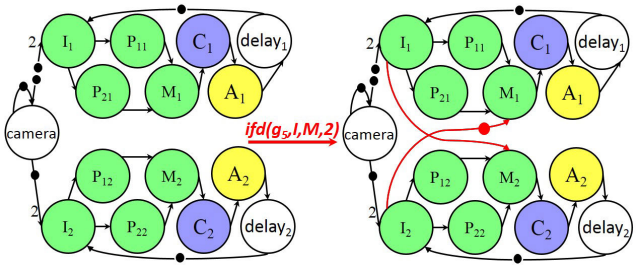

FIGURE 11. Illustration of inter-frame dependencies between the actors I and M. The channels added by *ifd*$(g_5, I, M, 2)$ are shown in red.

actors a_i and b_j, for $a_i, b_j \in \mathcal{A}$, with

$$\mathcal{C}' = \mathcal{C} \cup \{c_d = (a_d, b_1)\} \cup \{c_j = (a_j, b_{j+1}) \mid 1 \leq j < d\},$$
$$r'_p = r_p \cup \{(c_d, \rho(b_1))\} \cup \{(c_j, \rho(b_{j+1})) \mid 1 \leq j < d\},$$
$$r'_c = r_c \cup \{(c_d, \rho(a_d))\} \cup \{(c_j, \rho(a_j)) \mid 1 \leq j < d\},$$
$$i' = i \cup \{(c_d, \rho(a_d)))\} \cup \{(c_j, 0) \mid 1 \leq j < d\}.$$

During SPADe analysis, the τ and h values are updated based on our implementation choices. The **re-timing** transformation helps to update the execution time of the actors (camera, delay, and T) in our models when required (see Fig. 10, $ReT(g_4, a, e_2)$).

Definition 6 (Re-Timing $ReT(\mathcal{G}, a, t)$): Let $\mathcal{G} = (\mathcal{A}, \mathcal{C}, e, r_p, r_c, i)$ be an SDFG. Transformation $ReT(\mathcal{G}, a, t) = (\mathcal{A}, \mathcal{C}, e', r_p, r_c, i)$ updates the execution time of actor $a \in \mathcal{A}$ to $t \in \mathbb{R}_{\geq 0}$, with

$$e' = e \setminus \{(a, e(a))\} \cup \{(a, t)\}.$$

VI. SPADe FLOW REVISITED

This section makes the SPADe design flow (illustrated in Figure 6 and introduced in Section IV) precise in the form of Algorithm 2, using the model transformations of Section V. The model transformations are primarily used to construct implementation-aware graphs for workload and system scenarios. The integrated transformation from an IBC graph to an implementation-aware graph is captured in Algorithm 1. Due to platform resource constraints (the number of available cores n_c^{avl}), design choices (number of pipes p, number of cores allocated per pipe $n_c^{//}$), application characteristics, inter-frame dependencies (inter-frame dependence time f_d) and the possible camera frame-arrival period (f_h), the effective implementation can be: i) a non-pipelined implementation without parallelised sensing; ii) a non-pipelined implementation with parallelised sensing; iii) a pipelined implementation without parallelised sensing; and iv) a pipelined parallelism implementation. Section VI-A explains Algorithm 1 for constructing implementation-aware graphs. Section VI-B elaborates SPADe in Algorithm 2. We then explain the refinements for non-pipelined and pipelined implementations in Sections VI-C and VI-D. The differentiation between non-pipelined and pipelined implementation is mainly needed for control design and switching. Further, we also explain the challenge due to inter-frame dependencies in a pipelined implementation in Section VI-D1.

Algorithm 1 impAwGrTrans($\mathcal{G}, \tau, h, e_{CA}, p$)

input : $\mathcal{G}$, τ, h, e_{CA}, p, IFD (the set of known inter-frame dependencies)
output: Implementation-aware graph $\mathcal{G}_I$

1 $\mathcal{G}_1 = ReT(\mathcal{G}, \text{delay}, (p \times h - \tau))$;
2 $\mathcal{G}_3 = Cam(Rep(\mathcal{G}_1, p), p)$;
3 $\mathcal{G}_{31} = ReT(\mathcal{G}_3, \text{camera}, h)$
4 **if** $p > 1$, *i.e. pipelining is allowed* **then**
5 **foreach** $(a, b) \in IFD$ **do**
6 $\mathcal{G}_{31} = ifd(\mathcal{G}_{31}, a, b, p)$;
7 **end**
8 **end**
9 $\mathcal{G}_4 = Wld(\mathcal{G}_{31}, p)$;
10 $\mathcal{G}_I = ReT(\mathcal{G}_4, T_j, (\tau - e_{CA}))$, $1 \leq j \leq p$;

A. IMPLEMENTATION-AWARE GRAPH TRANSFORMATION

In this section, we explain the steps needed to obtain the implementation-aware graph of a workload or system scenario and formalise those in Algorithm 1. The **input** SDFG is the SDFG of a single parallelised pipe of a single scenario, obtained after applying the *Pipe* transformation (as explained in Section V), as illustrated in Figure 10. The other inputs to the algorithm are the delay τ, period h, the total execution time of the control compute and actuation tasks e_{CA}, and the number of pipes p. Also, we assume that if there exist inter-frame dependencies, they are known as a subset of actors IFD, $IFD \subseteq \mathcal{A}^2$. An ordered pair of actors $(a, b) \in IFD$ models the inter-frame dependency between the actors a and b. The **output** of the algorithm is the implementation-aware graph $\mathcal{G}_I$.

Step 1 ensures that we can achieve a constant sensor-to-actuator delay during mapping by assigning the execution time of the delay actor as $p \times h - \tau$ in $\mathcal{G}$ to obtain $\mathcal{G}_1$. The delay actor fills up the time between when the actuation task (modelled by actor A) finishes its execution until the completion of one pipe (see Fig. 10). This ensures that each parallelised pipe, when mapped to the platform, can periodically execute with the period $p \times h$ and has a constant delay of τ. If we have p pipes, we can ensure that the effective control sampling period is h. **Step 2** replicates the single parallelised pipe model $\mathcal{G}_1$ p times to model the pipelined execution and then adds camera-awareness to the graph $\mathcal{G}_1$ as explained in Section V. **Step 3** updates the execution time of the camera actor in line with the sampling period h. Our model has to execute with the period h even though the camera frame arrival period is f_h. h, however, is a multiple of f_h so that we can align the arrival of camera frames with the sampling period.

If pipelining is allowed, i.e., if $p > 1$, and there exist inter-frame dependencies that are known as ordered pairs of actors IFD, the *ifd* transformation, explained in Section V and in Fig. 11, is applied for each of the dependencies (see **Step 4** and the for loop in **Step 5**). **Step 9** adds workload-awareness to the resulting graph $\mathcal{G}_3$, also explained in Section V. The

Wld transformation adds the actors T_j whose execution times should be equal to the sensor-to-actuator delay minus the execution times of the control compute and actuate tasks e_{CA} (an input to the algorithm). We obtain the final refined implementation-aware graph $\mathcal{G}_I$ after updating the execution time of actors T_j in **Step 10** with $e(T_j) = \tau - e_{CA}$ so that we can ensure that the control compute task starts at the right time and is not affected by the workload variations in sensing.

B. UNIFIED SPADe FLOW FOR PIPELINED PARALLELISM

The SPADe flow of Figure 6 is made precise in Algorithm 2. Algorithm 2 captures the design-time formal modelling, analysis and design for the SPADe flow. The **outputs** of the design flow are the system configurations and the LUT for runtime implementation. Runtime implementation for SPADe has been explained earlier in Section IV-C. The **inputs** to the SPADe flow are the camera frame rate f_h, the number of pipes p, the number of cores for parallelism per pipe $n_c^{//}$, the application IBC SADFG $(\Sigma, \mathcal{F})$ (satisfying the assumptions given in Section V), and the maximum number of available cores n_c^{avl}. 'Map $\mathcal{G}$' denotes the mapping of the SDFG graph $\mathcal{G}$ to the given platform allocation, as mentioned in **Step 1**. In our implementation, the mapping is done using the SDF3 [14] tool. But any mapping tool can be used that ensures a mapping onto the platform that guarantees the maximal throughput obtainable by the SADFG being mapped.

We explain the steps in Algorithm 2 in relation to Fig. 6. The 'for loop' in **Step 2** derives, for each workload scenario s_i, the initial implementation-aware IBC graph $\mathcal{G}_{11_i}$ (Steps 3,4), mapping of the control compute and actuation tasks to obtain $\mathcal{G}^b_{CA_i}$ (Step 5), mapping of the initial implementation-aware graph $\mathcal{G}_{11_i}$ to obtain the binding-aware graph $\mathcal{G}^b_{11_i}$ (Step 6), and timing analysis for computing τ_i and h_i (Steps 7, 8). If pipelining is enabled, we refine the implementation-aware graph using the timing analysis information and the implementation choice p (Step 10) to compute the inter-frame dependence time for the scenario f_d^i (Step 13). The 'for loop' in Step 2 is illustrated in Fig. 6 from the implementation-aware IBC graph node to the timing analysis block and back. The model transformations required to compute the implementation-aware graphs have already been illustrated in Fig. 10 and Fig. 11 and the transformation of Step 10 has been made precise in Algorithm 1.

Steps 3 and 4 create a model of a single parallelised pipe $\mathcal{G}_{11_i}$, as explained in Section V. The parallelisation transformations are optional. As explained in Section V, the parallelisation may also be done manually. **Step 5** maps the control compute and actuate tasks to the given platform allocation to compute its execution time e_{CA}. If the control compute and actuate tasks are single actors C and A respectively, then $e_{CA} = e(C) + e(A)$. However, if the control compute and actuate tasks are parallelised (sub)graphs $\mathcal{G}_C$ and $\mathcal{G}_A$, we need to find the latency from the combined binding-aware graph after the *RepA* and *Pipe* transformations (**Step 5**). In this case, the latency is equal to the inverse throughput due to

Algorithm 2 SPADeFlow(f_h, p, $n_c^{//}$, $(\Sigma, \mathcal{F})$, n_c^{avl})

input : f_h, p, $n_c^{//}$, $(\Sigma, \mathcal{F})$ (SADFG), n_c^{avl} (platform)
output: System configurations $\chi^s_{s_s}$, LUT

1 Let 'Map $\mathcal{G}$' denote the mapping of an SDFG $\mathcal{G}$ to the given n_c^{avl} cores using the SDF3 tool;
2 **foreach** *workload scenario* $s_i \in \Sigma$ **do**
3 $\quad \varphi_{X_i} = min(\rho_{X_i}, n_c^{//})$, where $X_i \in \{S_i, C_i, A_i\}$ and ρ_{X_i} is the repetition vector of X_i;
4 $\quad \mathcal{G}_{11_i} = Pipe(RepA(\mathcal{G}_{S_i}, \varphi_{S_i}), RepA(\mathcal{G}_{C_i}, \varphi_{C_i}), RepA(\mathcal{G}_{A_i}, \varphi_{A_i}))$;
5 $\quad \mathcal{G}^b_{CA_i} \leftarrow$ Map $Pipe(RepA(\mathcal{G}_{C_i}, \varphi_{C_i}), RepA(\mathcal{G}_{A_i}, \varphi_{A_i}))$;
6 $\quad \mathcal{G}^b_{11_i} \leftarrow$ Map $\mathcal{G}_{11_i}$;
7 $\quad \tau_i = \mathcal{L}(s_i^\omega, \mathbf{0}, \frac{1}{\nu(\mathcal{G}^b_{11_i})})$;
8 $\quad h_i = \left\lceil \frac{\tau_i}{f_h \times p} \right\rceil f_h$;
9 $\quad$ **if** $p > 1$, *i.e. pipelining is allowed* **then**
10 $\quad\quad \mathcal{G}_{I_i} = \text{impAwGrTrans}(\mathcal{G}_{11_i}, \tau_i, h_i, \frac{1}{\nu(\mathcal{G}^b_{CA_i})}, p)$;
11 $\quad\quad \mathcal{G}_{32_i} = ReT(\mathcal{G}_{I_i}, \text{camera}, 0)$;
12 $\quad\quad \mathcal{G}^b_{32_i} \leftarrow$ Map $\mathcal{G}_{32_i}$;
13 $\quad\quad f_d^i = \frac{1}{\nu(\mathcal{G}^b_{32_i})}$;
14 $\quad$ **end**
15 **end**
16 **if** $p > 1$, *i.e. pipelining is allowed* **then**
17 $\quad \tau_{wc} = \max_i \tau_i$; $h_{wc} = \max_i h_i$;
18 $\quad f_d = \max_i f_d^i$;
19 $\quad n_s = \max\left(\left\lceil \frac{f_d}{f_h} \right\rceil, 1\right)$;
20 $\quad n_{f_{wc}} = \left\lceil \frac{\tau_{wc}}{f_h} \right\rceil$;
21 $\quad p_{max} = \left\lceil \frac{n_{f_{wc}}}{n_s} \right\rceil$;
22 $\quad n_{c_{max}} = n_c^{//} \times p_{max}$;
23 $\quad h_{min} = \begin{cases} n_s \times f_h, & \text{if } n_c^{avl} \geq n_{c_{max}}, \\ \left\lceil \frac{n_{c_{max}}}{n_c^{avl}} n_s \right\rceil \times f_h, & \text{otherwise;} \end{cases}$
24 $\quad h_{eff} = \max(h_{min}, h_{wc})$;
25 **end**
26 Controller design and system-scenario identification:
if p>1, see Sections VI-D3 and VI-D4;
else see Sections VI-C and IV-B4;
27 $s_s \leftarrow$ identified system scenarios with (τ_s, h_s);
28 $\tau_{s_{wc}} = \max_s \tau_s$; $h_{s_{wc}} = \max_s h_s$;
29 $s_{s_{wc}} = \arg\max_{s_s} \tau_s$;
30 $p_s = \left\lceil \frac{\tau_{s_{wc}}}{h_{s_{wc}}} \right\rceil$; **foreach** *identified system scenario* s_s *with* (τ_s, h_s) **do**
31 $\quad \mathcal{G}_{11_s} \leftarrow \mathcal{G}_{11_i}$ of the s_i in s_s with max τ_i;
32 $\quad \mathcal{G}^b_{CA_s} \leftarrow \mathcal{G}^b_{CA_i}$ of the s_i in s_s with max τ_i;
33 $\quad \mathcal{G}_{I_s} = \text{impAwGrTrans}(\mathcal{G}_{11_s}, \tau_s, h_s, \frac{1}{\nu(\mathcal{G}^b_{CA_s})}, p_s)$;
34 $\quad \mathcal{G}^b_s \leftarrow$ Map $\mathcal{G}_{I_s}$;
35 $\quad$ **if** $h_s = \frac{1}{\nu(\mathcal{G}^b_s)}$ **then**
36 $\quad\quad \chi^s_{s_s} = (\mathcal{G}^b_s, h_s, \tau_s, K_s, F_s)$;
37 $\quad$ **else**
38 $\quad\quad$ // the mapping of s_s is not feasible
go to Step 26 and choose a different (sub)set of system scenarios (possibly reverting to the worst-case scenario $s_{s_{wc}}$ as the single system scenario);
39 $\quad$ **end**
40 **end**
41 Create a LUT for runtime use (as explained in Section IV-C);

the *Pipe* transformation, i.e., $e_{CA} = 1/\nu(\mathcal{G}^b_{CA_i})$. **Step 6** maps initial implementation-aware graph $\mathcal{G}_{11_i}$ to the given platform allocation to obtain the binding-aware graph $\mathcal{G}^b_{11_i}$. **Steps 7 and 8** compute the sensor-to-actuator delay τ_i and sampling period h_i for s_i from this binding-aware graph (as explained earlier in Eq. 6).

If pipelining is allowed, i.e., $p > 1$, then we go through an extra iteration of the timing-analysis loop (in Fig. 6) to compute the inter-frame dependence time for the scenario at hand, f^i_d, for the pipelined implementation (**Steps 9 - 13**). **Step 10** refines the initial implementation-aware graph for the scenario at hand with delay-awareness, pipe replication, camera-awareness, and workload-awareness through Algorithm 1 with the timing values computed in the previous steps. In order to compute the inter-frame dependence time, we then set the execution time of the camera actor to zero (Step 11) so that the inter-frame dependency is the throughput limiting factor in our graph. We map the refined graph $\mathcal{G}_{32_i}$ to obtain $\mathcal{G}^b_{32_i}$ and compute f^i_d as the inverse throughput of $\mathcal{G}^b_{32_i}$ (Step 13). A point to note is that the actors camera, delay$_j$ and T_j being added in the process are not mapped to the given platform allocation (while mapping in SDF3, we bind each of these actors to separate dummy processors). These actors are required to simulate time-triggering of tasks and ordering of pipes.

In **Steps 16 - 24**, we proceed with the timing analysis in Fig. 6 to compute the inter-frame dependence time f_d for the IBC application as a whole and the constant effective sampling period h_{eff} for a pipelined implementation (i.e. $p > 1$). Recall from Section IV-C that we enforce a constant sampling period for the pipelined implementation to limit the design space to be explored, reduce runtime overhead, and facilitate controller design. The computation of h_{eff} starts with determining the worst-case delay τ_{wc}, worst-case period h_{wc}, and corresponding inter-frame dependence time f_d. We can then compute the maximum number of pipes feasible p_{max} due to inter-frame dependencies and the maximum number of cores we require $n_{c_{max}}$ to realise p_{max}. We can then compute the smallest realisable sampling period h_{min}, after which we set h_{eff} to the maximum of h_{min} and h_{wc}.

Step 17 determines the worst-case delay τ_{wc} and worst-case period h_{wc}. Because delay and sampling period are determined from a single pipe, the scenario with the largest delay also has the largest sampling period. Next, we compute the maximum inter-frame dependence time f_d (**Step 18**) over all the workload scenarios s_i. The maximum (and not any other) inter-frame dependence time is considered for further analysis since the order of the workload scenario sequence at runtime is not known apriori. Because of inter-frame dependencies, not all frames can be used for sensing. With n_s as computed from f_d and f_h as indicated in **Step 19**, $n_s - 1$ is the effective number of frames skipped between processing the arriving camera frames due to the inter-frame dependencies. The maximum operation is required to avoid a corner case in the subsequent analysis when $f_d = 0$. $n_{f_{wc}}$ (**Step 20**) is the number of camera frames arriving within any worst-case sensor-to-actuator delay interval for the single pipe execution. The realisable maximal number of pipes p_{max} captures the maximum number of pipes possible for our pipelined implementation considering the frames we have to skip due to inter-frame dependencies and the total number of frames arriving within the worst-case delay interval $n_{f_{wc}}$ (see **Step 21**). We can then compute the maximum number of cores required for realising our design choices of $n^{//}_c$ and p_{max} during runtime implementation as $n_{c_{max}}$ (**Step 22**). For instance, if we allocate two cores per pipe for parallelism and we would like to have two pipes, then we need a maximum of four cores. h_{min} is then the minimum realisable sampling period possible for the controller implementation considering the given choice of parameters; it can be computed as shown in **Step 23**. If more cores are allocated than the maximum number of cores required to realise our design choices, i.e., $n^{avl}_c \geq n_{c_{max}}$, then h_{min} is limited only by the inter-frame dependencies, as captured by n_s. In this case, the SPADe implementation utilises a maximum of $n_{c_{max}}$ cores, as having more cores does not improve h_{min} and, in effect, does not improve the control performance. However, if the resources we require to realise a sampling period of $n_s \times f_h$ are not allocated, i.e., $n^{avl}_c < n_{c_{max}}$, then h_{min} has to be increased proportionally to the fraction $\frac{n_{c_{max}}}{n^{avl}_c}$. E.g., let $n_{c_{max}} = 4$, $n_s = 1$. If $n^{avl}_c \geq 4$, we can achieve the sampling period $h_{min} = f_h$. However, if $n^{avl}_c = 2$, we cannot realise $h_{min} = f_h$. In this case, we increase h_{min} as many times as the fraction $\frac{4}{2}$, i.e., h_{min} becomes $\frac{4}{2} n_s \times f_h = 2 f_h$. The effective realisable sampling period h_{eff} is finally taken as the maximum of h_{min} and h_{wc} (**Step 24**).

Steps 26 - 38 design controllers for the workload scenarios, identify the system scenarios s_s, derive the binding-aware graph $\mathcal{G}^b_s$ for s_s, check feasibility of the scenario definitions, and define the system configurations. These steps are illustrated in Fig. 6 using the blocks controller design, system-scenario identification, and system configurations. For a non-pipelined implementation, controllers are designed as explained in Section VI-C and the system-scenario identification is done as explained in Section IV-B4. For a pipelined or pipelined-parallelism implementation, controller design and system-scenario identification are explained in Sections VI-D3 and VI-D4, respectively. Recall that if we cannot guarantee the stability of the switched system being defined, our controller design reverts to a periodic worst-case-based design with a single worst-case system scenario. **Step 29** identifies this worst-case system scenario $s_{s_{wc}}$ as the scenario with the largest delay $\tau_{s_{wc}}$ (and hence also the largest period). If any of the identified system scenarios cannot be mapped onto the allocated resources in such a way that all timing requirements are met, then SPADe reverts to this worst-case scenario as the only system scenario as well (**Step 38**). **Step 30** computes the realisable number of pipes, i.e. the effective number of pipes in implementation, based on the worst-case timing analysis, controller design and scenario identification. p_s is always less than or equal to p.

For each of the identified system scenarios, **Steps 31 - 34** derive its binding-aware graph. We start from the initial implementation-aware graph of the contributing workload scenario with the maximum delay (**Step 31**). Then we identify the contributing workload scenario's binding-aware graph for the control compute and actuation tasks (**Step 32**). **Step 33** refines the initial implementation-aware graph with the updated timing information on delay τ_s, period h_s, execution time for the control compute and actuate tasks ($e_{CA_s} = 1/\nu(\mathcal{G}^b_{CA_s})$ and the realisable number of pipes p_s using Algorithm 1. Finally, we map the updated implementation-aware graph to the platform (see **Step 34**) and check if the control timing is realisable in the binding-aware graph (**Step 35**). Control timing is realisable if a feasible mapping exists for the binding-aware graph, and a feasible mapping implies that the inverse throughput of the $\mathcal{G}^b_s$ is equal to h_s. If a feasible mapping exists, we can define the system configuration $\chi^s_{s_s}$ (**Step 36**) for the system scenario s_s. If the mapping is infeasible, we need to choose a different subset of system scenarios and re-do the controller design as explained earlier (**Step 38**).

Once the feasible system scenarios have successfully been identified, we create the LUT in **Step 41**, as explained in Section IV-C. A design-space exploration (DSE) (illustrated in Fig. 6) is performed if we want to explore which implementation choice gives the best control performance. In this paper, we consider a brute-force DSE by varying the inputs to Algorithm 2, and analysing the performance of the resulting system configurations.

C. SPADe CONTROL DESIGN AND SWITCHING FOR NON-PIPELINED IMPLEMENTATION

This section explains the controller design, in particular, system augmentation and switching, for non-pipelined implementation ($p = 1$). Recall from Section IV-B4 that we aggregate workload scenarios based on the camera frame rate to limit the number of switching scenarios. Controllers are designed for the aggregated workload scenarios and system scenarios are identified as the switching-stable aggregated workload scenarios as explained in Section IV-B4.

The control timing parameters τ_i and h_i for any scenario s_i are computed during the SPADe analysis and design (see Steps 7 and 8 of Algorithm 2). For the non-pipelined implementation, $\tau_i \leq h_i$ for all scenarios s_i, which means that we need only one augmented delay state. First, Eq. 1 can be reformulated in the discrete-time domain for a given (τ_i, h_i) as follows:

$$\begin{aligned} x[k+1] &= A_{s_i}x[k] + B_{0,s_i}u[k] + B_{1,s_i}u[k-1], \\ y[k] &= C_c x[k] + D_c u[k] \end{aligned} \tag{11}$$

$$\text{where, } A_{s_i} = e^{A_c h_i}, \quad B_{0,s_i} = \int_0^{h_i-\tau_i} e^{A_c s} ds \cdot B_c,$$

$$B_{1,s_i} = \int_{h_i-\tau_i}^{h_i} e^{A_c s} ds \cdot B_c. \tag{12}$$

Note that the discretization of the system using τ_i and h_i does not affect the output and feedforward matrices C_c and D_c [45]. 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:

$$\begin{aligned} z[k+1] &= A_{aug,s_i}z[k] + B_{aug,s_i}u[k], \\ y[k] &= C_{aug}z[k] + D_c u[k], \end{aligned}$$

$$\text{where } A_{aug,s_i} = \begin{bmatrix} A_{s_i} & B_{1,s_i} \\ 0 & 0 \end{bmatrix}, \quad B_{aug,s_i} = \begin{bmatrix} B_{0,s_i} \\ I \end{bmatrix}, \tag{13}$$

$C_{aug} = \begin{bmatrix} C_c \; 0 \end{bmatrix}$. 0 and I represent the zero and identity matrices of appropriate dimensions. Controllers are then designed for this higher-order augmented system, as explained in Sec. IV-B3.

Switching due to workload variations and switching stability has been explained in Section IV-B4. It was explored in [2], [7] for non-pipelined implementation. Having numerous switching scenarios often results in instability [39] and degrades control performance due to the non-smooth response associated with the switching overhead [7]. For optimising control performance and stability, it is essential that we limit the number of switching scenarios through system-scenario identification (as explained in Section IV-B4).

For a non-pipelined implementation, switching results in both variable delay and variable period. SPADe flow for the non-pipelined implementation does not have any restrictions on the value of τ_s we can have for a system scenario. h_s can vary, but should always be a multiple of f_h to align the start of the sensing task with the camera frame rate.

D. SPADe REFINEMENTS FOR PIPELINED IMPLEMENTATION

For a pipelined implementation, i.e., for $p > 1$, we have that $\tau_{wc} > h_{eff}$ where τ_{wc} is the worst-case sensor-to-actuator delay, and $0 < \tau_i \leq \tau_{wc}$. For the scope of this work, we enforce a constant sampling period h_{eff} (which is a multiple of f_h) for the overall pipelined implementation. The constant sampling period means that we ensure a constant start of sensing and also, a constant actuation rate. The computation of h_{eff} depends on the inter-frame dependencies, τ_{wc}, f_h, p, $n_c^{//}$, and n_c^{avl} (see Algorithm 2 for the precise details). This section explains the significance of inter-frame dependencies, switching due to image workload variations in pipelining, controller design, system-scenario identification, the need for implementation-aware matrices and how we compute control configuration for pipelined implementation.

1) INTER-FRAME DEPENDENCIES IN A PIPELINED IMPLEMENTATION

Pipelining is inherently limited by inter-frame dependencies, i.e., the data or algorithmic dependencies between consecutive frame processing, e.g., due to video coding [10] or

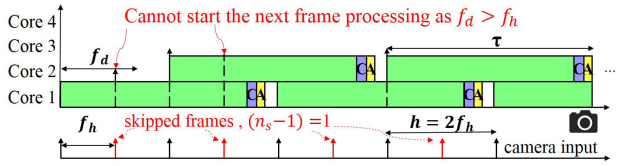

FIGURE 12. Illustration of inter-frame dependencies with $f_h < f_d \leq 2f_h$.

visual tracking [11]. Considering inter-frame dependencies is crucial for a practical pipelined implementation. Inter-frame dependence time (denoted by f_d) can be quantified for the current image frame as the maximum time required to complete the processing of (parts of) the IBC algorithm the subsequent image frame processing depends on. Alternatively, f_d is the minimum time required to wait between the start of processing consecutive image frames. Fig. 12 illustrates the impact of inter-frame dependence time on sampling period h. In a pipelined implementation, considering inter-frame dependencies means that strictly $h_{eff} \geq f_d$. The number of frames that has to be skipped after processing every frame is $n_s - 1$ with n_s computed as in Step 19 of Algorithm 2. This is illustrated in Fig. 12 where $n_s = 2$ and one frame is skipped after every frame processing.

The computation of f_d is explained in Algorithm 2. The inter-frame dependencies are modelled using the *ifd* model transformation explained in Section V. Computing f_d helps to determine the effective image arrival period or the minimum possible sampling period h_{min} we can have. Inter-frame dependencies mean that sometimes image frames have to be skipped for processing with respect to the given image arrival period f_h and the sampling period h. Skipping a frame means that h increases and thus degrades the control performance. In some cases, e.g., when the sensing uses video coding, inter-frame dependencies limit the effective camera frame rate and the video needs to be encoded/decoded at the effective rate $(1/h_{eff})$. In case the video encoding is closed-source and the video encoding cannot be done at a new rate, then sufficient resources need to be allocated first for decoding at the original camera frame rate and only the remaining resources can be utilised for the rest of the application. In this case, the video decoding is periodically executed at the camera frame rate and mapped first to the given platform allocation. The IBC application is then mapped to the remaining allocation.

For a non-pipelined implementation, the inter-frame dependencies can be ignored since the frames are processed in sequence.

2) SWITCHING IN PIPELINED IMPLEMENTATION

Switching in a multiprocessor pipelined IBC system implementation due to workload variations has not been explicitly explored in literature apart from our previous work [46]. When we do not consider workload variations, a pipelined implementation effectively results in a constant τ and h. Considering workload variations implies that we would have varying sensing delays, e.g. as illustrated in Figure 13. Here,

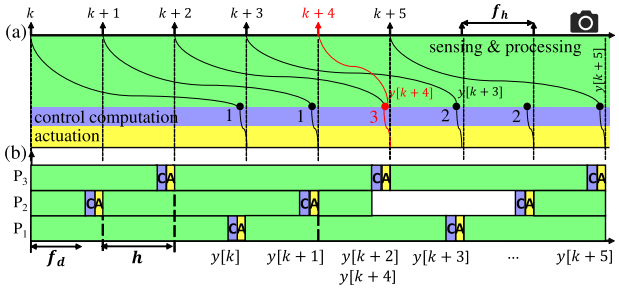

FIGURE 13. Illustration of switching due to workload variations in a multiprocessor pipelined implementation. (a) Logical delay diagram illustrating the cases explained in Sec VI-D2 and (b) its corresponding Gantt chart. The sample $k+4$ has a lower workload and thus the latest output measurement $y(k+4)$ is available within one f_h.

notice that the camera input frame at $k + 4$ has a sensing delay of one frame ($\tau_1 = h$) due to lower image workload, and all other frames have a sensing delay of three frames ($\tau = 3h$). This scenario results in multiple sensing and image processing (S) tasks completing their execution at the same time. This means that multiple output measurements $y[k+2]$, and $y[k+4]$ are available for control computation task C at the same time instance. For the scope of this work, the sampling period is kept constant and the switching happens due to variable delay.

Notice that by having just one frame with a lower workload, we can have the following three switching cases as illustrated in Fig. 13 (a): case 1) the new measurement is available with the same sensing delay as in the previous step; case 2) the sensing delay is increased compared to the previous step as the latest measurement is not available. Older past measurements may be available during this time (e.g. $y[k + 3]$ becomes available one period after $y[k + 4]$). However, they are used only to update the state estimates and not directly for control input computation; case 3) the sensing delay is reduced by one or more steps: when multiple pipes finish processing a corresponding sequence of frames, both the latest measurement(s) along with the past measurements are now available. The past measurements are used to update the state estimates, and the latest measurement is used to compute the control input.

Thus, the main challenge for the pipelined IBC system design in order to maximise performance, i.e. QoC, is to effectively use the sensor measurements as early as possible for control computation without any unnecessary idling and to estimate the system state when there are no sensor measurements available. Modelling this behaviour is far from trivial. This problem was explored with respect to long network delays in [47]. We leverage these results in our design.

3) CONTROL DESIGN AND SYSTEM-SCENARIO IDENTIFICATION

We consider a pipelined implementation with workload scenarios s_i having $(\tau_i,\ h_i)$ as timing parameters. It is possible to have a varying h_i similar to the non-pipelined

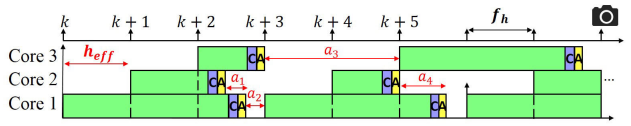

FIGURE 14. Illustration of the challenge with varying actuation rates a_i due to variable τ_i'.

implementation. However, proving stability guarantees then becomes challenging and as such is not explored in this work. For the scope of this work, we assume a constant period h_{eff} for all scenarios in the pipelined implementation. How we compute h_{eff} was explained earlier in Algorithm 2. For brevity, $h = h_{eff}$ in the rest of this section.

For a workload scenario s_i, we can represent τ_i based on [45] as

$$\tau_i = (n_{f_i} - 1)h + \tau_i', \text{ where } 0 < \tau_i' \leq h, \; n_{f_i} = \left\lceil \frac{\tau_i}{h} \right\rceil. \quad (14)$$

This representation divides the delay τ_i into n_{f_i} regions in the time domain. This results in $(n_{f_i} - 1)$ regions of h and the left-over delay τ_i' for τ_i. n_{f_i} is the number of frames arriving in one delay period for the scenario at hand. The above n_{f_i} computation assumes the practical situation where $\tau_i > 0$. In case one wants to consider $\tau_i = 0$, $n_{f_i} = max\left(\left\lceil \frac{\tau_i}{h} \right\rceil, 1\right)$.

Next, we enforce a constant actuation rate since a varying actuation rate results in undesired behaviour as illustrated in Fig. 14, even in the case of a pipelined implementation with a constant sampling period. Fig. 14 executes a scenario sequence $(s_3 s_2 s_1 s_3 s_1 s_3)^{\omega}$ with sampling period $h_{eff} = f_h$. The scenario s_1 has the best-case delay $\tau_1 = f_h$, s_2 has a delay $\tau_2 = 1.2f_h$, and s_3 has a delay $\tau_3 = 2.5f_h$. If we now apply Eq. 14, we get a varying τ_i' and the Gantt chart as illustrated in Fig. 14. Notice that the actuation rates illustrated by the a_i are not periodic anymore and we have ordering issues as well with respect to the actuation task. Further, guaranteeing controller stability for this case is challenging.

We mitigate varying actuation rates by defining $\tau' = \max_i(\tau_i')$ and then designing controllers with $\tau_i = (n_{f_i} - 1)h + \tau'$ and $h = h_{eff}$ for workload scenarios s_i. A constant τ' over all scenarios is required to maintain a constant actuation rate between switching scenarios in a pipelined implementation. Notice that by defining and fixing τ' we are aggregating workload scenarios with the same n_{f_i} (see Eq. 14). The controllers are designed for these aggregated workload scenarios. Also, the design space for system-scenario identification is narrowed down by the workload scenario aggregation.

To design the controllers, Eq. 1 can then be reformulated as follows [45]:

$$x[k+1] = A_{s_i} x[k] + B'_{0,s_i} u[k - (n_{f_i} - 1)] + B'_{1,s_i} u[k - n_{f_i}], \quad (15)$$

where A_{s_i}, B'_{0,s_i} and B'_{1,s_i} are given by replacing τ_i by τ' and h_i by h in Eq. 12. It is interesting to note that the matrices A_{s_i}, B'_{0,s_i} and B'_{1,s_i} are identical for all the scenarios due to this formulation. However, Eq. 15 is still different for different aggregated scenarios due to varying n_{f_i}. We leverage the identical matrices during the runtime implementation as we only have to store a few key matrices as explained in Section VI-D4.

Next, we define new augmented system states $z'[k] = \left[x[k] \; u[k-(n_{f_i}-1)] \cdots u[k-2] \; u[k-1]\right]^T$ to obtain a higher-order augmented system as follows:

$$z'[k+1] = A'_{s_i} z'[k] + B'_{s_i} u[k], \; y[k] = C' z'[k] + D_c u[k],$$
$$A'_{s_i} = \begin{bmatrix} A_{s_i} & B'_{1,s_i} & B'_{0,s_i} & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}, \; B'_{s_i} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ I \end{bmatrix},$$
$$C' = \begin{bmatrix} C_c \; 0 \; 0 \cdots 0 \end{bmatrix}. \quad (16)$$

Controllers are then designed for this higher-order augmented system, as explained in Sec. IV-B3. The stability criterion for the switched pipelined implementation is similar to the problem for long network delays [47], [48] and as such is not explained here.

For the pipelined implementation, we identify system scenarios using the following steps, where the first two steps are done as part of the controller design: i) classify the workload scenarios s_i with the same $n_{f_i} = \left\lceil \frac{\tau_i}{h} \right\rceil$ into a maximum of $\left\lceil \frac{\tau_{wc}}{f_h} \right\rceil$ aggregated workload scenarios; ii) for each aggregated workload scenario, design a controller with $\tau_i = (n_{f_i} - 1)h + \tau'$ and $h = h_{eff}$; iii) check for control stability for the switched system with the identified aggregated workload scenarios. If the switched system is unstable, find an aggregation of workload scenarios for which the switched system is stable through an exhaustive search. If multiple scenario sets provide a stable system, we choose the set with the highest cardinality and shortest average sensor-to-actuator delay; iv) the identified aggregated workload scenarios that result in a stable switched system are the system scenarios s_s with $\tau_s = (n_{f_s} - 1)h + \tau'$ and $h = h_{eff}$.

4) IMPLEMENTATION-AWARE CONTROL MATRICES AND CONTROL CONFIGURATIONS

When we allow switching for pipelined implementation, the challenge is the varying dimensions of matrices in Eq. 16 for varying delays due to workload variations. The varying dimensions affect the runtime computation of $u[k]$ (see Eq. 8), where the matrix K_i needs to be multiplied with $z[k]$ at every time-step. This challenge also occurs when some of the system states are estimated and not directly obtained from sensor measurements. For the LKAS, only the third state is computed from the sensor and the other states are estimated using the system model (see Eq. 7). In this case, $z[k+1]$ (see Eq. 7) needs to be computed at every time step.

We tackle the challenge of varying dimensions of matrices by unifying/normalising the dimensions of matrices

considering the worst-case delay τ_{wc} over all system scenarios s_s. The matrices for the worst-case delay scenario s_{wc} annotated with (h, τ_{wc}) are the same as in Eq. 16. Let n be the order of the square matrix $A_{s_{wc}}$ and $n_{f_{wc}} = \lceil \frac{\tau_{wc}}{h} \rceil$. For each system scenario s_s, with $\tau_s > h$, $n_{f_s} = \lceil \frac{\tau_s}{h} \rceil$, the system matrices are given below. For brevity, let $n_{f_{wc}} = n_f$ and n is the order of matrix A_{s_s}. It is interesting to note that $A_{s_s} = A_{s_{wc}}$ when the period h is constant for the system scenarios s_s. The order of the square matrix A'_{s_s} is $(n + n_f)$.

$$A'_{s_s} = \begin{bmatrix} A_{s_s} & 0_{(n_f-1)\times n} & 0_{1\times n} \\ 0_{n\times(n_f-n_{f_s})} & & \\ B'_{1,s_s} & 0_{(n_f-1)\times 1} & 0_{1\times 1} \\ B'_{0,s_s} & & \\ 0_{n\times(n_{f_s}-2)} & I_{(n_f-1)\times(n_f-1)} & 0_{1\times(n_f-1)} \end{bmatrix}^T,$$

$$B'_{s_s} = \begin{bmatrix} 0_{(n+n_f-1)\times 1} & I_{1\times 1} \end{bmatrix}^T, \; C' = \begin{bmatrix} C_c & 0_{1\times n_f} \end{bmatrix}.$$

Further, in a pipelined implementation due to workload variations we could have a scenario s_s with $\tau_s \leq h$ (see for instance iteration $[k+4]$ in Fig. 13). We derive the matrices for such scenarios as follows. Also, since $\tau_s \leq h$, $n_{f_s} = 1$.

$$A'_{s_s} = \begin{bmatrix} A_{s_s} & 0_{n\times(n_f-n_{f_s})} & B'_{1,s_s} \\ 0_{(n_f-1)\times n} & 0_{(n_f-1)\times 1} & I_{(n_f-1)\times(n_f-1)} \\ 0_{1\times n} & 0_{1\times 1} & 0_{1\times(n_f-1)} \end{bmatrix},$$

$$B'_{s_s} = \begin{bmatrix} B'_{0,s_s} & 0_{(n_f-1)\times 1} & I_{1\times 1} \end{bmatrix}^T,$$

$$C' = \begin{bmatrix} C_c & 0_{1\times n_f} \end{bmatrix}.$$

The matrices A_{s_s}, B'_{0,s_s} and B'_{1,s_s} are obtained for scenario s_s as explained in Eq. 16. We can apply standard control design techniques for these implementation-aware system models. State-feedback and feed-forward controllers can be designed as shown in Eq. 8 for the system scenarios to obtain K_s and F_s. The control configurations are then $\chi^c_{s_s} = (h_{eff}, \tau_s, K_s, F_s)$.

Note that when storing matrices for a pipelined implementation, we only need to store K_s and F_s for each scenario s_s, $A_{s_{wc}}$, B'_{0,s_s}, B'_{1,s_s}, and C_c for the constant period h and τ'. We only need to store one $A_{s_{wc}}$, B'_{0,s_s}, B'_{1,s_s}, and C_c matrix as the period and actuation rate are constant.

VII. EXPERIMENTAL RESULTS AND DISCUSSION

This section explores the SPADe experimental results with a focus on design-space exploration (DSE). The DSE is used for identifying the degrees of parallelism and pipelining for the pipelined parallelism implementation. A DSE is required since the parallelisation is limited by the degree of application parallelism quantified using $n_c^{//}$ and n_c^{avl}, and the maximum number of active pipes we can have is limited by $f_d, f_h, n_c^{//}$ and n_c^{avl}. After exploring the DSE results for the running example, we discuss a few observations for the SPADe flow and compare the state-of-the-art methods with our proposed SPADe flow for pipelined parallelism. The simulations in this section consider the LKAS case study introduced in Section III-C. The LKAS is modelled using the IBC graph given in Fig. 7 (a) with the worst-case workload scenario having $z = 6$ and the execution times of the actors as explained in Section IV-A1. The platform we consider is the predictable CompSOC platform (as explained in Section III-A) with n_c^{avl} as an input to the SPADe flow.

A. DESIGN-SPACE EXPLORATION (DSE)

The SPADe flow has been explored till now with a fixed number of pipes p and a fixed number of cores per pipe for parallelism $n_c^{//}$. A DSE with design parameters f_h, p, $n_c^{//}$, n_c^{avl} is needed to identify the Pareto-optimal implementation choice, i.e., the degree of pipelining and the degree of parallelism. Note that p quantifies the degree of pipelining, and $n_c^{//}$ quantifies the degree of application parallelism for an implementation choice. Typically, f_h and n_c^{avl} are given or fixed. We only vary these two parameters if the goal is to identify the optimal frame rate or minimise resource usage.

We use a brute force method to identify the possible implementation choices $< n_c^{avl}, \; n_c^{//}, p >$ over the design space. For each implementation choice, i.e., with fixed design parameters, we compute its GM and PM (see Section III-D) for the worst-case system scenario $s_{s_{wc}}$ (see Step 29 in Algorithm 2). GM and PM are analytical metrics and can be computed easily from the discretized control system model in Eq. 7 for $s_{s_{wc}}$. We suggest using GM and PM to prune the design space to be explored for multiple implementation choices, as the MSE and ST QoC metrics that we ultimately want to optimise are simulation-based and obtaining them is compute-intensive. A performance comparison for different implementation choices using MSE and ST is moreover unfair unless we can do exhaustive simulation considering different initial conditions and environments (e.g., weather conditions, as illustrated in our previous work [58]).

Thus, the Pareto-optimal implementation choice for a given platform allocation n_c^{avl} is chosen as the one with the highest GM and PM. If GM and PM are incomparable for multiple implementation choices for a platform allocation, in the sense that one implementation choice has a higher GM and the other one a higher PM, then we consider both these choices as part of the Pareto front. The Pareto-optimal implementation choices are further analysed using MSE and ST. To do so, controllers are designed for the Pareto-optimal implementation choices, and optimal system scenarios considering workload variations are identified based on control performance metrics MSE and ST.

As a proof-of-concept, we performed DSE with $f_h = \frac{1}{60}$ and given (maximum) platform allocation of six processing cores, i.e., $n_c^{avl} = 6$, considering the IBC graph in Fig. 7 (a). The QoC over various design points is illustrated in Fig. 15. For the purpose of validation, we explored more implementation choices through simulation than only the Pareto-optimal ones as described above. The results show that the implementation choices with higher gain and phase margins indeed have better control performance (MSE and ST) in the simulations, confirming that GM and PM can be used to prune

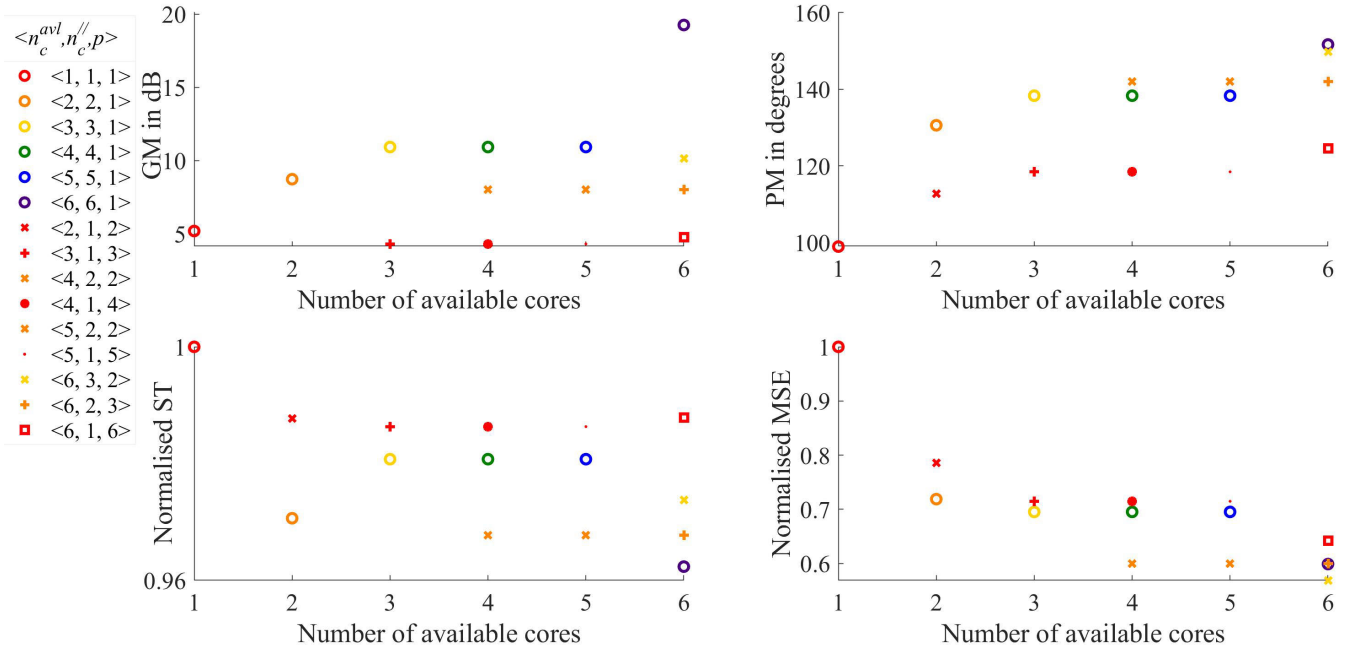

FIGURE 15. Pareto-plot for simulation - QoC vs given platform allocation, i.e., n_c^{avl}. Here, we consider the SADFG in Fig. 7 (a) with $z = 6$ (worst-case workload). The MSE and ST are normalised with respect to the maximum (worst-case) value. The legend denotes $< n_c^{avl}, n_c^{//}, p >$. Higher GM and PM, and lower MSE and ST are better.

the design space. The only anomaly is the normalised settling time (ST) for the case $< 2, 2, 1 >$ for two cores which is better than the ST for the configurations with a higher number of cores. We observe that this is due to our simulations proceeding at the rate of the sampling period. E.g., the sampling period for both the cases $< 2, 2, 1 >$ and $< 3, 3, 1 >$ is the same (0.0667 ms), but their sensor-to-actuator delays are 0.065 ms and 0.055 ms, respectively. When we proceed with the simulations at the rate of the sampling period, the slight differences in the settling time are due to approximations in the model fitting.

In terms of optimising QoC for the running LKAS example, an important first observation from the DSE results is that all configurations that exploit parallelism and/or pipelining improve QoC over the fully sequential implementation $< 1, 1, 1 >$. Our simulations moreover show that, typically, for the optimal QoC, we should parallelise as much as possible (increasing $n_c^{//}$) and then pipeline (increasing p). For example, in Fig. 15, let us consider the cases of $n_c^{avl} = 3$ and $n_c^{avl} = 6$. Among the configurations with $n_c^{avl} = 3$, we notice that the configuration $< 3, 3, 1 >$ has the highest GM and PM and the best control performance. The normalised MSE is similar for both $< 3, 3, 1 >$ and $< 3, 1, 3 >$. There is a visible improvement in the normalised ST for the first of these two configurations that maximises parallelism compared to the second that maximises pipelining. Similarly, when $n_c^{avl} = 6$, we notice that configuration $< 6, 6, 1 >$ has the highest GM and PM and best normalised settling time. The normalised MSE is similar to the cases $< 6, 3, 2 >$ and $< 6, 2, 3 >$. We observe that this is due to having the same sampling period for all these cases. The trend that parallelisation should be prioritised over pipelining is true for other platform allocations.

Let us now consider the cases of $n_c^{avl} = 4$ and $n_c^{avl} = 5$. It is interesting to notice that the control performance for both these cases is identical, and the performance does not improve by allocating one more core when $n_c^{avl} = 4$. This is due to identical control timing parameters, delay and period, when $n_c^{avl} = 4$ and $n_c^{avl} = 5$. It is also interesting to note that the fully parallelisable implementations $< 4, 4, 1 >$ and $< 5, 5, 1 >$ do not have the best performance for both these cases. This is because of identical delay (=55 ms) and period (=66.7 ms) when $n_c^{//} = n_c^{avl}$ and $p = 1$, for the cases with $n_c^{avl} = 3, 4, 5$. This happens because when we distribute six RoIs over 3, 4 or 5 cores, there are always two sequential RoI executions needed on (at least) one core, meaning that the effective delay does not change. This means that we already have a fully parallelisable implementation with $n_c^{avl} = 3$. When we have a fully parallelisable implementation and still have more cores available, we can pipeline. E.g., the cases with the best performance for $n_c^{avl} = 4, 5$ are $< 4, 2, 2 >$ and $< 5, 2, 2 >$. These cases have delay and period equal to 65 ms and 33.3 ms, respectively. We observe that considering parallelism and pipelining integrally has better performance than considering only one of the two options.

The main conclusion of the DSE is that we should *parallelise as much as possible and then pipeline while taking into account the delay and period*. It is important to note that increasing the number of available cores does not necessarily improve the control performance. This is due to several

factors. The most prominent factors, the camera frame rate and the inter-frame dependencies, have been discussed earlier. Our results also show the significance of GM and PM in pruning the design space for exploration. A higher GM and PM typically imply a better control performance (MSE and ST). The GM and PM are computed analytically, whereas the MSE and ST can only be computed through simulations. For SPADe, we use this knowledge to prune the design space.

B. IS HIGHER FRAME RATE ALWAYS BETTER?

Our observation with respect to the frame rate is that having a higher frame rate is not always better. Having a higher frame rate means processing the arriving frames at a higher rate, which is compute-intensive, and may not always improve the control performance (as shown in Fig. 16). We observe that the control performance is similar for 60 fps and 120 fps for all the cases. The control performance is the worst for 30 fps when we have only one available processing core. A slight degradation in control performance is noticed for 30 fps when considering 2-5 available processing cores. If we have six available processing cores, all the frame rates have similar performance. The control performance is dependent mainly on the τ_s and h_s of the system scenarios. If improving the frame rate does not effectively decrease the average delay and/or period, it becomes an overhead to do so and wastes compute resources in the given platform.

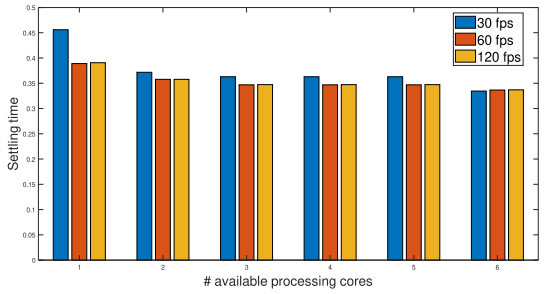

FIGURE 16. n_c^{avl} vs settling time (in s) considering different frame rates for the SADFG in Fig. 7 (a) with $z = 6$ (workload) and $p = 1$.

C. IMPACT OF INTER-FRAME DEPENDENCIES

In a pipelined implementation, the inter-frame dependencies have a significant impact on the maximum number of pipes we can realise p_s (see Step 30 of Algorithm 2). The impact of the inter-frame dependence time f_d on p_s is illustrated in Fig. 17 using an example. In this example, we allocate sufficiently many cores $n_c^{avl} = 12$ since $n_{c_{max}} = 12$ (see Step 22 in Algorithm 2) for a camera frame rate of 120 fps, considering a worst-case delay of $\tau_{wc} = 95$ ms, and $n_c^{//} = 1$.

If $f_d = 0$, the maximum number of realisable pipes we can have when the camera frame rates are 30, 60 or 120 fps are 3, 6 or 12, respectively. E.g., if the camera frame rate is 120 fps, then we can have a maximum of $\lceil \tau_{wc}/f_h \rceil = \lceil 0.095 \times 120 \rceil = 12$ pipes. Note that, as f_d increases, the number of realisable pipes reduces exponentially.

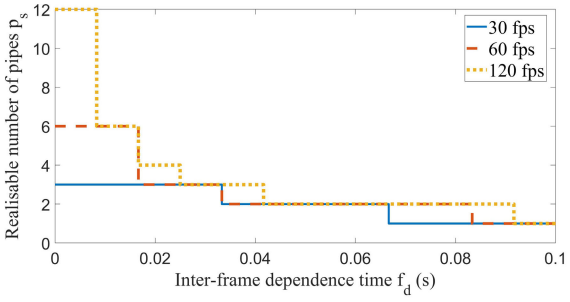

FIGURE 17. Impact of f_d on maximum number of realisable pipes p_s. In this example, we consider $\tau_{wc} = 95$ ms, $n_c^{//} = 1$ and $n_c^{avl} = 12$.

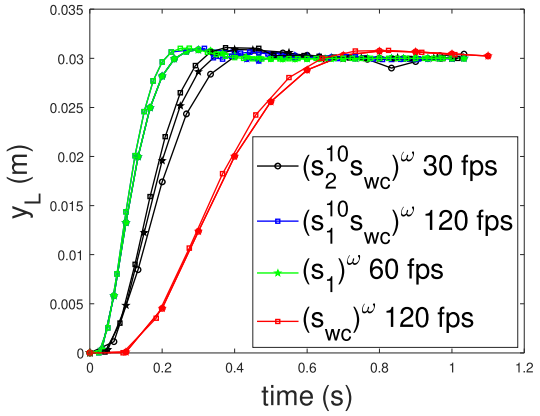

FIGURE 18. Impact of system scenarios switching due to workload variations on y_L with $n_c^{avl} = 1$. Note that the legends mention representative scenario sequences; the plot contains more scenario sequences than the ones mentioned in the legends. The markers denote the frame rate and colours denote the scenario sequences.

E.g., if $f_h < f_d \leq 2f_h$, then the number of frames to skip after processing each camera frame is one. Now for the camera frame of 120 fps, skipping one frame after every frame processing means that the realisable number of pipes will become 6 (see the interval $f_d = (0.0083, 0.01667]$ in Fig. 17). A higher f_d implies a smaller number of realisable pipes. Also, as f_d increases beyond a certain point, pipelining is no longer feasible (when $p_s = 1$). E.g., in Fig. 17, $p_s = 1$ in the intervals $f_d = (0.067, 0.100]$ for 30 fps, $f_d = (0.083, 0.100]$ for 60 fps, and $f_d = (0.092, 0.100]$ for 120 fps.

D. IMPACT OF SYSTEM-SCENARIO SWITCHING ON CONTROL PERFORMANCE

It is possible for switching to degrade the control performance compared to the periodic worst-case-based design (presented in Fig. 15). However, we choose system scenarios such that the control performance improves even if we have switching at runtime. Further, we discard the combinations of aggregated workload scenarios that degrade control performance compared to the single worst-case workload scenario s_{wc} during system-scenario identification (see Section IV-B4). We illustrate this observation using Fig. 18 where we consider the SADFG in Fig. 7 having three system scenarios s_1, s_2

and s_{wc} with 1, 3 and 6 RoI respectively. s_1 is the best-case workload scenario with the smallest delay and s_{wc} is the worst-case scenario with the worst-case delay. Notice that, in all switching cases, the control performance settles faster than the periodic worst-case based design (denoted by $(s_{wc})^\omega$ in Fig. 18). The trend is similar for different camera frame rates as well. We observe that the control performance is better if the frequently occurring workload scenario is closer to the best case than the worst case. E.g., in Fig. 18, the scenario sequence $(s_1^{10}s_{wc})^\omega$ has s_1 as the frequently occurring scenario, which leads to a better performance than $(s_2^{10}s_{wc})^\omega$ with s_2 as the most frequently occurring scenario. Further, we observe that the control performance of the scenario sequences cluster towards the performance of its most frequently occurring scenario, as is observed in Fig. 18 with distinct colour regions (where the blue and green scenarios form one cluster).

E. COMPARISON WITH THE STATE-OF-THE-ART

The DSE, as explained in Section VII-A and illustrated in Fig. 15, already shows quantitatively that integrally considering the combination of parallelism and pipelining in multiprocessor IBC design outperforms the state-of-the-art approaches in which only one of the two options is considered. The integral consideration of parallelism and pipelining differentiates SPADe from any earlier work, as explained in Section II. In this subsection, we compare the proposed SPADe method qualitatively on several other relevant criteria with state-of-the-art multiprocessor IBC design techniques in Table 1. For brevity, we only compare with multiprocessor IBC system implementations and not with traditional sequential control design techniques based on the worst-case sensing delay, as it has already been shown in [2], [50] that multiprocessor implementations are beneficial for optimising control performance. The multiprocessor implementations can be classified into pipelined [8], [9] with constant delay, pipelined with variable delay [49] and sequential implementation with parallelisable sensing [2], [7]. The camera frame rate, however, is not explicitly considered in [8]. A brief comparison between [2] and [9] is already reported in [7].

The proposed approach is advantageous to other multiprocessor IBC system implementations with respect to: 1) coverage of the design space considering both pipelining and parallelisation; 2) considering inter-frame dependencies and resource sharing among pipes; and 3) not imposing any restrictions on τ, thereby enabling shorter τ and h compared to other approaches.

VIII. SPADe ADAPTATION FOR AN INDUSTRIAL PLATFORM

We have explained the SPADe flow until now assuming that the timing of the implementation is predictable. However, for an industrial platform, it is difficult to predict the timing analytically (as users have restricted freedom in scheduling due to caching, resource sharing, etc.), and any computed timing is pessimistic. This section shows how we can apply the SPADe flow even when it is difficult to give tight predictable timing guarantees. The idea is that we use the frequently occurring task execution times instead of the pessimistic worst-case execution time (WCET) estimates to obtain temporal bounds.

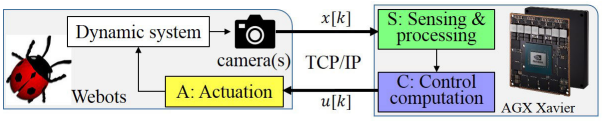

FIGURE 19. HiL setting overview with NVIDIA AGX Xavier.

Our previous work [7] explored this idea for a non-pipelined parallelisable implementation. In this work, we show that the SPADe adaptation for an industrial platform is relevant for pipelined parallelism as well.

The assumption for implementation is that it is possible to time-trigger tasks either through polling or through interrupt timers in the industrial platform. We provide validation through a HiL simulation, showing that we can guarantee control stability for the SPADe design outcomes, despite the fact that the execution-time estimates for tasks used in the SPADe flow are not conservative.

A. HiL SETTING FOR OUR CASE STUDY

Fig. 19 illustrates our HiL validation setup for LKAS adapted from [51]. It simulates a vehicle with a top look-ahead camera using the Webots [52] physics simulator engine and interacts with an NVIDIA AGX Xavier platform using the TCP/IP protocol. The simulator works in a server-client configuration. Webots acts as the server while the NVIDIA platform acts as the client. The server (Webots) progresses simulation in full synchronisation with the client (NVIDIA AGX Xavier) [37]. At each simulation step, the camera sensor simulated in Webots generates a raw image containing state information $x[k]$, that is fed to the NVIDIA platform. It executes the sensing (S) and control (C) tasks to generate control input $u[k]$, which is communicated back to Webots for actuation. After actuation, the simulation progresses to the next step.

For our evaluation, the camera sensor in Webots is modelled based on the AR1335 CMOS digital image sensor [53] and is set to a resolution of 720p.[3] The camera frame rate is varied between 30 fps, 60 fps, and 120 fps, depending on the sampling period of the controller. The actuation dynamics are modelled based on [55]. A lane width of 3.25 m is considered, as per standard road-safety guidelines. The vehicle is initially positioned with a fixed bias of 15 cm from the lane centre to test the control performance. The Webots simulation step is set to 1 ms, while the vehicle speed is set to 50 km/hr.

[3]State-of-the-art lane detection algorithms [54] operate on low-resolution images. So, we perform our evaluation using downscaled (512×256) sensor images. Our approach is also effective for high-res images.

TABLE 1. Comparing the proposed SPADe approach with the state-of-the-art multiprocessor IBC system implementations.

Criteria	Proposed SPADe	Pipelined		Non-pipelined [2], [7]
		constant delay [8], [9]	variable delay [49]	
Inter-frame dependencies	explicitly considered	not considered	not considered	independent
Runtime overhead due to reconfiguration	explicitly considered as a time cost in the SADFG model	not considered	not explicitly considered	explicitly considered
Algorithm	white/gray/black box	white/gray/black box	white/gray/black box	white/gray box
Parallelisation potential	explicitly taken into account	independent	independent	should be high
Workload variations	explicitly considered in design	not considered	indirectly considered	explicitly considered
Platform	can be adapted for all	suitable for homogeneous	can be adapted for all	directly applicable for all
Resource sharing between pipes	explicitly considered during mapping	not considered	not considered	not applicable
Restrictions on sampling period h[1]	strictly periodic for pipelined, $h < \tau_{wc}$; switched for non-pipelined;	strictly periodic; $h < \tau$	strictly periodic; $h < \tau_{wc}$	switching possible
Restrictions on sensor-to-actuator delay τ	none	strictly $\tau > h$; in [9], τ is strictly a multiple of h	strictly[2] $\tau_{wc} > h$	$\tau_{wc} \leq h$

τ_{wc}: worst-case delay; [1] If camera frame arrival period f_h is considered, always h is a multiple of f_h; [2] if $\tau_{wc} \leq h$ design reverts to non-pipelined;

B. PLATFORM GRAPH

We proceed to explain the abstraction we make for the platform graph in Fig. 6. The NVIDIA AGX Xavier platform has a Carmel CPU complex with eight cores and a Volta GPU with 512 CUDA cores and 64 Tensor cores, as shown in Fig. 4. Modelling all the GPU cores separately may lead to state-space explosion and is inefficient for the dataflow timing and mapping analysis. Also, the proprietary GPU scheduler is closed-source and needs to be accessed through the CPU. The execution times are difficult to predict for the tasks mapped to the GPU when there are other shared tasks on the GPU. Therefore, we abstract and combine the execution times of the tasks mapped to the GPU along with the execution time of the CPU task that accesses it. Thus, the platform allocation n_c^{avl} is defined based on the number of CPU cores allocated. For the NVIDIA AGX Xavier platform, we have a maximum of eight CPU cores, i.e., $n_c^{avl} = 8$.

C. IBC GRAPH

Next, we explain the IBC graph of Fig. 6 used in this experiment and how we populate it with the profiling information. We model the IBC sensing algorithm using the IBC graph illustrated in Fig. 20. This graph structure is for the approximate setting 'S3' of the IBC system implemented in [56], [57]. The execution time numbers for the actors in the graph and the rates in the channels are obtained by mapping and profiling the IBC application on the NVIDIA AGX Xavier platform. We perform a model-fitting using the profiled timing information to update the execution time of the actors for each workload scenario. The resulting IBC SADFG is then one of the inputs to the SPADe flow explained in Algorithm 2.

For profiling, around 100 images are identified with varying image workloads. We execute each stage in the LKAS 100 times for every image in the dataset to reduce sensitivity to access locality. This helps to characterise the profiling information as a PERT distribution [3] for the latency of an iteration of the graph. Workload scenarios are classified based on the resulting PERT distribution by identifying regions in the distribution based on occurrence frequency. This results in scenario graphs with the same graph structure and channel rates, but different actor execution times. The workload scenarios for this setup are not based on RoI, as in our running example. For the worst-case scenario, we use the worst-case profiling numbers for the execution times of the actors. Other workload scenarios use the best-case, first quartile, median and third quartile profiling data. As an example, the model parameters for the workload scenario using third quartile profiling data are $e_{dm} = 10.3$ ms, $e_{dn} = 4$ ms, $e_1 = 0.3$ ms, $e_2 = 4.65$ ms, $e_d = 5$ ms, $e_p = 1.4$ ms, $e_m = 0.16$ ms (see Fig. 20), $e_C = 0.016$ ms, and $e_A = 0.5$ ms (assuming single actors C and A for the control compute and actuation tasks). Workload scenarios may also be classified based on other parameters, like RoI in the running example. Our previous work [7] details the case where the workload scenarios are classified based on different operating modes of an

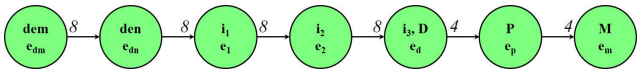

FIGURE 20. LKAS IBC graph of the sensing algorithm implementation derived from [57]. The actors are: dem - demosaicing; den - denoising; $i_{1,2,3}$ - abstract colour mapping and white balancing, tone mapping, and compression; and D-P-M model the lane detection, processing and merging tasks. The output is the lateral deviation y_L.

application due to environmental conditions. Here, we use the observed overall latency of the sensing, because it is difficult to predict execution times on the considered platform based on other parameters.

D. SPADe FLOW, DSE AND HiL VALIDATION

Once we model the IBC SADFG with the profiling information for every workload scenario, the rest of the steps in the SPADe flow follows Algorithm 2. A DSE is performed for different implementation choices $n_c^{//}$ and p (as explained in Section VII-A). A DSE involves analytical computation of GM and PM (in Matlab) for the different implementation choices (results are shown in Fig. 21). We consider three given platform allocations - single-core, four cores and eight cores, i.e., $n_c^{avl} = 1, 4, 8$. The implementation choices with the highest GM and PM are Pareto-optimal.

The results in Fig. 21 show that parallelising as much as possible gives the best result, i.e. the configurations $< 4, 4, 1 >$ and $< 8, 8, 1 >$ have the best GM and PM. The configurations $< 4, 4, 2 >$ and $< 8, 4, 2 >$ have similar GM and PM and can be considered for further analysis. The results confirm the earlier conclusion of Section VII-A that parallelisation should be prioritised over pipelining. In this case, pipelining does not have any added value ($p = 1$ in both Pareto-optimal configurations), because the LKAS implementation is highly parallelisable. For applications with a lower degree of parallelism, e.g., when maximum parallelism is achieved with 4 cores and $n_c^{avl} = 8$, this would likely not be the case. In such a case, we can use the additional four cores for an additional pipe, implying that configuration $< 8, 4, 2 >$ would have been ideal.

Next, we validate the Pareto-optimal implementation choices in the HiL setting of Section VIII-A for control stability, considering both parallelism and pipelining together. Recall that the SPADe flow has a built-in stability check for the system configurations considering the initial system model. However, the actual LKAS implementation has to cater to different environment conditions, noise levels, and model uncertainties. HiL validation is often used to simulate the designed system configurations under real-life environment conditions. Therefore, we also perform a HiL validation experiment. From the experiment, we observe that stability of the implementation choices of the closed-loop system is confirmed. We can moreover verify that the order of the control system performance obtained in HiL conforms to the GM and PM predictions. As explained, we rely on analytical metrics GM and PM to select our design points. The MSE and ST comparisons are omitted as performing an exhaustive, fair simulation for even a single design point is too compute intensive. But the DSE performed with the SPADe flow and the HiL validation of the results show that SPADe can be adopted for industrial platforms.

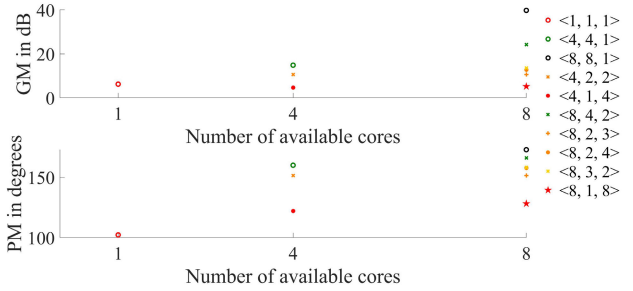

FIGURE 21. Pareto-plot for GM and PM vs n_c^{avl}. The legend denotes $< n_c^{avl}, n_c^{//}, p >$.

IX. CONCLUSION

We have presented a scenario- and platform-aware design flow (SPADe) for IBC system implementation that considers both pipelining and parallelism in an integral fashion to improve QoC of multiprocessor IBC implementations. We propose model transformations for modelling, analyzing and mapping the IBC system. We explain how the SPADe approach can explicitly take into account inter-frame dependencies in pipelining, image workload variations, application parallelism, resource sharing, camera frame rate and a given platform allocation. We validate the SPADe approach using Matlab simulations considering the predictable CompSOC platform and using hardware-in-the-loop experiments with an NVIDIA AGX Xavier platform. We observe that considering pipelined parallelism has inherent advantages over considering pipelining and parallelism separately. Exploiting parallelism should be prioritized over pipelining. But pipelined parallelism is better when an application has a limited degree of parallelism or when the inter-frame dependencies are significant. Future work may involve exploring the optimal degree of pipelining and parallelism when multiple IBC systems are sharing a platform.

REFERENCES

[1] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, vol. 118, 2nd ed. Springer, 2017.
[2] S. Mohamed, A. U. Awan, D. Goswami, and T. Basten, "Designing image-based control systems considering workload variations," in *Proc. IEEE 58th Conf. Decis. Control (CDC)*, Dec. 2019, pp. 3997–4004.
[3] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, and R. Schiffelers, "Robustness analysis of multiprocessor schedules," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation (SAMOS)*, Jul. 2014, pp. 9–17.
[4] P. M. Sharkey and D. W. Murray, "Delays versus performance of visually guided systems," *IEE Proc.-Control Theory Appl.*, vol. 143, no. 5, pp. 436–447, Sep. 1996.
[5] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*. Chelmsford, MA, USA: Courier Corporation, 2013.
[6] S. Mohamed, D. Zhu, D. Goswami, and T. Basten, "Optimising quality-of-control for data-intensive multiprocessor image-based control systems considering workload variations," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 320–327.

[7] S. Mohamed, D. Goswami, V. Nathan, R. Rajappa, and T. Basten, "A scenario- and platform-aware design flow for image-based control systems," *Microprocessors Microsyst.*, vol. 75, Jun. 2020, Art. no. 103037.
[8] P. Krautgartner and M. Vincze, "Performance evaluation of vision-based control tasks," in *Proc. IEEE Int. Conf. Robot. Automat.*, vol. 3, May 1998, pp. 2315–2320.
[9] R. Medina, J. Valencia, S. Stuijk, D. Goswami, and T. Basten, "Designing a controller with image-based pipelined sensing and additive uncertainties," *ACM Trans. Cyber-Phys. Syst.*, vol. 3, no. 3, pp. 1–26, Oct. 2019.
[10] S. Li, C. Zhu, Y. Gao, Y. Zhou, F. Dufaux, and M.-T. Sun, "Lagrangian multiplier adaptation for rate-distortion optimization with inter-frame dependency," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 26, no. 1, pp. 117–129, Jan. 2016.
[11] A. W. M. Smeulders, D. M. Chu, R. Cucchiara, S. Calderara, A. Dehghan, and M. Shah, "Visual tracking: An experimental survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 7, pp. 1442–1468, Jul. 2014.
[12] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[13] B. D. Theelen, M. C. Geilen, T. Basten, J. P. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Proc. Formal Methods Models Co-Design (MEMOCODE)*, 2006, pp. 185–194.
[14] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF for free," in *Proc. 6th Int. Conf. Appl. Concurrency Syst. Design (ACSD)*, 2006, pp. 276–278.
[15] S. Anssi, K. Albers, M. Dörfel, and S. Gérard, "chronVAL/chronSIM: A tool suite for timing verification of automotive applications," in *Proc. Embedded Real Time Softw. Syst. (ERTS)*, 2012, pp. 1–11.
[16] M. Lattuada and F. Ferrandi, "Modeling pipelined application with synchronous data flow graphs," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation (SAMOS)*, Jul. 2013, pp. 49–55.
[17] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Design Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–24, Jan. 2009.
[18] K.-E. Arzén, A. Cervin, J. Eker, and L. Sha, "An introduction to control and scheduling co-design," in *Proc. 39th IEEE Conf. Decis. Control (CDC)*, vol. 5, Dec. 2000, pp. 4865–4870.
[19] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf, "Future automotive systems design: Research challenges and opportunities: Special session," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2018, pp. 1–7.
[20] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen, "How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime," *IEEE Control Syst. Mag.*, vol. 23, no. 3, pp. 16–30, Jun. 2003.
[21] D. Goswami, A. Masrur, R. Schneider, C. J. Xue, and S. Chakraborty, "Multirate controller design for resource- and schedule-constrained automotive ECUs," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1123–1126.
[22] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, Jan. 2012.
[23] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. D. Bosschere, "System-scenario-based design of dynamic embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–45, Jan. 2009.
[24] Y. Wang, B. M. Nguyen, H. Fujimoto, and Y. Hori, "Multirate estimation and control of body slip angle for electric vehicles based on onboard vision system," *IEEE Trans. Ind. Electron.*, vol. 61, no. 2, pp. 1133–1143, Feb. 2014.
[25] A. Kawamura, K. Tahara, R. Kurazume, and T. Hasegawa, "Robust visual servoing for object manipulation with large time-delays of visual information," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2012, pp. 4797–4803.
[26] G. Macesanu, V. Comnac, F. Moldoveanu, and S. M. Grigorescu, "A time-delay control approach for a stereo vision based human-machine interaction system," *J. Intell. Robot. Syst.*, vol. 76, no. 2, pp. 297–313, Nov. 2014.
[27] E. van Horssen, "Data-intensive feedback control: Switched systems analysis and design," Ph.D. dissertation, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2018.
[28] H. Fujimoto, "Visual servoing of 6 DOF manipulator by multirate control with depth identification," in *Proc. 42nd IEEE Conf. Decis. Control (CDC)*, vol. 5, Dec. 2003, pp. 5408–5413.
[29] R. Agrawal, S. Gupta, J. Mukherjee, and R. K. Layek, "A GPU based real-time CUDA implementation for obtaining visual saliency," in *Proc. Indian Conf. Comput. Vis. Graph. Image Process.*, Dec. 2014, pp. 1–8.
[30] S. Kestur, M. S. Park, J. Sabarad, D. Dantara, V. Narayanan, Y. Chen, and D. Khosla, "Emulating mammalian vision on reconfigurable hardware," in *Proc. IEEE 20th Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2012, pp. 141–148.
[31] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[32] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, vol. 4, May 2000, pp. 101–104.
[33] S. Stuijk, M. Geilen, and T. Basten, "A predictable multiprocessor design flow for streaming applications with dynamic behaviour," in *Proc. Euromicro Conf. Digit. Syst. Design (DSD)*, 2010, pp. 548–555.
[34] S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proc. Design, Automat. Test Eur. Conf. (DATE)*, 2003, pp. 1–6.
[35] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, Feb. 2007.
[36] S. Stuijk, "Predictable mapping of streaming applications on multiprocessors," Ph.D. dissertation, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2007.
[37] D. Franklin. (2018). NVIDIA jetson AGX xavier delivers 32 TeraOps for new era of AI in robotics. NVIDIA Developer Blog. [Online]. Available: https://devblogs.nvidia.com/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/
[38] J. Kosecka, R. Blasi, C. J. Taylor, and J. Malik, "Vision-based lateral control of vehicles," in *Proc. Conf. Intell. Transp. Syst.*, 1997, pp. 900–905.
[39] Z. Sun and S. S. Ge, *Stability Theory of Switched Dynamical Systems*. Springer, 2011.
[40] H. A. Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten, "Scalable analysis for multi-scale dataflow models," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 4, pp. 1–26, Aug. 2018.
[41] C. Leake, "Synchronization and linearity: An algebra for discrete event systems," *J. Oper. Res. Soc.*, vol. 45, no. 1, pp. 118–119, Jan. 1994.
[42] F. Siyoum, M. Geilen, and H. Corporaal, "Symbolic analysis of dataflow applications mapped onto shared heterogeneous resources," in *Proc. 51st Annu. Design Automat. Conf. Design Automat. Conf. (DAC)*, 2014, pp. 1–6.
[43] R. C. Dorf and R. H. Bishop, *Modern Control Systems*. London, U.K.: Pearson, 2011.
[44] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Boca Raton, FL, USA: CRC Press, 2018.
[45] K. Ogata, *Discrete-Time Control Systems*, vol. 2. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995.
[46] S. Mohamed, N. Saraf, D. Bernardini, D. Goswami, T. Basten, and A. Bemporad, "Adaptive predictive control for pipelined multiprocessor image-based control systems considering workload variations," in *Proc. 59th IEEE Conf. Decis. Control (CDC)*, Dec. 2020, pp. 5236–5242.
[47] B. Lincoln and B. Bernhardsson, "Optimal control over networks with long random delays," in *Proc. Int. Symp. Math. Theory Netw. Syst.*, vol. 7, 2000, pp. 1–14.
[48] M. B. G. Cloosterman, N. van de Wouw, W. P. M. H. Heemels, and H. Nijmeijer, "Stability of networked control systems with uncertain time-varying delays," *IEEE Trans. Autom. Control*, vol. 54, no. 7, pp. 1575–1580, Jul. 2009.
[49] R. Medina, S. Stuijk, D. Goswami, and T. Basten, "Implementation-aware design of image-based control with on-line measurable variable-delay," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 240–245.
[50] D. Fontantelli, L. Palopoli, and L. Greco, "Optimal CPU allocation to a set of control tasks with soft real-time execution constraints," in *Proc. 16th Int. Conf. Hybrid Syst., Comput. Control (HSCC)*, 2013, pp. 233–242.
[51] S. Mohamed, S. De, K. Bimpisidis, V. Nathan, D. Goswami, H. Corporaal, and T. Basten, "IMACS: A framework for performance evaluation of image approximation in a closed-loop system," in *Proc. 8th Medit. Conf. Embedded Comput. (MECO)*, Jun. 2019, pp. 1–4.
[52] O. Michel, "Cyberbotics Ltd. Webots: Professional mobile robot simulation," *Int. J. Adv. Robot. Syst.*, vol. 1, no. 1, p. 5, 2004, doi: 10.5772/5618.
[53] *AR1335 CMOS Digital Image Sensor for Automotive Applications*. Accessed: Aug. 7, 2021. [Online]. Available: https://www.onsemi.com/products/sensors/image-sensors/ar1335

[54] *NVIDIA LaneNet, High-Precision Lane Detection*. Accessed: Aug. 7, 2021. [Online]. Available: https://blogs.nvidia.com/blog/2019/07/10/drive-labs-neural-nets-predict-lane-lines/
[55] R. Frank, "Steering in the right direction," Electron. Des., 2016. Accessed: Aug. 7, 2021. [Online]. Available: https://www.electronicdesign.com/markets/automotive/article/21801204/steering-in-the-right-direction
[56] S. De, S. Mohamed, D. Goswami, and H. Corporaal, "Approximation-aware design of an image-based control system," *IEEE Access*, vol. 8, pp. 174568–174586, 2020.
[57] S. De, S. Mohamed, K. Bimpisidis, D. Goswami, T. Basten, and H. Corporaal, "Approximation trade offs in an image-based control system," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1680–1685.

SAJID MOHAMED (Member, IEEE) received the B.Tech. degree in electrical and electronics engineering from the National Institute of Technology (NIT) Calicut, India, and the M.Tech. degree in embedded controls and software from Indian Institute of Technology (IIT) Kharagpur. He is currently a Researcher with the Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology (TU/e). His research interests include model-based design of embedded and cyber-physical systems.

DIP GOSWAMI (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the National University of Singapore (NUS), in 2009. From 2010 to 2012, he was an Alexander von Humboldt Postdoctoral Fellow at TU Munich, Germany. He is currently an Assistant Professor with the Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology (TU/e). His research interests include various design aspects of embedded control systems in resource-constrained domains, such as automotive and robotics.

SAYANDIP DE (Graduate Student Member, IEEE) received the B.Tech. degree in electronics and communication engineering from the Kalyani Government Engineering College (KGEC), India, and the M.Tech. degree in VLSI design from Indian Institute of Engineering Science and Technology (IIEST) Shibpur, India. He is currently pursuing the Ph.D. degree with the Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology (TU/e). His research interests include approximate computing, and design automation for low power circuits and systems.

TWAN BASTEN (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computing science from Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands. He is currently a Professor with the Department of Electrical Engineering, TU/e. He is also a Senior Research Fellow with ESI, TNO, Eindhoven. His current research interests include design of embedded and cyber-physical systems, dependable computing, and computational models.