Control/Architecture Codesign for Cyber-Physical Systems

37

Wanli Chang, Licong Zhang, Debayan Roy, and Samarjit Chakraborty

Abstract

Control/architecture codesign has recently emerged as one popular research focus in the context of cyber-physical systems. Many of the cyber-physical systems pertaining to industrial applications are embedded control systems. With the increasing size and complexity of such systems, the resource awareness in the system design is becoming an important issue. Control/architecture codesign methods integrate the design of controllers and the design of embedded platforms to exploit the characteristics on both sides. This reduces the design conservativeness of the separate design paradigm while guaranteeing the correctness of the system and thus helps to achieve more efficient design. In this chapter of the handbook, we provide an overview on the control/architecture codesign in terms of resource awareness and show three illustrative examples of state-of-the-art approaches, targeting respectively at communication-aware, memory-aware, and computation-aware design.

Acronyms

CFG	Control-Flow Graph
CPS	Cyber-Physical System
DSE	Design Space Exploration
ECU	Electronic Control Unit
E/E	Electric and Electronic
EMB	Electro-Mechanical Brake
ET	Event-Triggered
FTDMA	Flexible Time Division Multiple Access

W. Chang (✉)
Singapore Institute of Technology, Singapore, Singapore
e-mail: wanli.chang@singaporetech.edu.sg

L. Zhang • D. Roy • S. Chakraborty
TU Munich, Munich, Germany
e-mail: licong.zhang@tum.de; debayan.roy@tum.de; samarjit@tum.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_37

LCS	Live Cache States
MILP	Mixed Integer Linear Programming
OS	Operating System
PSO	Particle Swarm Optimization
RCS	Reaching Cache States
RTOS	Real-Time Operating System
TDMA	Time-Division Multiple Access
TT	Time-Triggered
WCET	Worst-Case Execution Time

Contents

37.1 Introduction

Cyber-physical systems refer to systems where tight interaction between the computational elements (cyber) and the physical entities (physical) is emphasized. A typical example of a cyber-physical system is an embedded control system. In such a system, software implementation of the controllers running on processing units are used to control physical plants. As shown in Fig. 37.1, the processing units are connected with sensors and actuators where the sensors measure the states of the plants, the controllers compute the control input, and the actuators apply the control input onto the physical plants. Today, cyber-physical systems have become commonplace and can be found in the domains like automotive, avionics, industrial automation, chemical engineering, etc. The automotive Electric and Electronic (E/E) system is an example of such a system. In a modern vehicle, increasingly more functions are realized by software mapped on the Electronic Control Unit (ECU). These include

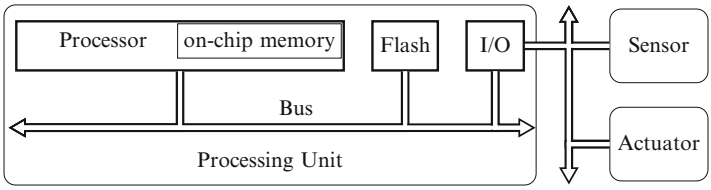

Fig. 37.1 A processing unit with a processor and on-chip memory for program execution. Instructions are stored in the flash memory. Programmable I/O peripherals are used for communication with sensors, actuators, and other processing units. For instance, Infineon XC23xxB Series, which is widely used in automotive systems, has a single processor with a minimum operating frequency of 20MHz. It is typically equipped with a small size of on-chip SRAM memory and up to 256 kB flash memory [7].

the functions for vehicle dynamics control, body components control (e.g., doors and lights), infotainment, and advanced driver assistance systems (ADAS). Some of these functions have stringent timing requirements, and some demand processing and transport of intensive data amount. The characteristics and performance of the cyber part, i.e., the electronics and software, strongly influence the performance of the physical part. In the case of safety-critical control functions, the timing properties of the software implementation of the controllers, e.g., the sampling period and the sensor-to-actuator delay, play a vital role in the control performance. Therefore, with the Cyber-Physical System (CPS)-oriented thinking, more attention is necessary for the implementation of the controllers in an embedded platform and interplay between the embedded platform design and the control design.

The hardware architecture of the computational part of a cyber-physical system consists mainly of one or more processing units. In case of a multiprocessor architecture, the processing units are commonly connected by a communication network, where data between different processing units can be transmitted. Typical communication networks in this context include the FlexRay [3], CAN [13], LIN [4], and MOST [5] in the automotive domain; AFDX ţteAFDX and AS6802 [6] in the avionics domain; and Profibus, Profinet [33], and EtherCAT in the industrial automation domain. ▸ Chapter. 24, "Networked Real-Time Embedded Systems" provides a more detailed study on some important real-time communication protocols. These communication protocols implement different data transmission approaches and are each suitable for a specific set of requirements. On each processing unit, the computation is performed by tasks, each of which is typically implemented by a piece of code. Multiple tasks can be grouped together to form an application, where an independent function (e.g., a feedback control loop) is performed. In a distributed application, where the tasks are mapped onto different processing units, the data between the relevant tasks are transmitted over the communication network. It is common that multiple tasks belonging to the same or different applications are mapped on one processing unit. In this case, an operating system (e.g., OSEK [1], eCos [18]) is sometimes used to coordinate task executions and allocate resources for the tasks.

In many embedded systems in the context of cyber-physical systems, the applications are control applications, where the software implementation of the controllers controls physical plants [28]. The design of controllers for these applications from a control-theoretical perspective are well established. The control design methods can be drawn from a large pool of research and practical expertise and experience that have been accumulated in the control community in the past few decades. However, little attention has been paid to the actual implementation of the controllers in the embedded platforms. In this case, not only the control theoretical aspect of the design problem needs to be taken into account, e.g., type of controllers and control gains, but also the characteristics of the underlying embedded platforms. The design aspects on the embedded system side include, for example, the task partitioning and mapping, the scheduling of tasks and communication, and the allocation of memory and cache. There is a tight interconnection between the control and the embedded platform design [16]. For example, the results of the embedded platform design can strongly influence the control performance through properties like sampling period, delay, and jitter. Reversely, the requirements from the control design side also influence the platform design. Conventionally the control and embedded platform design are done separately and then integrated afterward. In this case, the engineers on both sides need to make assumptions of the other side. Since most control applications are safety critical, such assumptions are inevitably quite conservative to guarantee the safety of the control applications. Due to this conservativeness, usually the resources on the embedded platform, e.g., computation, communication, memory, and energy resources, are not optimally utilized. On the other hand, these resources on an embedded platform are quite limited, constrained by the size and cost reasons. In recent years, both the size and complexity of the embedded systems in industrial domains have increased drastically. In the automotive domain, for example, a modern premium passenger car can contain up to 100 million lines of software code [17]. In such a computation and data-intensive platform, resource-efficient design has become a quite important issue. Therefore, the CPS community has become increasingly conscious that some systematic design methods will be necessary for design of resource-aware embedded control systems.

The resources on an embedded platform can be divided into different categories, e.g., computation, communication, memory, energy, and input/output interfaces. In the context of this chapter, three of the most important resources, namely, the computation resource, the communication resource, and the memory resource, are considered. In the following paragraphs, each of the aforementioned resources will be explained in detail.

Communication resources can generally be represented as the bandwidth of a communication bus or a network link, which denotes the number of bits that can be transmitted per second. Therefore, there is only a limited amount of data that can be transmitted within a specific time frame. More precise characterization of the communication resource, however, is protocol specific. The communication protocols implement different data transmission approaches, which can be broadly divided into two different categories, namely, the Time-Triggered (TT) paradigm and the Event-Triggered (ET) paradigm. For example, a Time-Division Multiple Access

(TDMA) bus is a typical time-triggered bus communication. In this case, a period of time is divided into multiple time slots, and the usage of the communication resource can be represented directly by the number of utilized slots. In an industrial-sized distributed embedded system, the communication resource is quite constrained. As the size of the system increases, more processing units and data can be incrementally mapped on to the communication bus or network. However, the bandwidth of a communication protocol cannot be easily increased. Therefore, communication-efficient design could enable the system to accommodate more applications or enhance the performance of the applications. Related to this, in recent years, there have been several works on integrated controller synthesis and task and message scheduling of distributed embedded control systems, e.g., [20, 23, 34, 35]. However, most of these works, e.g., [20, 23, 34] only consider optimization of control performance while satisfying communication constraints. In addition, there have been several works, e.g., [29, 39] on schedule optimization of distributed time-triggered embedded systems where the objective is to minimize communication bandwidth utilization while satisfying timing constraints. However, these works do not consider control applications.

Memory resources mainly refer to the size of cache due to its high cost. Within a processing unit, there are typically two levels of memory – cache and main memory. In Fig. 37.1, the on-chip memory works as cache and the flash memory serves as the main memory. The main memory has a large size and can thus store all the application programs and data, but experiences high read/write latencies (hundreds of processor cycles). The cache is faster (several processor cycles), but usually limited in size. In this chapter, the focus is on instruction memory. It is assumed that the access times of cache and main memory are t_c and t_m, respectively, where $t_c \ll t_m$. When a processor executes an instruction, it checks the cache first. If this instruction is located in the cache, it is a cache hit and the access time is t_c. If this instruction is not in the cache, the memory block containing it is fetched from the main memory and then written into cache. This is called a cache miss and the access time is t_m. Afterward, when the same instruction is called again by the processor, the access time is t_c if it is still in the cache without being replaced. Increasing the cache size and improving the cache reuse are two general methods to reduce the execution time of a program. A program usually has different execution paths resulting in different execution times, depending on the input. The Worst-Case Execution Time (WCET) is defined to be the maximum length of time a program takes to be executed. The WCET constrains the sampling period of a control application, which is defined to be the duration between two consecutive executions of a control program, and thus has significant impact on the control performance. In resource-aware embedded control systems design, it is desirable to minimize the cache size while satisfying the performance requirement or, equivalently, improve the performance for a given memory. Therefore, on one hand, the cache reuse should be maximized, and on the other hand, the controller must be suitably designed to exploit the shortened sampling periods. There have been some works on cache reuse maximization by employing code positioning during compile time [22, 26, 32] and also during run time [11], but these cannot directly be applied to embedded

control systems as code rearrangement would impact the timing properties, and this is difficult to incorporate while designing the controllers.

Computation resources usually mean the available execution time of a processor, when the processing speed is given. Considering multiple applications sharing one single-core processing unit, each application is allocated a certain period of execution time. In general, the performance of an application can be improved if it is allowed to access the processor longer. On a processor sometimes runs an Operating System (OS). For instance, ERCOSek [1, 19] is a widely used time-triggered OS on ECUs and only offers a limited set of predefined periods. It implies that the sampling periods of control applications have to be taken from this set. Generally, a shorter sampling period allows the controller to respond to its plant more frequently and is thus potentially able to achieve better control performance with an appropriately designed controller. The obvious downside is a higher processor utilization, which is defined to be the WCET of an application divided by its sampling period. This prevents more functions and applications from being integrated onto the processing unit. Therefore, the controller should use the largest possible sampling period that is able to fulfill the control performance requirement and satisfy the system constraints. In most cases, the optimal sampling period is not directly realizable on the OS. The conventional way to handle it is to use the largest sampling period offered by the OS that is smaller than the optimal one. This is a straightforward method, but leads to a waste of computational resources. Toward this, there have been several works on state-feedback-based optimal resource allocation to the control loops sharing the same processor, e.g., [14, 15, 21, 25, 30, 31]. All these works focus on online assignment of sampling periods of the control loops based on the system dynamics like plant states, disturbance, or error. However, an online decision-making must be very fast to be effective, and therefore, there must be some heuristics involved. Therefore, an offline schedule computation that guarantees performance and reduces the processor utilization will be more desirable.

The rest of this chapter is organized as follows. In Sect. 37.2, the basics of feedback control applications are briefly reviewed. In the three sections that follow, three state-of-art approaches of different aspects in terms of resource-aware algorithm/architecture codesign are explained, namely, the communication-aware design (Sect. 37.3), the memory-aware design (Sect. 37.4), and the computation-aware design (Sect. 37.5). Finally, Sect. 37.6 contains the concluding remarks.

37.2 Embedded Control Systems

In this section, some background knowledge for the embedded control systems considered in this chapter is provided. Firstly, a brief introduction in the embedded systems architecture is provided. Then the basics of feedback control systems as well as the control performance metrics and the method for optimal pole placement are explained.

37.2.1 Embedded Systems Architecture

The architecture considered in this chapter does not refer to the processor architecture, but the design parameters for the underlying hardware and the communication for the embedded controllers. The architecture can either be a single ECU, as shown in Fig. 37.1, or a distributed system consisting of multiple ECUs connected by a communication network, as shown in Fig. 37.2. An embedded controller mapped on such an architecture is usually implemented with one or multiple tasks, where each task is a piece of software code running on the processor. A controller can be partitioned into the sensor task, the controller task, and the actuator task. The sensor task measures the state of the physical plant, the controller task computes the control input, and the actuator task applies the control input onto the physical plant. In a single processor architecture, these tasks are executed on the same ECU, while in a distributed architecture, the sensor, controller, and actuator tasks can also be mapped on different ECUs and the data between the tasks are transferred over the network as messages. It is also common that tasks of different controllers are mapped on common ECUs, where the communication, computation, and memory resources are shared between these control applications. Therefore, how to allocate the resources for the software implementation of the controllers forms the problem of architecture design. More specifically, these design parameters may include the task partition and mapping, the task and network scheduling, the use of the cache, etc. Towards the design of these parameters, ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration" provides an overview of successful approaches for system-level design space exploration for complex embedded systems.

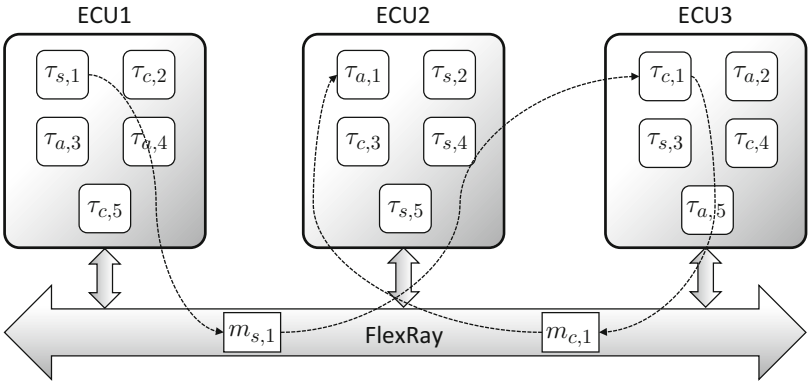

Fig. 37.2 An example of a distributed architecture for the embedded control systems. This example consists of 3 ECUs connected by a FlexRay bus. Five control applications are mapped on this architecture, where $\tau_{s,i}$, $\tau_{c,i}$, and $\tau_{a,i}$ denote respectively the sensor task, controller task, and actuator task of the ith control application. Two messages over the communication bus for first control application as well as the data dependency are shown

37.2.2 Feedback Control Systems

Throughout this chapter, linear single-input single-output (SISO) control applications are considered. The dynamic behavior is modeled by a set of differential equations,

$$\dot{x}(t) = Ax(t) + Bu(t), \quad y(t) = Cx(t), \tag{37.1}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $y(t)$ is the system output, and $u(t)$ is the control input. The number of system states is n. A, B, and C are system matrices of appropriate dimensions. System poles are eigenvalues of A. In a state-feedback control algorithm, $u(t)$ is computed utilizing $x(t)$ (feedback signals) and is then applied to the plant, which is expected to achieve certain desired behavior. In an embedded implementation platform, the operations (measure $x(t)$, compute $u(t)$, and apply $u(t)$) of a control loop are performed only at discrete time instants. In the case where the sensor-to-actuator delay d is ignored, the continuous-time system in (37.1) can be transformed into a discrete-time system with the sampling period h which can be represented as [10]

$$x[k+1] = A_d x[k] + B_d u[k], \quad y[k] = C_d x[k], \tag{37.2}$$

where sampling instants are $t = t_k$ $(k = 1, 2, 3, \cdots)$ and $h = t_{k+1} - t_k$. $x[k]$ and $u[k]$ are the values of $x(t)$ and $u(t)$ at $t = t_k$ and

$$A_d = e^{Ah}, \; B_d = \int_0^h (e^{At} dt) \cdot B, \; C_d = C. \tag{37.3}$$

A system is asymptotically stable if the steady-state impulse response is zero, i.e., $\lim_{k\to\infty} y_\delta[k] = 0$. Toward this, $u[k]$ needs to be designed utilizing the states $x[k]$ in a state-feedback controller. The general representation is as follows:

$$u[k] = K_d \cdot x[k] + F_d \cdot r, \tag{37.4}$$

where K_d is the feedback gain, F_d is the feedforward gain, and r is the reference value. Then, the system dynamics in (37.2) becomes

$$x[k+1] = (A_d + B_d K_d)x[k] + B_d F_d r, \tag{37.5}$$

i.e., closed-loop dynamics. Different locations of closed-loop system poles, i.e., eigenvalues of $(A_d + B_d K_d)$, result in different system behaviors. Pole locations can be decided by the pole-placement technique, and then the following characteristics equation of H can be constructed with these poles as roots:

$$H^n + \gamma_1 H^{n-1} + \gamma_2 H^{n-2} + \cdots + \gamma_n = 0. \tag{37.6}$$

Define

$$\gamma_c(A_d) = A_d^n + \gamma_1 A_d^{n-1} + \gamma_2 A_d^{n-2} + \cdots + \gamma_n \mathbf{I}, \tag{37.7}$$

where $\mathbf{I}$ is the n-dimensional identity matrix. According to Ackermann's formula [8], the feedback gain to stabilize the system is calculated as

$$K_d = -[0 \ \cdots \ 0 \ 1]\, \zeta^{-1}\, \gamma_c(A_d), \tag{37.8}$$

where ζ represents the controllability matrix of the system and is given by

$$\zeta = [B_d \ \ A_d B_d \ \ \ldots \ \ A_d^{n-1} B_d] \tag{37.9}$$

The static feedforward gain F is designed to achieve $y[k] \to r$ as $k \to \infty$ and can be computed by

$$F_d = \frac{1}{C_d(\mathbf{I} - A_d - B_d K_d)^{-1} B_d}. \tag{37.10}$$

However, in a realistic implementation of a control application, a non-negligible sensor-to-actuator delay needs to be taken into account. In the case, where the delay is smaller or equal to one sampling period, i.e., $0 \le d \le h$, the discrete-time system in (37.2) becomes a sampled-data system [12] as

$$x[k+1] = A_d x[k] + B_{d1}(d)u[k-1] + B_{d0}(d)u[k], \tag{37.11}$$

where

$$B_{d0}(D_c) = \int_0^{h-d} e^{At}dt \cdot B, \ B_{d1}(d) = \int_{h-d}^{h} e^{At}dt \cdot B. \tag{37.12}$$

In (37.11), it is assumed that $u[-1] = 0$ for $k = 0$. Notice that $x[k+1]$ depends on both $u[k]$ and $u[k-1]$, since during the sensor-to-actuator delay, $u[k]$ is not available and $u[k-1]$ is applied to the plant. A new system state $z[k] = \left[\, x[k] \ u[k-1] \,\right]^T$ is defined, and the transformed system becomes

$$z[k+1] = A_{aug} z[k] + B_{aug} u[k], \quad y[k] = C_{aug} z[k], \tag{37.13}$$

where

$$A_{aug} = \begin{bmatrix} A_d & B_{d1}(d) \\ 0 & 0 \end{bmatrix}, \ B_{aug} = \begin{bmatrix} B_{d0}(d) \\ \mathbf{I} \end{bmatrix}, \ C_{aug} = \begin{bmatrix} C_d \ 0 \end{bmatrix}. \tag{37.14}$$

Next, apply the following input signal:

$$u[k] = K_{aug} \cdot z[k] + F_{aug} \cdot r. \tag{37.15}$$

The closed-loop system is then

$$z[k+1] = (A_{aug} + B_{aug} K_{aug}) z[k] + B_{aug} F_{aug} r. \tag{37.16}$$

The feedback gain K_{aug} can then be calculated according to (37.8) by replacing A_d with A_{aug} and also replacing A_d and B_d with A_{aug} and B_{aug} while computing the controllability matrix ζ in (37.9). Similarly, the feedforward gain F_{aug} is computed according to (37.10) by replacing A_d, B_d, C_d, and K_d with A_{aug}, B_{aug}, C_{aug}, and K_{aug}, respectively.

37.2.2.1 Control Performance Metrics

There are different metrics to measure the performance of a control system. In this chapter, two common metrics to measure the control performance are considered. (i) The *steady-state performance* of a control application which can be commonly measured by a *cost function* [35], which in the discrete case can be represented as

$$J = \sum_{k=0}^{n} [\lambda u[k]^2 + (1-\lambda)\sigma[k]^2] h, \tag{37.17}$$

where λ is a weight taking the value between 0 and 1, $u[k]$ is the control input and $\sigma[k] = |r - y[k]|$ is the tracking error. (ii) The *settling time*, ξ, where ξ denotes the time necessary for the system to reach and remain within 1% of the reference value

$$J = \xi. \tag{37.18}$$

37.2.2.2 Optimal Pole Placement

For a control application, in order to design the controller which optimizes the control performance for a given sampling period, an optimization problem for the pole placement can be formulated. Decision variables are poles of the closed-loop system. Therefore, the number of dimensions in the decision space is equal to the number of states in the closed-loop system. The objective is to optimize the value of the selected control performance metric. Absolute values of all poles have to be less than unity to ensure system stability. The control input saturation needs to be respected as well.

It is challenging to solve such a constrained non-convex optimization problem with significant nonlinearity. Here, the Particle Swarm Optimization (PSO) technique, which is highly efficient and scalable [36], can be used. A group of particles are randomly initialized in the decision space with positions and velocities. They search for the optimum by iteratively updating their positions. The search is led by

two points. The first is the local best point that has been reached by a particle. Every particle has its own local best point. The second is the global best point that has been reached considering all particles. A point that respects all constraints is always better than a point that violates at least one constraint, no matter what their objective values are. When comparing two points that either respect all constraints or violate at least one constraint, the point with a better objective value is better.

The velocity of a particle is determined by the following equation:

$$\begin{aligned} V_{\text{new}} = {} & \alpha_0 V_{\text{current}} + \alpha_1 \text{rand}(0,1)(P_{\text{lbest}} - P_{\text{current}}) \\ & + \alpha_2 \text{rand}(0,1)(P_{\text{gbest}} - P_{\text{current}}), \end{aligned} \tag{37.19}$$

where V_{new} is the new velocity, V_{current} is the current velocity, P_{current} is the current position, P_{lbest} is the local best point of this particle, and P_{gbest} is the best point of all particles. rand(0, 1) is a random number with uniform distribution from the open interval (0, 1). α_0, α_1 and α_2 are parameters that can be determined empirically. The new position of this particle is

$$P_{\text{new}} = P_{\text{current}} + V_{\text{new}}. \tag{37.20}$$

The algorithm is terminated once all particles have converged or the maximum number of iterations has been reached. The time complexity of PSO is clearly polynomial.

37.3 Communication-Aware Control/Architecture Codesign

In this section, a codesign approach that synthesizes simultaneously controllers, task, and communication schedules for a FlexRay-based ECU network will be introduced. The approach consists mainly of two stages, namely, the control design stage and the cooptimization stage. This separation is necessary because the problem deals with a large design space combining the dimensions of both control and platform design. Therefore, the whole space is partitioned into smaller subspaces while considering all feasible regions in the design space by exploiting some domain-specific characteristics. In the control design stage, an optimal controller is synthesized at each possible sampling period for each control application. This is done by using the pole-placement control design method and exploring the design space for poles using heuristics. In the cooptimization stage, a bi-objective optimization problem is formulated, and a customized method is employed to generate a number of feasible design parameter sets, where each set represents a Pareto point reflecting the trade-off between the objectives of control performance and bus utilization. Here, we will first explain the problem setting and then discuss in detail the state-of-the-art control-communication codesign technique applicable to such a setting.

37.3.1 Problem Setting

Distributed implementation: Consider a distributed architecture where a set of ECUs represented by $p_i \in \mathbb{P}$ are connected through a FlexRay bus. A number of control applications denoted by $C_i \in \mathscr{C}$ are mapped on such an embedded platform. Each control application C_i can be partitioned into three dependent application tasks: (i) *sensor task*, $\tau_{s,i}$, measures the system states (using sensors) of the physical system if measurable; (ii) *controller task*, $\tau_{c,i}$, computes the controller input based on the measured system states; and (iii) *actuator task*, $\tau_{a,i}$, applies the control input (using actuators) to the physical system. Without loss of generality, assume that the three tasks are mapped on different ECUs. Then the sensor values measured by $\tau_{s,i}$ are sent on the bus through message $f_{s,i}$, and the control input calculated by $\tau_{c,i}$ is sent as message $f_{c,i}$. The time between the start of sensor task and the end of actuator task is defined as the *sensor-to-actuator delay*, denoted as d. As shown in Fig. 37.3, this delay depends on the interplay between the task and communication schedules.

ECU task model: Here, consider the case where time-triggered non-preemptive scheduling scheme is exhibited by the Real-Time Operating System (RTOS) on the ECUs. Each task of the control applications is considered to be periodic and is defined by the tuple $\tau_{x,i} = \{O_{x,i}, P_{x,i}, E_{x,i}\}$, where $O_{x,i}$, $P_{x,i}$, and $E_{x,i}$ denote respectively the offset, the period, and the WCET of the task. Here, the subscript $x \in \{s, c, a\}$ where s, c, or a respectively identifies sensor, controller, or actuator task. The subscript i identifies the control application C_i it constitutes. Thus, if $\bar{t}(\tau_{x,i}, k)$ and $\tilde{t}(\tau_{x,i}, k)$ are defined as the starting and the latest finishing time of the k^{th} ($k \in \mathbb{Z}^*$) instance of task τ_i, then

$$\bar{t}(\tau_{x,i}, k) = O_{x,i} + kP_{x,i}, \quad \tilde{t}(\tau_{x,i}, k) = O_{x,i} + kP_{x,i} + E_{x,i}. \tag{37.21}$$

A set of *communication tasks* are required besides the application tasks. The communication task on the sending ECU writes the data produced by the application tasks into the corresponding *transmit buffers* of the communication controller, and on the receiving ECU, it reads the data from the corresponding *receive buffers* and

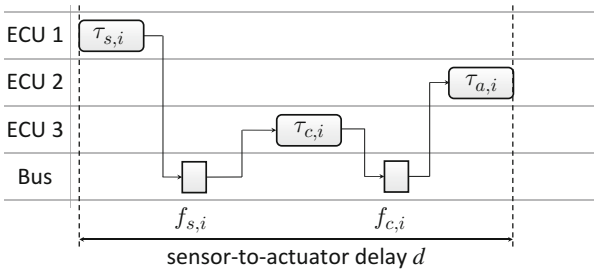

Fig. 37.3 Distributed embedded control application

forwards them to the application tasks. The nature of these communication tasks depends on the specific implementation. Here, consider that the execution time of all communication tasks is bounded by ϵ, and assume that a communication task is scheduled directly after its corresponding application task at the sending side and directly before the application task at the receiving side.

FlexRay communication: FlexRay [3] is an automotive communication protocol usually applied for safety-critical applications. Although FlexRay communication is discussed in detail in ▸ Chap. 24, "Networked Real-Time Embedded Systems", few important points are reiterated here for better understanding of the problem and the subsequent solution. Being a hybrid protocol, it offers both TT and ET communication services. FlexRay is organized as a series of *communication cycles*, the length of which is denoted as T_{bus}. Each communication cycle contains mainly the *static segment* (ST) and optionally *dynamic segment* (DYN), where the TT and ET communication services are implemented respectively. The static segment applies the TDMA scheme and is split into a number of *static slots* of equal length Δ. Here, the slots on the static segment can be represented as $\mathscr{S}_{ST} = \{1, 2, \ldots, N_s\}$, where N_s is the number of static slots. Once a static slot is assigned, if no data is sent in a specific communication cycle, the static slot will still be occupied. The dynamic segment follows a Flexible Time Division Multiple Access (FTDMA) approach, where the segment is divided into a number of *mini-slots* of equal length δ. A *dynamic slot* is a logical entity, which can consist of one or more mini-slots, depending on whether data is sent on the slot and how much data is sent. Once a dynamic slot is assigned, if no data is sent in a communication cycle, only one mini-slot is consumed. If data is to be sent, a number of mini-slots are occupied to accommodate the data. The dynamic slots can be represented as $\mathscr{S}_{DYN} = \{N_s + 1, \ldots, N_s + N_{ms}\}$, where N_{ms} is the number of mini-slots.

The communication cycles are organized as sequences of 64 cycles. In a sequence, each communication cycle is indexed by a *cycle counter* which counts from 0 to 63 and is then set to 0. A FlexRay schedule corresponding to the message $f_{x,i}$ can be defined as $\Theta_{x,i} = (s_{x,i}, q_{x,i}, r_{x,i})$, where $s_{x,i}$ represents the slot number, $q_{x,i}$ represents the *base cycle*, and $r_{x,i}$ represents the *repetition rate*. Here, the subscript $x \in \{s, c\}$ where s or c respectively identifies sensor or control message. The subscript i identifies the control application C_i it constitutes. Here, the repetition rate $r_{x,i}$ is the number of communication cycles that elapse between two consecutive transmissions of the same frame and takes the value $r_{x,i} \in \{2^n | n \in \{0, \ldots, 6\}\}$. The base cycle $q_{x,i}$ is the offset of the cycle counter. The sequence of 64 communication cycles and some examples of FlexRay schedules are shown in Fig. 37.4. Here, the FlexRay Version 3.0.1 [3] is considered, where *slot multiplexing* among different ECUs is allowed. It means that a particular slot $s \in \mathscr{S}_{ST} \cup \mathscr{S}_{DYN}$ can be assigned to different ECUs in different communication cycles. Further consider all messages are sent over the static segment of the FlexRay bus, i.e., on the static slots. The starting and ending time of the k^{th} instance ($k \in \mathbb{Z}^*$) of the FlexRay schedule Θ_i, which are denoted respectively as $\bar{t}(\Theta_i, k)$ and $\tilde{t}(\Theta_i, k)$, can be defined as

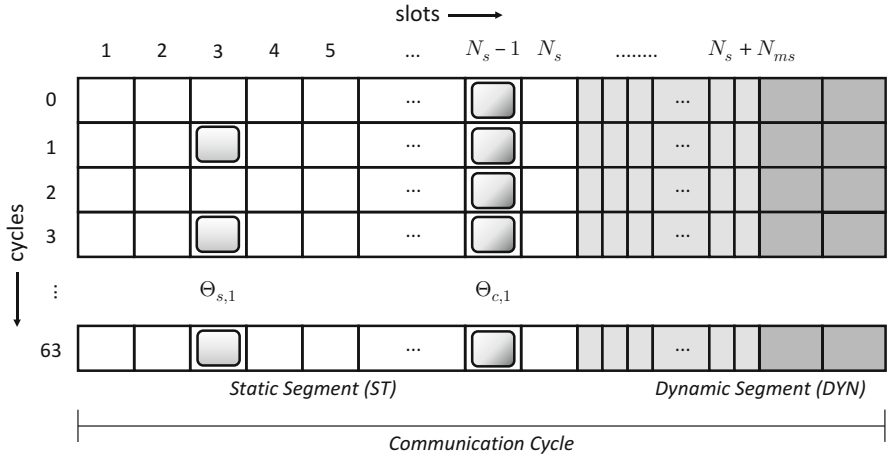

Fig. 37.4 An example of FlexRay schedules

$$\begin{aligned} \bar{t}(\Theta_{x,i}, k) &= q_{x,i} T_{bus} + k r_{x,i} T_{bus} + (s_{x,i} - 1)\Delta, \\ \tilde{t}(\Theta_{x,i}, k) &= q_{x,i} T_{bus} + k r_{x,i} T_{bus} + s_{x,i} \Delta. \end{aligned} \tag{37.22}$$

For FlexRay time-triggered communication, the bus utilization can be defined as the percentage of bandwidth of the static segment that is allocated to the control applications. This can be represented as the percentage of static slots allocated to the control applications in 64 consecutive communication cycles. In this case, the smaller the value of U, the better is the resource efficiency as more number of slots can be left vacant for use by other non-control applications. Now, let Γ denote the set of all FlexRay schedules allocated to the control applications on the static segment, where $\Theta_{x,i} \in \Gamma$; then the bus utilization U can be defined as

$$U = \frac{100}{64 N_s} \sum_{\Theta_{x,i} \in \Gamma} \frac{64}{r_{x,i}}, \tag{37.23}$$

where $64N_s$ is the total number of static slots in 64 consecutive communication cycles. Here, $r_{x,i}$ represents the repetition rate of the message $f_{x,i}$, and therefore, $\frac{64}{r_{x,i}}$ represents the number of static slot allocated to the message $f_{x,i}$ in 64 consecutive communication cycles.

Control performance: Depending on the specific requirements of the control application, one of the two performance metric discussed in Sect. 37.2.2.1 can be used. For a specific control application C_i, J_i depends both on the sampling period h_i and the control gains $K_{aug,i}$ and $F_{aug,i}$. In both the performance metrics, smaller value of J implies better control performance. In a system consisting of multiple control applications with different plant models and performance metrics,

it is required to normalize the control performance in order to compare and combine them. Each control system C_i with control performance J_i must satisfy some control performance requirement J_i^r defined by the user. Thus, the control performance can be normalized as follows:

$$J_i^n = \frac{100 \cdot J_i}{J_i^r} \tag{37.24}$$

and thus the overall control performance of a set of control applications $\mathscr{C}$ can be represented as a weighted sum

$$J_o = \sum_{C_i \in \mathscr{C}} w_i J_i^n, \tag{37.25}$$

where w_i stands for the weight and $\sum_i w_i = 1$.

Cooptimization problem: The cooptimization problem boils down to finding a set of parameters for each $C_i \in \mathscr{C}$, which can be denoted as $par_i = \{\tau_{s,i}, \tau_{c,i}, \tau_{a,i}, \Theta_{s,i}, \Theta_{c,i}, h_i, K_{aug,i}, F_{aug,i}\}$, while optimizing the total FlexRay bus utilization and the overall control performance given by Eqs. (37.23) and (37.25), respectively. Here, the control parameters of C_i can be further defined as $par_i^c = \{h_i, K_{aug,i}, F_{aug,i}\}$ and similarly the embedded platform parameters as $par_i^s = \{\tau_{s,i}, \tau_{c,i}, \tau_{a,i}, \Theta_{s,i}, \Theta_{c,i}\}$, where $par_i = par_i^s \cup par_i^c$. The parameter set of the whole system is represented as $\mathscr{P}$, where $par_i \in \mathscr{P}$.

37.3.2 The Codesign Approach

37.3.2.1 Design Flow

Figure 37.5 shows the design flow of the codesign approach. The whole design process is divided into two stages. In the first stage, for each control application, possible controllers that optimize the control performance at different sampling periods are synthesized and the results are recorded in a look-up table. In the second stage, the cooptimization stage, both the control and the platform parameters are synthesized based on the constraints, objectives, and the look-up tables obtained in the first stage. Here, a bi-objective optimization problem is formulated, and a customized approach is used to generate a Pareto front of the two objectives considered. In this stage, the fact that the bus utilization objective U can only take selected discrete values is exploited, and therefore, for each of those values, a nested two-layered optimization technique is employed to find a feasible set of parameters that represents a Pareto point and optimizes the control performance or to prove that a corresponding Pareto point is not possible. Here, Layer 1 tries to find a set of values of sampling periods corresponding to the set of control applications such that it can represent a Pareto point and it optimizes the overall system control performance for a given value of bus utilization. Then, Layer 2 tries to find a

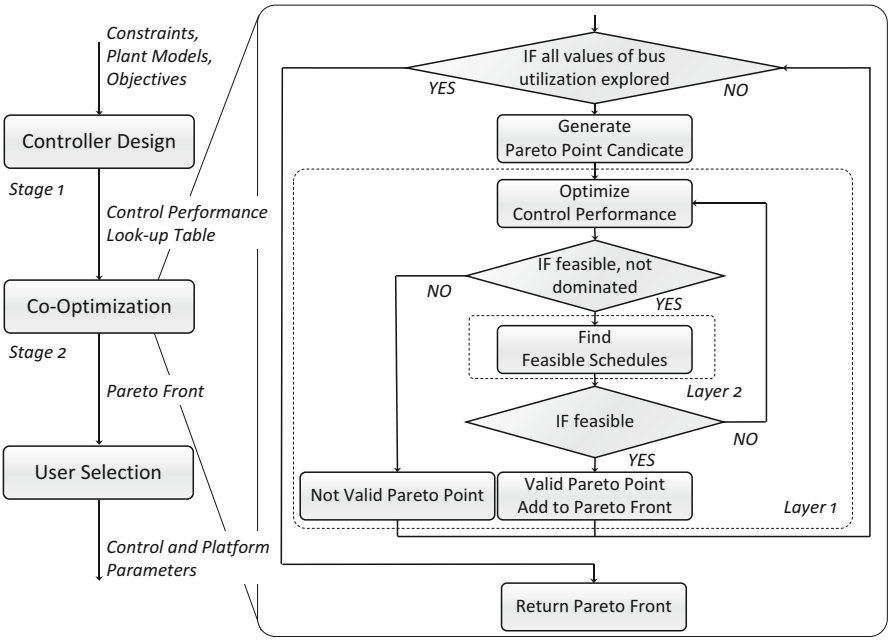

Fig. 37.5 Design flow of the cooptimization approach

feasible schedule set (by solving a constraint programming problem) and control gains (from the look-up tables) corresponding to the sampling period values of the control applications determined in Layer 1. The nested two-layered optimization technique is discussed in further detail in Sect. 37.3.2.4. Based on the Pareto front thus obtained, the designer can select one set of parameters that is the most suitable for the overall design requirements. The control design stage and the cooptimization stage of this approach will be explained in detail in the following sections.

37.3.2.2 Controller Design

Besides the control plant model, the performance J_i of the control application C_i depends mainly on three factors: (i) the sampling period h_i, (ii) the sensor-to-actuator delay d_i, and (iii) the control gains $K_{aug,i}$ and $F_{aug,i}$. Depending on each combination of the sampling period and delay, a set of optimal control gains needs to be designed. Here, consider schedules for the control tasks and the messages leading to the case where the delay equals to the sampling period, i.e., $d_i = h_i$. This would reduce the dimensions of the design space from all three factors (i)–(iii) to only (i) and (iii), thus reducing the complexity and enhancing the scalability. It should be noted that this approach can be easily adapted to other cases with a fixed delay value (e.g., $d_i = D_i$, where D_i is a constant and $D_i \leq h_i$) or a delay value proportional to the sampling period (e.g., $d_i = \psi h_i$, where $\psi \leq 1$). With $d_i = h_i$, the closed-loop system experiences one sampling period delay, and the pole-placement method

reported in Sect. 37.2 can be used for such delayed system. To the best of our knowledge, there is no standard closed-form optimal control design framework that can be directly applied in such a delayed system. Therefore, the PSO-based optimal pole-placement technique described in Sect. 37.2.2.2 is employed, which can be quite computationally costly for higher-order control plants. However, making use of the fact that the sampling period can only take discrete values, the design space can be pruned. Since each control application C_i is implemented by the tasks $\tau_{s,i}$, $\tau_{c,i}$, and $\tau_{a,i}$ and messages $f_{s,i}$, $f_{c,i}$, there is a dependency between the sampling period h_i and the repetition rate of the messages $r_{s,i}$, $r_{c,i}$, which can be represented as

$$h_i = r_{s,i} T_{bus} = r_{c,i} T_{bus}. \tag{37.26}$$

Due to the fact that $r_{s,i}$, $r_{c,i}$ can only take discrete values in $\{2^k | k \in \{0, \ldots, 6\}\}$, the choice of h_i is also constrained to the corresponding discrete values.

Denote the control performance as $J_i = f(h_i, K_{aug,i}, F_{aug,i})$. Then the control performance at each discrete value $h_i^k = 2^k T_{bus}$ of the sampling period can be represented as $J_i(h_i^k) = g(K_{aug,i}^k, F_{aug,i}^k)$. The purpose of the controller design step is to determine the control gains for each possible value of the sampling period that optimizes the control performance. Employing the optimal pole-placement technique, determine the set of control gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ that optimizes the control performance to G_i^{k*} at sampling period h_i^k, then represent the optimal control performance at h_i^k as $J_i^*(h_i^k) = G_i^{k*}$. The control design problem can be translated into the problem of finding for each discrete value h_i^k, a set of gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ that optimizes the control performance $J_i(h_i^k)$ to the value of G_i^{k*}.

After this stage, a look-up table for each control application C_i can be formulated where for each of the possible sampling period h_i^k an optimal control performance G_i^{k*} corresponding to the control gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ can be assigned. In the cooptimization stage, this set of tables will be used to formulate the objective of overall control performance.

37.3.2.3 Optimization Problem Formulation

The system constraints for the FlexRay-based ECU system are well studied and discussed in [23, 29, 35]. Here, we will state the majority of the constraints formulated there.

(C1) Sampling period constraint: The tasks and messages of a control application must have the same period of repetition which is also the sampling period of the system. This constraint can be formulated as

$$\forall C_i \in \mathscr{C}, x \in \{s, c, a\}, y \in \{s, c\}, \qquad P_{x,i} = r_{y,i} T_{bus} = h_i. \tag{37.27}$$

(C2) Data-flow constraint: In a control application, all task executions and message transmissions must be in correct temporal order, as illustrated in Fig. 37.3. This can be formulated as set of constraints as

$$\begin{aligned}
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{s,i}, k) + \epsilon < \bar{t}(\Theta_{s,i}, k),\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\theta_{s,i}, k) < \bar{t}(\tau_{c,i}, k) - \epsilon,\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{c,i}, k) + \epsilon < \bar{t}(\Theta_{c,i}, k),\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\theta_{c,i}, k) < \bar{t}(\tau_{a,i}, k) - \epsilon.
\end{aligned} \tag{37.28}$$

(C3) Sensor-to-actuator delay constraint: The constraint stating that the sensor-to-actuator delay for the control applications is equal to exactly one sampling period can be formulated as

$$\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{a,i}, k+1) - \bar{t}(\tau_{s,i}, k) = h_i. \tag{37.29}$$

(C4) Non-overlapping task constraint: In a time-triggered non-preemptive scheduling scheme as considered in this paper, when more than one task is mapped on an ECU, they must be scheduled in such a way that they do not overlap. This can be formulated as a constraint as

$$\begin{aligned}
&\forall \quad C_i, C_j \in \mathscr{C}, \quad x, y \in \{s, c, a\}, \quad p_k \in \mathbb{P}\\
&\forall \quad \{m \in \mathbb{Z}^* | 0 \le m < lcm(P_{x,i}, P_{y,j})/P_{x,i}\},\\
&\quad \{n \in \mathbb{Z}^* | 0 \le n < lcm(P_{x,i}, P_{y,j})/P_{y,j}\}\\
&\textbf{if} \quad \tau_{x,i}, \tau_{y,j} \in \mathscr{T}_{p_k} \quad \textbf{then} \quad \tilde{t}(\tau_{x,i}, m) + \epsilon \cdot \mathbb{1}(x \in \{s, c\}) < \bar{t}(\tau_{y,j}, n)\\
&\quad - \epsilon \cdot \mathbb{1}(y \in \{c, a\})\\
&\textbf{or} \quad \tilde{t}(\tau_{y,j}, n) + \epsilon \cdot \mathbb{1}(y \in \{s, c\}) < \bar{t}(\tau_{x,i}, m) - \epsilon \cdot \mathbb{1}(x \in \{c, a\}),
\end{aligned} \tag{37.30}$$

where $\mathscr{T}_{p_k}$ denotes the set of all tasks mapped on ECU E_k. $\mathbb{1}(.)$ is the indicator function and takes the value of 1 if the input is true and 0 if otherwise.

(C5) Nonoverlapping message constraint: FlexRay messages must be scheduled in such a way that no two messages share the same slot in the same cycle. This constraint can be established as

$$\begin{aligned}
&\forall \quad C_i, C_j \in \mathscr{C}, \quad x, y \in \{s, c\}\\
&\forall \{n \in \mathbb{Z}^* | 0 \le n < max(r_{x,i}, r_{y,j})/r_{x,i}\},\\
&\quad \{m \in \mathbb{Z}^* | 0 \le m < max(r_{x,i}, r_{y,j})/r_{y,j}\},\\
&\textbf{if} \quad s_{x,i} == s_{y,j} \quad \textbf{then} \quad q_{x,i} + n r_{x,i} \neq q_{y,j} + m r_{y,j}.
\end{aligned} \tag{37.31}$$

(C6) FlexRay scheduling constraint: Taking into consideration the scheduling constraints imposed by the FlexRay protocol, it is required to constrain $s_{x,i}$ and $q_{x,i}$ as

$$\begin{aligned} &\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c\}, \quad 1 \leq s_{x,i} \leq N_s \\ &\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c\}, \quad 0 \leq q_{x,i} < r_{x,i}. \end{aligned} \tag{37.32}$$

In addition, the bus utilization U is constrained by the total number of static slots available in 64 communication cycles.

$$U \leq 100. \tag{37.33}$$

(C7) ECU scheduling constraint: On ECUs, for task schedules, consider

$$\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c, a\}, \quad 0 \leq O_{x,i} + E_{x,i} < P_{x,i}. \tag{37.34}$$

Moreover, the ECU load cannot be more than 100%.

$$\forall p_k \in \mathbb{P}, x \in \{s, c, a\}, \sum_{\tau_{x,i} \in \mathscr{T}_{p_k}} \frac{E_{x,i} + \epsilon + \epsilon \cdot \mathbb{1}(x \in \{c\})}{P_{x,i}} \leq 1., \tag{37.35}$$

(C8) Performance constraint: For each control system C_i with sampling period h_i, user specifies a control performance requirement J_i^r. As mentioned in Sect. 37.3.2.2, a look-up table for each control system is developed which contains the performance of seven possible controllers corresponding to seven possible sampling periods. Therefore, the domain of h_i, denoted as $dom[h_i]$ is constrained according to control performance requirement as

$$\forall k \in \{0, 1, \ldots 6\}, \quad J_i^*(h_i^k) \leq J_i^r \iff h_i^k \in dom[h_i]. \tag{37.36}$$

Now, let J_i represent the control performance of C_i. Therefore,

$$h_i == 2^k T_{bus} \iff J_i == J_i^*(h_i^k). \tag{37.37}$$

As the objectives for the optimization problem, the overall system control performance and the bus utilization are considered.

(O1) Overall system control performance:

$$J_o = \sum_{C_i \in \mathscr{C}} w_i J_i^n = \sum_{C_i \in \mathscr{C}} w_i \sum_k \mu_{i,k} J_i^{n*}(h_i^k), \tag{37.38}$$

where $\mu_{i,k}$ are binary variables satisfying $\sum_k \mu_{i,k} = 1$ and $J_i^{n*}(h_i^k)$ represents the normalized optimal control performance of C_i at h_i^k, which can be formulated as

$$J_i^{n*}(h_i^k) = \frac{100 J_i^*(h_i^k)}{J_i^r}. \tag{37.39}$$

(O2) Bus utilization: The bus utilization in this case can be defined as

$$U = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \left(\frac{64}{r_{s,i}} + \frac{64}{r_{c,i}} \right) = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{h_i}. \tag{37.40}$$

The value of the bus utilization can only take certain discrete values and is bounded by the upper and lower limit U^+ and U^-, which can be expressed as

$$U^+ = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{\max\limits_{h_i \in dom[h_i]} (h_i)}, \qquad U^- = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{\min\limits_{h_i \in dom[h_i]} (h_i)}. \tag{37.41}$$

37.3.2.4 Multi-objective Optimization

As discussed above, the control and system codesign of the setting considered can be formulated as a constrained optimization problem with two objectives, namely, the bus utilization and overall control performance. In this case, the two design objectives are noticed to be often conflicting, and therefore, as discussed in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration", a much more informative and designer-friendly cooptimization approach is to first generate a Pareto front, and let the designer explore the trade-off between the two objectives according to his customized preference.

▸ Chapters 6, "Optimization Strategies in Design Space Exploration", ▸ 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration", and ▸ 9, "Scenario-Based Design Space Exploration" have emphasized on hybrid optimization techniques to solve such a Design Space Exploration (DSE) problem. Such techniques depend heavily on problem characteristics, desired accuracy and scalability, etc. Consequently, for this problem, a customized hybrid optimization approach as shown in Fig. 37.5 is employed to obtain the desired Pareto front. Since the objective on bus utilization U is discrete and only takes a limited number of integers, first compute the maximum and minimum bus utilization U^+ and U^-, which bound the set of U. For each possible value of U from U^- to U^+, i.e., given the equality constraint on U, solve the optimization problem with J_o as the single objective and obtain a solution. The additional constraint is that J_o of this solution has to be better than J_o of the last solution (Pareto criterion), in order to ensure that all solutions are non-dominated. Therefore, the cooptimization problem with two objectives is turned into a series of single-objective optimization problems, where each may generate a Pareto point on the Pareto front.

Popular approaches like Mixed Integer Linear Programming (MILP) or meta-heuristic methods cannot be applied directly to solve each of the single-objective optimization problems. However, considering that some decision variables only appear in constraints, but are not related to the objective, a nested two-layered

technique is employed to solve each of the problems. On Layer 1, the outer layer, consider only constraint (C8) and an equality constraint on bus utilization U translated from (O2), and optimize the (O1). Decision variables related to the objectives, i.e., the sampling periods, are determined. On Layer 2, the inner layer, the remaining decision variables are synthesized satisfying the constraints (C1)–(C7) while substituting the values of sampling periods based on the results of Layer 1. This process is iterative in the way that if the synthesis fails in Layer 2, the algorithm goes back to Layer 1 for the next best solution until Pareto criterion is satisfied. This optimization technique ensures optimality and also efficiency.

37.3.3 Case Study

In the case study, five control applications denoted as $\mathscr{C} = \{C_1, C_2, C_3, C_4, C_5\}$ are considered. For each of the control applications, a plant model derived from the automotive domain is used. C_1 to C_5 represent respectively the DC motor speed control (DCM), servo motor position control (DCP), the electronic braking control (EBC), the car suspension (CSS), and the adaptive cruise control (ACC). The hardware platform consists of three ECUs $\{E_1, E_2, E_3\}$ connected by FlexRay bus. Tables 37.1 and 37.2 show the task mappings and FlexRay bus configuration, respectively.

Figure 37.6 shows the results of the normalized optimal control performance for each control application as the sampling period increases. The thick red dashed line in the plot shows the normalized required performance for all the control applications (i.e., 100%). Only the points below the red line meet the design requirement for performance, and only these points will be considered in the following cooptimization stage. The Pareto front of the whole system in the case

Table 37.1 Task mapping

ECUs	Tasks
E_1	$\tau_{s,1}$, $\tau_{c,2}$, $\tau_{a,3}$,
	$\tau_{a,4}$, $\tau_{c,5}$
E_2	$\tau_{a,1}$, $\tau_{s,2}$, $\tau_{c,3}$,
	$\tau_{s,4}$, $\tau_{s,5}$
E_3	$\tau_{c,1}$, $\tau_{a,2}$, $\tau_{s,3}$,
	$\tau_{c,4}$, $\tau_{a,5}$

Table 37.2 FlexRay bus configuration

Bus parameters	Values
Bus speed	10 Mbps
T_{bus}	5 ms
N	25
M	237
Δ	100 ms
δ	10 ms

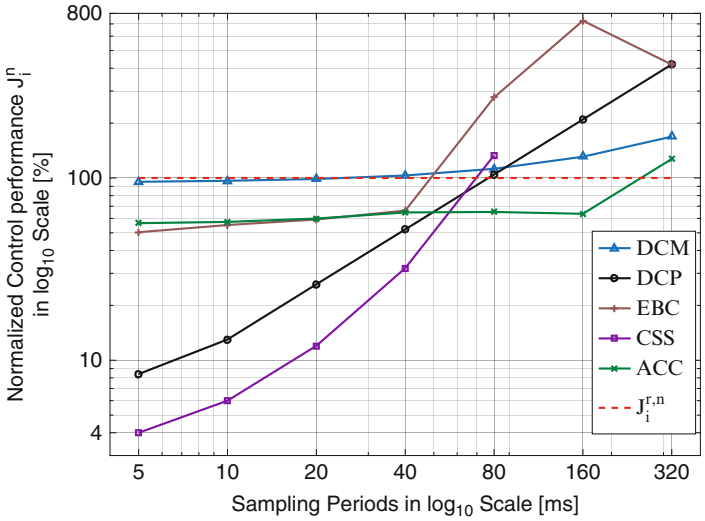

Fig. 37.6 Control performance

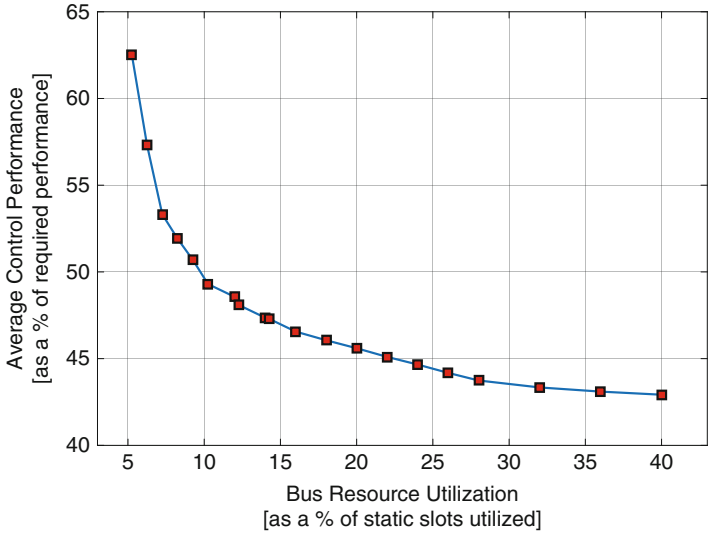

Fig. 37.7 Pareto front

study obtained in the cooptimization stage is shown in Fig. 37.7. The value of the bus utilization ranges from 5.25% to 40% of the bus bandwidth in the static segment. The value of the control performance varies on an average from 42.92% to 62.54% of the required value for each control application. It should be noted that for the control performance defined here, the smaller the value, the better the performance. It is obvious that there is a large freedom among these viable design points.

37.4 Memory-Aware Control/Architecture Codesign

While the memory-aware optimization of embedded software has been discussed in ▸ Chap. 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives", in this section, how to exploit the instruction cache reuse to improve the control performance is shown. Given a collection of control applications (e.g., C_1, C_2, C_3) on one processing unit, it is conventional to run the control loops of them in a round-robin fashion ($C_1, C_2, C_3, C_1, C_2, C_3, \cdots$). Since the programs for different control applications are different, the cache in this process is frequently refreshed. This results in poor cache reuse and long WCET. In order to address this issue, a memory-aware sampling order for the control applications can be applied, using which cache reuse is improved and the WCET of each application is reduced. In particular, we study a nonuniform sampling scheme, where the control loop of each application is consecutively run multiple times – in order to increase the cache reuse – before moving on to the next application (e.g., $C_1, C_1, C_1, C_2, C_2, C_2, C_3, C_3, C_3, \cdots$). As illustrated in Fig. 37.8, where $C_i(j)$ denotes the jth execution of the control application C_i, before the first execution $C_i(1)$, the cache is either empty (i.e., cold cache) or filled with instructions from other applications that are not used by C_i (equivalent to cold cache). The WCET of $C_i(1)$ can be computed by a number of existing standard techniques [9,37,38]. Before the second execution $C_i(2)$, the instructions in the cache are from the same application C_i and thus can be reused. This results in more cache hits and hence shorter WCET. Depending on which path the program takes, the amount of WCET reduction varies. Therefore, a technique is required to compute the guaranteed WCET reduction of $C_i(2)$ and $C_i(3)$, independent of the path taken, which will be presented later in this section. Control parameters of the systems, such as sampling periods and sensor-to-actuator delays, are derived from the WCET results. A controller must be tailored for the memory-aware nonuniform sampling orders, in order to improve the control performance. In summary, two main techniques are required and explained as

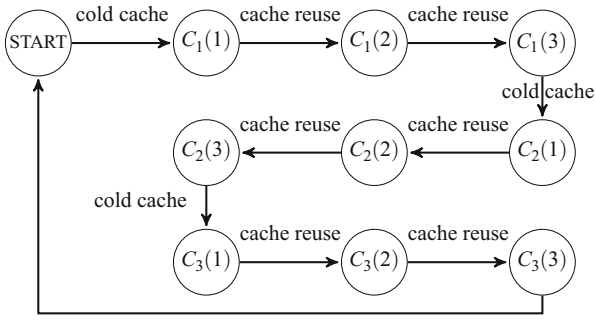

Fig. 37.8 An example memory-aware sampling order with three applications. Each application is consecutively executed three times. After the first execution $C_i(1)$, some instructions in the cache can be reused, and thus the WCETs of the following two executions are shortened

follows: (i) cache analysis to compute the guaranteed WCET reduction between two consecutive executions of one program and (ii) controller design for the nonuniform sampling.

37.4.1 Cache Analysis for Consecutive Executions of a Control Application

As discussed in Sect. 37.1, a two-level memory hierarchy – cache and main memory – is considered. More information about the memory architecture can be found in ▸ Chap. 13, "Memory Architectures". There are N_c cache lines, denoted as $CL = \{c_0, c_1, \ldots, c_{N_c-1}\}$, and the main memory has N_m blocks, denoted as $M = \{m_0, m_1, \ldots, m_{N_m-1}\}$. Each memory block is mapped to a fixed cache line. An example is shown in Fig. 37.9 for the illustration purpose, where there are four cache lines and five memory blocks. A basic block is a straight-line sequence of code with only one entry point and one exit point. This restriction makes a basic block

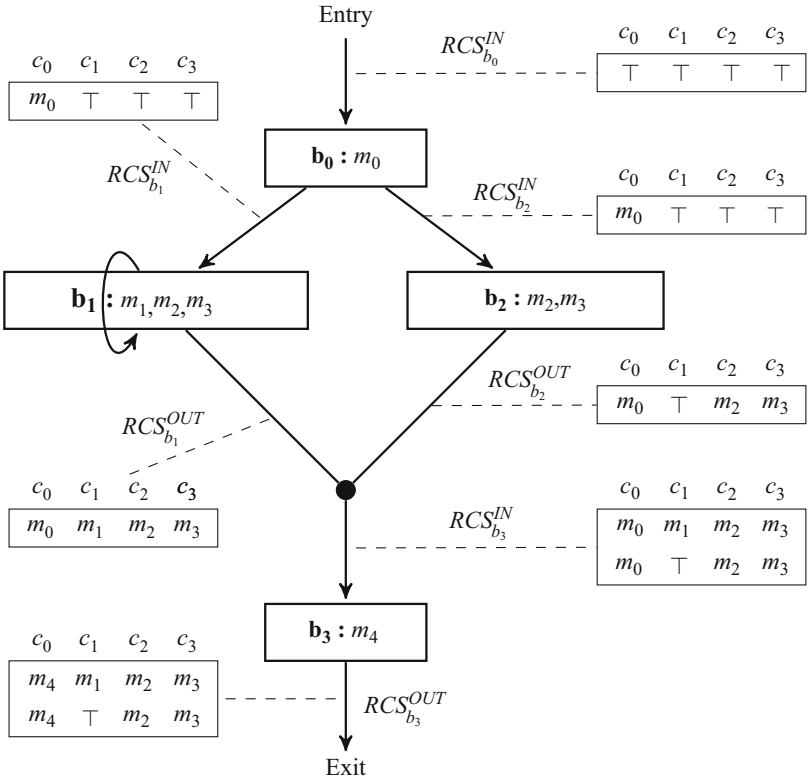

Fig. 37.9 A motivational example for cache analysis. Five memory blocks are mapped to four cache lines. Memory blocks executed by each basic block are shown. RCS^{IN} and RCS^{OUT} in the initialization phase are illustrated

highly amenable for program analysis. The presented Control-Flow Graph (CFG) in Fig. 37.9, consisting of four basic blocks $B = \{b_0, b_1, b_2, b_3\}$, has all the three key elements of a control program, i.e., sequential basic blocks, branches, and a loop. Therefore, it is suitable for illustrating our cache analysis technique.

There are three key terms in cache analysis that are described as follows:

- *Cache States*: A cache state cs is described as a vector of N_c elements. Each element $cs[i]$, where $i \in \{0, 1, \ldots, N_c - 1\}$, represents the memory block in the cache line c_i. When the cache line c_i holds the memory block m_j, where $j \in \{0, 1, \ldots, N_m - 1\}$, $cs[i] = m_j$. If c_i is empty, it is denoted as $cs[i] = \perp$. If the memory block in c_i is unknown, it is denoted as $cs[i] = \top$. CS is the set of all possible cache states.
- *Reaching Cache States (RCS)*: RCS of a basic block b_k, denoted as RCS_{b_k}, is the set of all possible cache states when b_k is reached via any incoming path.
- *Live Cache States (LCS)*: LCS of a basic block b_k, denoted as LCS_{b_k}, is the set of all possible first memory references to cache lines at b_k via any outgoing path.

Since our focus is on WCET reduction between two consecutive executions of C_i, it is necessary to compute RCS of the exit point in the first execution of C_i and LCS of the entry point in the second execution of C_i. By comparing all possible pairs of cache states, the guaranteed number of cache hits, and thus WCET reduction can be calculated. In the following, computation of RCS and LCS is firstly discussed.

In RCS computation, gen_{b_k} is firstly defined as the cache state describing the last executed memory block in every cache line for the basic block b_k. Assuming that b_0 in Fig. 37.9 executes m_0 and then m_4, instead of only m_0, the last executed memory block in c_0 is m_4. Therefore, gen_{b_0} is $[m_4, \perp, \perp, \perp]$. For the example in Fig. 37.9,

$$\begin{aligned} gen_{b_0} &= [m_0, \perp, \perp, \perp], \quad & gen_{b_1} &= [\perp, m_1, m_2, m_3], \\ gen_{b_2} &= [\perp, \perp, m_2, m_3], \quad & gen_{b_3} &= [m_4, \perp, \perp, \perp]. \end{aligned} \tag{37.42}$$

There are two equations involved in the RCS computation that calculate RCS^{IN} and RCS^{OUT}, where RCS^{IN} of a basic block b_k is the RCS before b_k is executed and RCS^{OUT} is the set of all possible cache states after b_k is executed. First, $RCS^{OUT}_{b_k}$ can be calculated from $RCS^{IN}_{b_k}$ as

$$RCS^{OUT}_{b_k} = \{\mathscr{T}(b_k, cs) | cs \in RCS^{IN}_{b_k}\}, \tag{37.43}$$

where $\mathscr{T}$ is a transfer function defined as follows: For any cache state $cs \in CS$ and basic block $b_k \in B$, there is a cache state $cs' = \mathscr{T}(b_k, cs)$, where for any cache line $c_i \in CL$ and $i \in \{0, 1, \ldots, N_c - 1\}$,

Table 37.3 RCS computation for the motivational example

	Basic block	RCS^{IN}	RCS^{OUT}
Initialization	b_0	$\{[\top, \top, \top, \top]\}$	$\{[m_0, \top, \top, \top]\}$
	b_1	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, m_1, m_2, m_3]\}$
	b_2	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, \top, m_2, m_3]\}$
	b_3	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$
Fixed-point	b_0	$\{[\top, \top, \top, \top]\}$	$\{[m_0, \top, \top, \top]\}$
	b_1	$\{[m_0, \top, \top, \top], [m_0, m_1, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3]\}$
	b_2	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, \top, m_2, m_3]\}$
	b_3	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$

$$cs'[i] = \begin{cases} cs[i] & : if\ gen_{b_k}[i] = \perp; \\ gen_{b_k}[i] & : otherwise. \end{cases} \tag{37.44}$$

$RCS^{IN}_{b_k}$ can be calculated as

$$RCS^{IN}_{b_k} = \bigcup_{p \in predecessor(b_k)} RCS^{OUT}_{p}, \tag{37.45}$$

where $predecessor(b_k)$ is the set of all immediate predecessors of b_k.

The RCS computation is composed of two phases: initialization and fixed-point computation. As illustrated with the example in Fig. 37.9, the initialization phase starts from the entry basic block b_0 with $RCS^{IN}_{b_0} = \{[\top, \top, \top, \top]\}$. The element is $\top$ since our analysis is independent of the program executed before b_0. According to (37.43), $RCS^{OUT}_{b_0}$ is calculated to be $\{[m_0, \top, \top, \top]\}$. Since b_0 is the only immediate predecessor of b_2, $RCS^{IN}_{b_2}$ is equal to $RCS^{OUT}_{b_0}$ based on (37.45). Due to the self-loop, b_1 has both itself and b_0 as immediate predecessors. However, since $RCS^{OUT}_{b_1}$ has not been initialized yet, $RCS^{IN}_{b_1}$ is equal to $RCS^{OUT}_{b_0}$. In the same manner, $RCS^{OUT}_{b_1}$, $RCS^{OUT}_{b_2}$, $RCS^{IN}_{b_3}$, and $RCS^{OUT}_{b_3}$ can be computed, following the program flow as shown both in Fig. 37.9 and Table 37.3. The initialization phase is completed once all basic blocks have been visited. The next phase is fixed-point computation. RCS^{IN} and RCS^{OUT} of all basic blocks are computed iteratively with (37.45) and (37.43). This phase is terminated once the fixed point is reached, i.e., RCS^{IN} and RCS^{OUT} of all basic blocks remain unchanged. Let the program RCS be the RCS^{OUT} of the exit basic block, i.e., $RCS = RCS^{OUT}_{b_3}$. Results are reported in Table 37.3.

The LCS computation can be done in a similar fashion. gen_{b_k} is defined as the cache state describing the first executed memory block in every cache line for the basic block b_k. Taking the same assumption when defining gen_{b_k} for RCS computation that b_0 in Fig. 37.9 executes m_0 and then m_4, instead of only m_0, the

Table 37.4 LCS computation for the motivational example

	Basic block	LCS^{IN}	LCS^{OUT}
Initialization	b_3	$\{[\top, \top, \top, \top]\}$	$\{[m_4, \top, \top, \top]\}$
	b_2	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, \top, m_2, m_3]\}$
	b_1	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, m_1, m_2, m_3]\}$
	b_0	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$
Fixed-point	b_3	$\{[\top, \top, \top, \top]\}$	$\{[m_4, \top, \top, \top]\}$
	b_2	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, \top, m_2, m_3]\}$
	b_1	$\{[m_4, \top, \top, \top], [m_4, m_1, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3]\}$
	b_0	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$

first executed memory block in c_0 is m_0. Therefore, gen_{b_0} is $[m_0, \perp, \perp, \perp]$. LCS^{IN} of a basic block b_k is the LCS after b_k is executed and can be derived from

$$LCS_{b_k}^{IN} = \bigcup_{s \in successor(b_k)} LCS_s^{OUT}, \tag{37.46}$$

where $successor(b_k)$ is the set of all immediate successors of b_k. LCS^{OUT} of b_k is the LCS before b_k is executed with

$$LCS_{b_k}^{OUT} = \{\mathscr{T}(b_k, cs) | cs \in LCS_{b_k}^{IN}\}. \tag{37.47}$$

LCS computation also comprises two phases of initialization and fixed-point computation. The only difference is that the initialization phase starts from the exit basic block and ends in the entry basic block. Detailed results for the motivational example are reported in Table 37.4. Let the program LCS be the LCS^{OUT} of the entry basic block, i.e., $LCS = LCS_{b_0}^{OUT}$. It is noted that since the presented cache analysis technique is based on the fixed-point computation over the program CFG, it inherently handles loop structures in the CFG.

Conceptually, the program RCS is the set of all possible cache states after the program finishes execution by any execution path, and the program LCS is the set of all cache states, where each cache state contains memory blocks that may be firstly referenced after the program starts execution, for any execution path to follow. Both RCS and LCS could contain multiple cache states. Each pair with one cache state cs from the program RCS and one cache state cs' from the program LCS represents one possible execution path between the two consecutive executions. For any cache line c_i in a pair, if $cs[i]$ is equal to $cs'[i]$ and they are not equal to $\top$, then there is certainly a hit and thus WCET reduction. Whether there is a hit for a particular cache line can be determined by the function $\mathscr{H}$ defined as follows:
$\forall cs \in CS$, $cs' \in CS$ and $c_i \in CL$, where $i \in \{0, 1, \ldots, N_c - 1\}$,

$$\mathscr{H}(cs, cs', c_i) = \begin{cases} 1 : if\ cs[i] = cs'[i] \wedge cs[i] \neq \perp; \\ 0 : otherwise. \end{cases} \tag{37.48}$$

The number of hits can be counted with the function $\mathscr{HT}$ defined as $\forall cs \in CS$ and $cs' \in CS$,

$$\mathscr{HT}(cs, cs') = \sum_{i=0}^{N_c-1} \mathscr{H}(cs, cs', c_i). \tag{37.49}$$

The guaranteed number of hits among all possibilities is calculated as

$$\mathscr{G}(RCS, LCS) = \min_{cs \in RCS, cs' \in LCS} (\mathscr{HT}(cs, cs')). \tag{37.50}$$

Given that the main memory access time and the cache access time are respectively t_m and t_c, the guaranteed WCET reduction is computed as

$$\begin{aligned} \bar{E}^g &= \mathscr{G}(RCS, LCS) \times (t_m - t_c) \\ &\approx \mathscr{G}(RCS, LCS) \times t_m, \end{aligned} \tag{37.51}$$

where the approximation can be taken if $t_c \ll t_m$.

For the motivational example, there are two cache states in RCS ($RCS_{b_3}^{OUT}$) and two cache states in LCS ($LCS_{b_0}^{OUT}$). In total, there are four pairs, and the number of hits is calculated to be 3, 2, 2, and 2 with (37.49). Taking one of them as an example, $\mathscr{HT}([m_4, m_1, m_2, m_3], [m_0, m_1, m_2, m_3]) = 3$. Therefore, the guaranteed number of hits is 2 according to (37.50), no matter which path the program takes. From (37.51), the guaranteed WCET reduction is $2 \times (t_m - t_c)$, or approximately $2 \times t_m$, when $t_c \ll t_m$. It is noted that this result is obtained from the small example used for illustration. More WCET reduction for larger realistic programs can be expected.

Note that the direct-mapped cache (i.e., one-way set-associative cache) is assumed in Fig. 37.9. The presented technique can be adapted to handle set-associative cache. For example, considering fully associative cache, when computing $RCS_{b_3}^{OUT}$ from $RCS_{b_3}^{IN}$, the memory block m_4 can be loaded to any cache line, which gives $RCS_{b_3}^{OUT}$ five more cache states, i.e., $[m_0, m_4, m_2, m_3]$, $[m_0, m_1, m_4, m_3]$, $[m_0, \top, m_4, m_3]$, $[m_0, m_1, m_2, m_4]$, and $[m_0, \top, m_2, m_4]$. From this, it can be observed that the number of cache states in RCS and LCS is larger for set-associative cache, which means that the guaranteed WCET reduction could be smaller. Details can be found in [27]. Using the cache analysis technique presented in this section, together with standard WCET analysis approaches, the effective WCET of $C_i(2)$ and subsequent executions of C_i can be derived. Shorter WCET leads to smaller sampling period of the control system, which will be shown next.

37.4.2 Control Parameter Derivation

We explore the relationship between WCET results and control parameters of two example sampling schemes. S1 is the conventional memory-oblivious scheme and summarized as follows:

$$\begin{aligned} &C_1(1) \to C_2(1) \to C_3(1) \to C_1(2) \to C_2(2) \to \\ &C_3(2) \to C_1(3) \to C_2(3) \to C_3(3) \to \cdots . \end{aligned} \tag{37.52}$$

There is no cache reuse in S1 in the worst case, considering that different control applications typically have different instructions to execute. In other words, when $C_i(j)$ starts execution, all instructions of C_i need to be brought into the cache from the main memory. Therefore,

$$E_i^{wc}(1) = E_i^{wc}(2) = \cdots = E_i^{wc}, \tag{37.53}$$

where $E_i^{wc}(j)$ is the WCET of the jth execution for C_i. The WCET of the application C_i is denoted by E_i^{wc}, since all executions of the same application have equal WCET. Clearly, all control applications run with a uniform sampling period of

$$h = \sum_{i=1,2,3} E_i^{wc}. \tag{37.54}$$

Moreover, the sensor-to-actuator delay, which is defined to be the duration between measuring the system state $x(t)$ and applying the control input $u(t)$, is given by

$$d_i = E_i^{wc}. \tag{37.55}$$

It can be seen that a safe estimation of WCET, which can be done with standard static analysis techniques [37], is very important. If the actual execution time is longer than the computed WCET, the correct control input will not be ready when the actuation is supposed to occur. The consequence could be severe degradation of control performance. This is not acceptable especially for safety-critical control applications.

S2 is an example of memory-aware sampling order as shown in Fig. 37.8:

$$\begin{aligned} &C_1(1) \to C_1(2) \to C_1(3) \to C_2(1) \to C_2(2) \to \\ &C_2(3) \to C_3(1) \to C_3(2) \to C_3(3) \to \cdots . \end{aligned} \tag{37.56}$$

The effective WCET taking into account the cache reuse is denoted with $\bar{E}_i^{wc}(j)$. From the above discussion,

$\forall i \in \{1, 2, 3\}$,

$$\bar{E}_i^{wc}(1) = E_i^{wc}, \tag{37.57}$$

since there is no cache reuse for the first execution of every application $C_i(1)$. $\bar{E}_i^{wc}(2)$ and $\bar{E}_i^{wc}(3)$ are shorter than $\bar{E}_i^{wc}(1)$ due to cache reuse. The amounts of cache reuse are the same for $C_i(2)$ and $C_i(3)$ in the worst case. Denoting the guaranteed WCET reduction as $\bar{E}_i^g$,

$\forall i \in \{1, 2, 3\}$,

$$\bar{E}_i^{wc}(2) = \bar{E}_i^{wc}(3) = \bar{E}_i^{wc}(1) - \bar{E}_i^g. \tag{37.58}$$

From these varying WCETs, the sampling periods of all three applications can be calculated. Taking C_1 as an example, there are three sampling periods $h_1(1)$, $h_1(2)$, and $h_1(3)$, which repeat themselves periodically:

$$h_1(1) = \bar{E}_1^{wc}(1), \quad h_1(2) = \bar{E}_1^{wc}(2), \quad h_1(3) = \bar{E}_1^{wc}(3) + \Delta, \tag{37.59}$$

where Δ is computed as

$$\Delta = \sum_{i=2,3} \sum_{j=1,2,3} \bar{E}_i^{wc}(j). \tag{37.60}$$

Similar derivation can be done for C_2 and C_3. The average sampling period of an application h_{avg} is

$$h_{\text{avg}} = \frac{\sum_{i=1,2,3} \sum_{j=1,2,3} \bar{E}_i^{wc}(j)}{3} < h. \tag{37.61}$$

According to (37.57) and (37.58),

$$h_{\text{avg}} < \frac{\sum_{i=1,2,3} 3 \times E_i^{wc}}{3}. \tag{37.62}$$

From (37.54),

$$h_{\text{avg}} < h. \tag{37.63}$$

Moreover, the corresponding sensor-to-actuator delay $d_i(j)$ also varies with cache reuse as

$\forall i \in \{1, 2, 3\}$,

$$d_i(1) = h_i(1) = \bar{E}_i^{wc}(1), \quad d_i(2) = h_i(2) = \bar{E}_i^{wc}(2), \quad d_i(3) = \bar{E}_i^{wc}(3). \tag{37.64}$$

As all control parameters have been derived, it can be observed that the sampling period $h_i(j)$ of a control application is nonuniform for the memory-aware scheme. The average sampling period of S2 is shorter than the uniform sampling period of S1 as shown in (37.61), due to WCET reduction resulting from cache reuse. The sensor-to-actuator delay $d_i(j)$ varies as shown in (37.64). The next task is to develop a controller design method to exploit shortened nonuniform sampling periods and achieve better control performance. For the uniform sampling scheme, the sensor-to-actuator delay d_i is shorter than the sampling period h. Therefore, the technique reported in Sect. 37.2 is used. Details of the controller design technique considering the nonuniform sampling are reported in the next section.

37.4.3 Case Study

Here a commonly used processing unit, equipped with a processor, on-chip memory as cache and flash as main memory is considered, shown in Fig. 37.1 More about the flash memory has been discussed in ▶ Chap. 14, "Emerging and Nonvolatile Memory". As a case study, three control applications are considered. C_1 is position control of a servo motor. C_2 is speed control of a DC motor. C_3 is control of an electronic wedge brake system. All three control applications run on the same processor. The processor clock frequency is 20 MHz. The cache is set to have 128 cache lines and each cache line is 16 bytes. When there is a cache hit, it takes 1 clock cycle to fetch the instruction, and when there is a cache miss, it takes 100 clock cycles. WCET results are reported in Table 37.5. Sampling periods of the two sampling orders S1 and S2 are shown in Table 37.6. Control performances of three applications under S1 and S2 are presented in Table 37.7, where the settling time is taken as the performance metric. As an example, the system output responses of C_1 under both S1 and S2 are presented in Fig. 37.10. The control task considered is to change the system output (i.e., the angular position of the servo motor) from 0 to 0.3 rad. From the above experimental results, it can be clearly seen that the

Table 37.5 WCET results with and without cache reuse for all three control applications

Application	WCET without cache reuse	WCET with cache reuse	Reduction percentage
C_1	907.55 μs	452.15 μs	50.18%
C_2	645.25 μs	175.00 μs	72.88%
C_3	749.15 μs	234.35 μs	68.72%

Table 37.6 Comparison of sampling periods between S1 and S2 for all three control applications. The reduction percentage is computed according to the average sampling period

Application	Sampling periods in S1	Sampling periods in S2	Reduction percentage
C_1	2302 μs	452 μs – 452 μs – 3121 μs	42%
C_2	2302 μs	175 μs – 175 μs – 3675 μs	42%
C_3	2302 μs	234 μs – 234 μs – 3557 μs	42%

Table 37.7 Control performances for all three applications under S1 and S2

Application	C_1	C_2	C_3
Settling time for S1	31.2 ms	26.8 ms	25.2 ms
Settling time for S2	21.5 ms	21.1 ms	20.4 ms
Control performance improvement of S2 compared to S1	31.1%	21.3%	19.0%

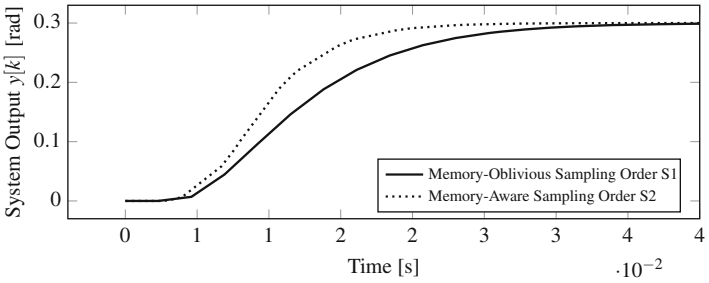

Fig. 37.10 Control system output of $\mathscr{C}_1$ under S1 and S2

memory-aware sampling order reduces the WCETs and sampling periods. With the controller design method tailored for nonuniform sampling, control performances are significantly improved.

37.5 Computation-Aware Control/Architecture Codesign

In this section, we show how to use a multirate controller to reduce the processor utilization of a control application, while still fulfilling the control performance requirement and system constraints. More information about the application-specific processors can be found in ▸ Chap. 12, "Application-Specific Processors".

37.5.1 Time-Triggered Operating System

As an example, ERCOSek with the OSEK/VDX standard [1] is considered, which specifies the basic properties of an OS to be used in the automotive domain. In general, as an OSEK/VDX OS, ERCOSek supports preemptive fixed-priority scheduling. That is, priorities are assigned to applications, and at any point in time, the task with the highest priority among all active ones is executed. On ERCOSek, tasks can be triggered by events (e.g., interrupts, alarms, etc.) or by time. In the time-triggered scheme, each application gets released and is allowed to access the processor periodically. There are various periods of release times and each application is assigned one. Every time an application is released, its task gets the chance to be executed. A time table containing all the periodic release times within

Table 37.8 Example of an ERCOSek time table

Time	Release
0 ms	Applications with periods of 2 ms/5 ms/10 ms
2 ms	Applications with the period of 2 ms
4 ms	Applications with the period of 2 ms
5 ms	Applications with the period of 5 ms
6 ms	Applications with the period of 2 ms
8 ms	Applications with the period of 2 ms
10 ms	**Repeat actions at 0 ms**

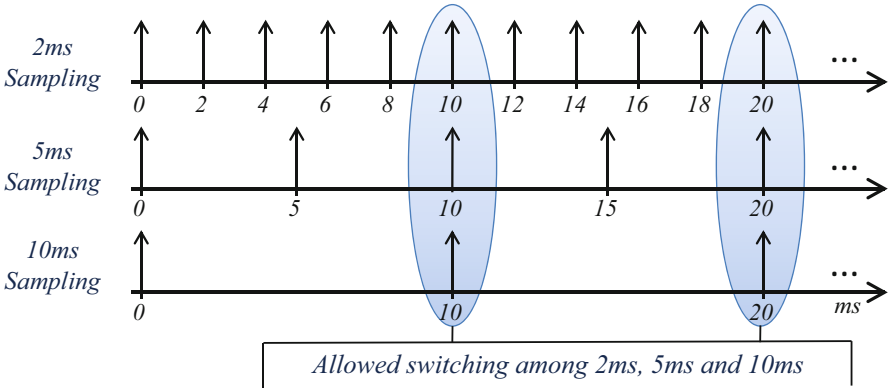

Fig. 37.11 Allowed switching instants among multiple sampling periods

the alleged hyperperiod (i.e., the minimum common multiple of all periods) needs to be configured. An example with a set of three periods 2, 5, and 10 ms is illustrated in Table 37.8. The hyperperiod is equal to 10 ms and the time table repeats itself every 10 ms by resetting the timer. Independent of the triggering mode (i.e., be it event or time triggered), the assigned priority will still determine the execution order of tasks. In the time-triggered scheme, a higher priority is typically assigned to the application released with a shorter period, since this generally results in a more efficient use of the processor.

Assuming the set of available periods restricted by ERCOSek to be ϕ, control applications have to be sampled with one period or a combination of multiple periods from ϕ. In the latter case, switching between two sampling periods can only occur at the common multiplier of them, as illustrated in Fig. 37.11, considering three sampling periods 2, 5, and 10 ms. Often, the optimal sampling period for a control application does not belong to the set ϕ. The simple and straightforward method used in practice is to select the largest sampling period in ϕ that is smaller than the optimal one. This results in a higher processor load, which is another important design aspect. Denoting E_i^{wc} to be the WCET of a control application C_i, if the uniform sampling period is T, the processor load for C_i is

$$L_i = \frac{E_i^{wc}}{T}. \tag{37.65}$$

The upper bound on the load of any processor is 1. Considering a single processor p,

$$\sum_{\{i|C_i \text{ runs on } p\}} L_i \leq 1. \tag{37.66}$$

Clearly, increasing the sampling period of a control application decreases its processor load and thus potentially enables more applications to be integrated on the processor.

37.5.2 Multirate Closed-Loop Dynamics

We consider a multirate controller switching between multiple sampling periods in ϕ, toward achieving an average sampling period close to the optimal one. The cyclic sequence of sampling periods for a control application defines a schedule S:

$$S = \{T_1, T_2, T_3, \ldots, T_N\}, \tag{37.67}$$

where $\forall j \in \{1, 2, \ldots, N\},\ T_j \in \phi$. It implies the sequence of sampling periods as

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_N \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_N \rightarrow repeat$$

Following the assumption in (37.65) that the WCET of C_i is E_i^{wc}, the processor load for C_i over S is

$$L_i = \frac{NE_i^{wc}}{\sum_{j=1}^{N} T_j}. \tag{37.68}$$

Dictated by the schedule S, N systems switch cyclically in a deterministic fashion. When the sampling period $t_{k+1} - t_k = T_{k,j}$, the dynamics is

$$x[k+1] = A_d(T_{k,j})x[k] + B_d(T_{k,j})K_j x[k] + B_d(T_{k,j})F_j r. \tag{37.69}$$

The controller design needs to be performance oriented, and the key is to compute the feedback gain K_j for each system with pole placement, based on which the static feedforward gain F_j can be derived with (37.10).

Referring to Fig. 37.12, after the first sampling interval of a switching cycle,

$$x[k+1] = A_d(T_{k,1})x[k] + B_d(T_{k,1})\hat{K}_1 x[k] + B_d(T_{k,1})F_1 r. \tag{37.70}$$

It is noted that K_1 is the feedback gain based on the most recent system state $x[k]$ and used to compute the control input. $\hat{K}_1$ is the equivalent feedback gain based on the starting system state $x[k]$ of a switching cycle. In this case that only one

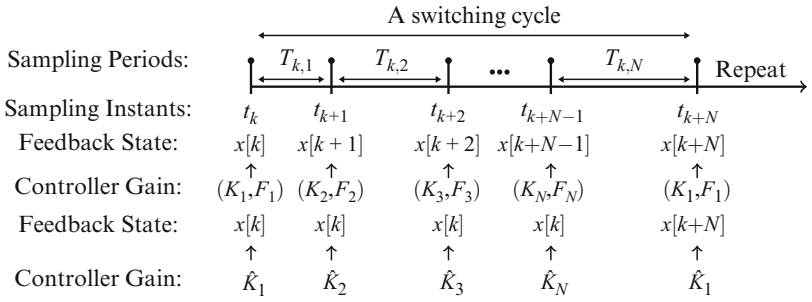

Fig. 37.12 Cyclically switched linear systems

sampling period is considered, $\hat{K}_1 = K_1$. The feedforward gain F_1, which is related to K_1, is also based on the most recent system state and used to compute the control input. The closed-loop system matrix is denoted as $A_{cl,1}$ and

$$A_{cl,1} = A_d(T_{k,1}) + B_d(T_{k,1})\hat{K}_1. \tag{37.71}$$

$\hat{K}_1$ can be designed by pole placement. Poles to place are eigenvalues of $A_{cl,1}$. F_1 is computed as per (37.10).

After the second sampling interval,

$$x[k+2] = A_d(T_{k,2})x[k+1] + B_d(T_{k,2})K_2x[k+1] + B_d(T_{k,2})F_2r. \tag{37.72}$$

To consider the overall dynamics of the first two sampling periods, the relation between $x[k+2]$ and $x[k]$ can be derived as

$$\begin{aligned} x[k+2] = {} & A_d(T_{k,2})A_{cl,1}x[k] + B_d(T_{k,2})K_2A_{cl,1}x[k] \\ & + (A_d(T_{k,2}) + B_d(T_{k,2})K_2)B_d(T_{k,1})F_1r + B_d(T_{k,2})F_2r. \end{aligned} \tag{37.73}$$

Let

$$\hat{K}_2 = K_2A_{cl,1}, \tag{37.74}$$

and (37.73) becomes

$$\begin{aligned} x[k+2] = {} & A_d(T_{k,2})A_{cl,1}x[k] + B_d(T_{k,2})\hat{K}_2x[k] \\ & + (A_d(T_{k,2}) + B_d(T_{k,2})K_2)B_d(T_{k,1})F_1r + B_d(T_{k,2})F_2r. \end{aligned} \tag{37.75}$$

Similar to (37.71),

$$A_{cl,2} = A_d(T_{k,2})A_{cl,1} + B_d(T_{k,2})\hat{K}_2. \tag{37.76}$$

It is noted that (37.75) has the same form as (37.70). $\hat{K}_2$ can be designed by pole placement and K_2 is derived with (37.74), as long as $A_{cl,1}$ is non-singular. Poles to place are eigenvalues of $A_{cl,2}$. F_2 is computed as per (37.10). Continuing the above analysis,

$\forall j \in \{1, 2, \ldots, N\}$

$$A_{cl,j} = A_d(T_{k,j})A_{cl,j-1} + B_d(T_{k,j})\hat{K}_j, \tag{37.77}$$

and $A_{cl,0} = \mathbf{I}$ can be defined. $\hat{K}_j$ can be designed by pole placement. Poles to place are eigenvalues of $A_{cl,j}$. As long as $A_{cl,j-1}$ is non-singular, K_j is derived by

$$K_j = \hat{K}_j A_{cl,j-1}^{-1}. \tag{37.78}$$

F_j is computed as per (37.10).

Here the sensor-to-actuator delay is approximately equal to the WCET of the executed control program. Since the control law is computed during the design phase, such a control program generally has a short WCET. The sensor-to-actuator delay is often negligible compared to the sampling periods given by the OS. In general, when the sensor-to-actuator delay of a control task is large compared to the sampling periods (e.g., in the memory-aware controller design of Sect. 37.4, where the sampling periods are directly constrained by WCETs), our proposed controller design technique can be extended to consider the delayed control input with a number of methods reported in the literature [24].

An optimization problem for the pole placement can be formulated as presented in Sect. 37.2.2.2. The number of dimensions in the decision space is nN – the number of states of the application multiplied by the number of sampling periods in the schedule. The optimization objective is the settling time. Absolute values of all poles have to be less than unity to ensure system stability and larger than 0 to make all $A_{cl,j}$ non-singular.

Optimization strategies for design space exploration have been discussed in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration". In this section, to solve one optimization problem, the PSO algorithm is run multiple times with the same number of particles, and we do not set the limit on the number of iterations. If the objective value variation of the solution points from these runs exceeds a certain threshold (e.g., 1%), the number of particles is increased. Considering the stochastic nature of PSO, it is very likely that the optimal point has been found when multiple runs generate similar objective values. It is noted that if the number of sampling periods in the schedule is very large, which makes the number of dimensions in the decision space very large, this method aiming to ensure optimality can be computationally expensive. In this case, the number of particles and iterations has to be limited, resulting in a compromise in optimality.

Table 37.9 Settling times of three schedules

Schedule	Settling time [ms]	Requirement
$S1 = \{5\,\text{ms}\}$	253.69	Violated
$S2 = \{2\,\text{ms}\}$	110.44	Satisfied
$S3 = \{2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 5\,\text{ms}\}$	128.6 ms	Satisfied

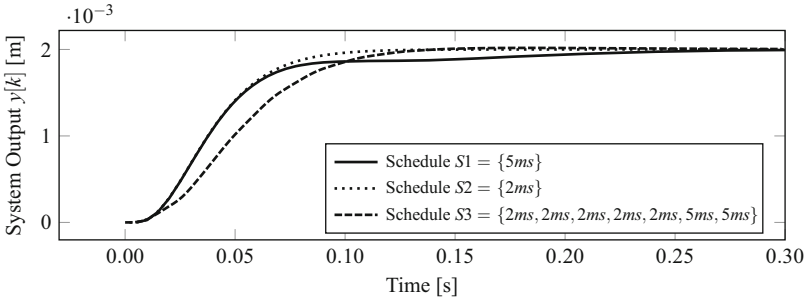

Fig. 37.13 System outputs of different schedules

37.5.3 Case Study

The presented multirate controller design technique is evaluated with an Electro-Mechanical Brake (EMB) system used in automobiles. It can be modeled as (37.1) with five system states. The control input is the voltage output of the onboard battery and thus cannot exceed 12 V. Different controllers require different battery voltage output profiles, and only those that respect the input constraint are possible to implement. The constraint on the settling time is 150 ms. In the optimization, the settling time is still treated as the objective to minimize and check the optimal solution against this requirement. The WCET of the control program is 0.2 ms. The control task is to change the system output (i.e., the position of the lever) from 0 to 2 mm. The set of available sampling periods offered by ERCOSek is

$$\phi = \{1\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 10\,\text{ms}, 20\,\text{ms}, 50\,\text{ms}, 100\,\text{ms}, 200\,\text{ms}, 500\,\text{ms}, 1\,\text{s}\}. \tag{37.79}$$

As shown in Table 37.9 and Fig. 37.13, the schedule $S1 = \{5ms\}$ cannot meet the settling time requirement. The largest sampling period smaller than 5 ms in ϕ is 2 ms. The schedule $S2 = \{2\,\text{ms}\}$ is able to fulfill all the requirements. According to (37.65), the processor load of $S2$ is 0.1. Then a schedule switching between 2 ms and 5 ms is considered, $S3 = \{2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 5\,\text{ms}\}$. This sequence of sampling periods satisfies the OS requirement. The multirate controller is designed as discussed earlier in this section. The WCET (0.2 ms) is much shorter than the sampling periods (2 ms, 5 ms), and thus we neglect the sensor-to-actuator

delay. $S3$ has a slightly longer settling time than $S2$, but still fulfills the requirement. According to (37.68), the processor load is 0.07, achieving a 30% reduction compared to $S2$.

37.6 Conclusion

In this chapter, a basic introduction into the subject of control/architecture codesign in the context of cyber-physical systems is provided. The control/architecture codesign is an emerging field of research, where the design of control parameters and embedded platform parameters are integrated in a holistic approach to reduce conservativeness and achieve more efficient design of embedded control systems. As the size and the complexity of the cyber-physical systems increase, resource-efficient design has become one of the most important aspects in this context. In this chapter, the motivation is firstly explained, and some basic concepts of the control/architecture codesign are introduced. In addition, a brief summary on the type of resources that can be considered in the codesign approaches is provided. Then three examples of state-of-art codesign approaches, targeting respectively at communication-aware design, memory-aware design, and computation-aware design, are used to illustrate the basic thinking behind the control/architecture codesign. In Sect. 37.3, a cooptimization framework is explained to codesign control and platform parameters by solving a constraint-based multi-objective optimization problem. This framework considers two objectives, namely, the resource utilization and the overall control performance, and generates a Pareto front depicting the trade-off options between the two objectives. In Sect. 37.4, how to exploit the instruction cache reuse in a memory-aware sampling order to improve the control performance is shown. Cache analysis is used to compute the guaranteed WCET reduction between two consecutive executions of one control program. Control parameters are derived based on the WCET results. The controller design is tailored for the nonuniform sampling scheme. In Sect. 37.5, the OS constraint that only a limited set of sampling periods are provided is considered. It is shown how a multirate controller is used to reduce the processor utilization of a control application, while still fulfilling the control performance requirement and system constraints. The control/architecture codesign is, of course, a relatively new and open research field, and thus the state-of-art approaches are certainly not limited to the ones shown in this chapter. There are also some other research directions in this context that can be explored. For example, power consumption is quite an important design factor, and thus power-aware codesign methods could potentially lead to more power-efficient designs. Furthermore, safety and fault tolerance are also important factors in cyber-physical systems which can also be considered in codesign methods. In addition, the three approaches shown in this chapter address individually a single resource. If the complexity of the problem due to many design dimensions can be tackled, it would be interesting to try to address simultaneous two or more resources in the codesign and thus offer an even greater freedom for design trade-offs.

References

1. OSEK/VDX operating system specification 2.2.3 (2005)
2. 664P7-1 aircraft data network, part 7, avionics full-duplex switched Ethernet network (2009)
3. The FlexRay communications system protocol specification, Version 3.0.1 (2010)
4. LIN specification package revision 2.2A (2010)
5. MOST specification rev. 3.0 E2 (2010)
6. AS6802 (2011) Time-triggered Ethernet
7. Infineon Product Brief XC2300B – Series (Accessed 12 May 2016). http://www.infineon.com/dgdl/Pb_XC2300B.pdf?fileId=db3a30432a7fedfc012ab3c3d7863706
8. Ackermann J, Utkin VI (1994) Sliding mode control design based on Ackermann's formula. In: Proceedings of the 33rd IEEE conference on decision and control, vol 4, Lake Buena Vista, pp 3622–3627. doi:10.1109/CDC.1994.411715
9. Andalam S, Sinha R, Roop P, Girault A, Reineke J (2013) Precise timing analysis for direct-mapped caches. In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), Austin, pp 1–10. doi:10.1145/2463209.2488917
10. Astrom KJ, Murray RM (2008) Feedback systems: an introduction for scientists and engineers. Princeton University Press, Princeton
11. Batcher KW, Walker RA (2008) Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In: 2008 Design, automation and test in Europe, Munich, pp 260–263. doi:10.1109/DATE.2008.4484893
12. Bhave AY, Krogh BH (2008) Performance bounds on state-feedback controllers with network delay. In: 47th IEEE conference on decision and control, CDC 2008, Cancun, pp 4608–4613. doi:10.1109/CDC.2008.4739330
13. Bosch (1991) CAN Specification version 2.0. Stuttgart, Bosch
14. Castane R, Marti P, Velasco M, Cervin A, Henriksson D (2006) Resource management for control tasks based on the transient dynamics of closed-loop systems. In: 18th Euromicro conference on real-time systems (ECRTS'06), Dresden, pp 10, 182. doi:10.1109/ECRTS.2006.24
15. Cervin A, Velasco M, Marti P, Camacho A (2009) Optimal on-line sampling period assignment. Research report, Lund University and Technical University of Catalonia
16. Chang W, Chakraborty S (2016) Resource-aware automotive control systems design: a cyber-physical systems approach. Found Trends© Electron Design Autom 10(4):249–369. http://dx.doi.org/10.1561/1000000045
17. Charette RN (2009) This car runs on code. IEEE Spectrum. http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code
18. eCos. http://ecos.sourceware.org
19. Feiler PH (2003) Real-time application development with OSEK: a review of the OSEK standards. Technical report, Carnegie Mellon University
20. Gaid MEMB, Cela A, Hamam Y (2006) Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system. IEEE Trans Control Syst Technol 14(4):776–787. doi:10.1109/TCST.2006.872504
21. Gaid MEMB, Cela A, Hamam Y, Ionete C (2006) Optimal scheduling of control tasks with state feedback resource allocation. In: 2006 American control conference, Minneapolis, pp 310–315. doi:10.1109/ACC.2006.1655373
22. Gloy N, Smith MD (1999) Procedure placement using temporal-ordering information. ACM Trans Program Lang Syst 21(5):977–1027. doi:10.1145/330249.330254
23. Goswami D, Lukasiewycz M, Schneider R, Chakraborty S (2012) Time-triggered implementations of mixed-criticality automotive software. In: 2012 Design, automation test in Europe conference exhibition (DATE), Dresden, pp 1227–1232. doi:10.1109/DATE.2012.6176680
24. Goswami D, Schneider R, Chakraborty S (2014) Relaxing signal delay constraints in distributed embedded controllers. IEEE Trans Control Syst Technol 22(6):2337–2345. doi:10.1109/TCST.2014.2301795

25. Henriksson D, Cervin A (2005) Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In: Proceedings of the 44th IEEE conference on decision and control, Seville, pp 4469–4474. doi:10.1109/CDC.2005.1582866
26. Kalamationos J, Kaeli DR (1998) Temporal-based procedure reordering for improved instruction cache performance. In: Proceedings of fourth international symposium on high-performance computer architecture, Las Vegas, pp 244–253. doi:10.1109/HPCA.1998.650563
27. Kleinsorge JC, Falk H, Marwedel P (2011) A synergetic approach to accurate analysis of cache-related preemption delay. In: 2011 Proceedings of the international conference on embedded software (EMSOFT), Taipei, pp 329–338. doi:10.1145/2038642.2038693
28. Liu X, Chen X, Kong F (2015) Utilization control and optimization of real-time embedded systems. Found Trends© Electron Design Autom 9(3):211–307. http://dx.doi.org/10.1561/1000000042
29. Lukasiewycz M, GlaßM, Teich J, Milbredt P (2009) Flexray schedule optimization of the static segment. In: Proceedings of the 7th IEEE/ACM international conference on hardware/-software codesign and system synthesis, CODES+ISSS'09. ACM, New York, pp 363–372. doi:10.1145/1629435.1629485
30. Marti P, Lin C, Brandt SA, Velasco M, Fuertes JM (2004) Optimal state feedback based resource allocation for resource-constrained control tasks. In: Proceedings of 25th IEEE international on real-time systems symposium, Lisbon, pp 161–172. doi:10.1109/REAL.2004.39
31. Martí P, Lin C, Brandt SA, Velasco M, Fuertes JM (2009) Draco: efficient resource management for resource-constrained control tasks. IEEE Trans Comput 58(1):90–105. doi:10.1109/TC.2008.136
32. Pettis K, Hansen RC (1990) Profile guided code positioning. In: Proceedings of the ACM SIGPLAN 1990 conference on programming language design and implementation, PLDI'90. ACM, New York, pp 16–27. doi:10.1145/93542.93550
33. Pigan R, Metter M (2008) Automating with PROFINET, 2nd edn. Publicis Publishing, Erlangen
34. Samii S, Cervin A, Eles P, Peng Z (2009) Integrated scheduling and synthesis of control applications on distributed embedded systems. In: 2009 Design, automation test in Europe conference exhibition, Nice, pp 57–62. doi:10.1109/DATE.2009.5090633
35. Schneider R, Goswami D, Zafar S, Lukasiewycz M, Chakraborty S (2011) Constraint-driven synthesis and tool-support for flexray-based automotive control systems. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS'11. ACM, New York, pp 139–148. doi:10.1145/2039370.2039394
36. Sedighizadeh D, Masehian E (2009) Particle swarm optimization methods, taxonomy and applications. Int J Comput Theory Eng 1(4):486–502
37. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):36:1–36:53. doi:10.1145/1347375.1347389
38. Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 28(7):966–978. doi:10.1109/T-CAD.2009.2013287
39. Zeng H, Natale MD, Ghosal A, Sangiovanni-Vincentelli A (2011) Schedule optimization of time-triggered systems communicating over the flexray static segment. IEEE Trans Ind Inf 7(1):1–17. doi:10.1109/TII.2010.2089465