From Timing Budgets to WCETs: Robust SIL- and BSW-Aware Clustering and Allocation for Iterative Automotive Software Development

Tobias Denzinger
CARIAD SE, Ingolstadt, Germany

Matthias Becker
KTH Royal Institute of Technology Stockholm, Sweden

Peter Ulbrich
Technische Universität Dortmund, Germany

Abstract

Automotive ECUs integrate thousands of AUTOSAR runnables, substantial Basic Software (BSW), and heterogeneous multicore hardware. In iterative software-defined vehicle development, engineers must repeatedly revisit designs while maintaining stable runnable clustering and core allocations, which are expensive structural decisions. Beyond timing, Safety Integrity Levels (SILs), BSW overheads, and per-core memory strongly constrain these decisions, yet are rarely modeled jointly. This paper addresses these challenges through a chain-based analysis model that treats SIL constraints and BSW costs as first-class citizens, as well as an integrated toolchain that constructs job-level data-age constraints, forms SIL-compliant clusters, synthesizes multirate tasks, and maps application and BSW tasks to heterogeneous multicore platforms while checking timing and memory feasibility.

A case study based on a real-world motion/drive controller from our industrial partner is described, which serves as the basis for our evaluation. The evaluations are conducted using synthetic systems that reflect the characteristics of the case study. Across 13,825 synthesized systems, SIL/BSW-aware clustering substantially reduces pessimism in analysis. In the industrial configuration, our approach yields a 7% decrease in utilization, demonstrating its practical value. A refinement study, which progressively replaces early budget assumptions with WCET samples, indicates that SIL/BSW-aware clustering preserves structural decisions better than less-informed variants under the same resampling setup.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded software; Software and its engineering → Software safety

Keywords and phrases cause-effect chains, end-to-end latency constraints, automotive software, SIL, ASIL, AUTOSAR BSW, WCET refinement, timing analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.19

Supplementary Material *Text (Case Study)*: `https://arxiv.org/abs/2605.04837` [15]
Software (Benchmark Generator): `https://doi.org/10.17877/TUDODATA-2026-MOR12ARE` [14]

Funding *Matthias Becker*: The work was partially supported by the Swedish Research Council (VR) under the project nr. 2023-04773, and by Sweden's Innovation Agency via the NFFP8 project 2024-01267: PARTI.

Acknowledgements We thank our industrial partners in the CARIAD SE, Germany, for its collaboration and constructive feedback, which improved the quality and practical relevance of this project. Generative AI tools were used to support language polishing and LaTeX formatting. The authors reviewed and edited all affected text and metadata and take full responsibility for the content.

1 Introduction

Automotive E/E systems continue to grow in complexity, while the industry is moving towards software-defined vehicles (SDV) with mixed-criticality functionality and frequent integration and update cycles. In such settings, architectural decisions are not made once and for all, but are revisited repeatedly as functions evolve, suppliers deliver incremental changes, and timing and memory characteristics are refined. Our goal is to keep expensive architectural decisions stable while refining budgets and incorporating feedback from timing analysis and measurement. A dominant architectural pattern is *cause–and–effect chains* (i. e., data flow from sensors to actuators via tasks that may run at different rates [35]), for which end-to-end timing must be met to uphold data age and quality constraints [21]. While chain-aware timing analysis and allocation are well studied, our collaboration with industry indicates that two additional drivers are often decisive for feasibility and cost, yet are underrepresented in chain-centric optimization: (i) safety integrity constraints that restrict colocation and resource sharing, and (ii) Basic Software (BSW) costs, including both task-local overheads (e. g., per-task service extensions) and global BSW tasks (e. g., providing communication and diagnosis) competing for CPU and memory. Crucially, these costs are strongly coupled to structural design choices such as how software components are clustered into tasks, how chains are decomposed, and how tasks and BSW are co-allocated to CPU cores and memory domains. This coupling makes structural changes costly: reorganizing clusters, chain partitions, and core assignments affects integration, configuration, safety arguments, and deployment effort. In contrast, timing budgets and WCET estimates are commonly refined as more implementation evidence becomes available. We therefore target a recurring architectural-analysis setting in which tighter WCET estimates progressively replace conservative budgets (required for early safety assurance). The objective is not to guarantee invariant architectures, but to reduce avoidable remapping by making SIL and BSW constraints explicit during clustering and allocation.

1.1 Problem Statement

Given (1) runnables and their chain structure with end-to-end/data-age constraints, (2) SIL labels and separation requirements for application and BSW, (3) a target multicore platform with core classes and per-core memory limits, and (4) an empirically grounded BSW cost model distinguishing global stacks and task-local overheads, compute (a) a SIL-compliant clustering of runnables into application tasks (including multirate task construction), (b) a placement of application tasks and required BSW tasks to cores/memory, and (c) a sound feasibility assessment w.r.t. timing, memory, and safety constraints under the stated modeling assumptions. We strive to provide engineers with a single integrated architectural timing-analysis toolchain that can be applied throughout iterative SDV development: given an evolving architectural model, it jointly constructs tasks, accounts for BSW extensions and global BSW tasks, and checks timing and memory feasibility under SIL and data-age constraints. A key practical challenge is that the global BSW task set is largely unknown early on; we therefore approximate its impact via a conservative envelope so feasibility analysis remains possible from the first iterations. Thus, our goal is to reduce avoidable pessimism in timing analysis by exploiting SIL/BSW knowledge during clustering and mapping, and additionally to minimize unnecessary changes to expensive structural decisions (clusters and core assignments) as timing inputs are refined from conservative budgets to WCETs.

1.2 Contribution and Outline

1. **Industrial case study and benchmark synthesis.** We empirically characterize the interplay of SIL constraints, BSW, and memory in a production-grade motion/drive controller and derive profiles that enable realistic synthetic benchmark generation. The companion case study [15] and the Driverator benchmark generator [14] are provided as supplemental material.
2. **An analysis model with SIL/BSW as first-class citizens.** We formalize a chain-based model that integrates (i) SIL constraints spanning application and BSW, (ii) explicit BSW accounting that distinguishes task-local overheads from global platform services, and (iii) per-core memory limits. To reflect the mixed-criticality practice observed with our partners, we also allow a mode-dependent chain structure. Our model integrates feasibility-preserving estimates for parameters that are still unknown but supports sound analysis when worst-case bounds are available.
3. **Integrated analysis/optimization toolchain and empirical validation.** We implement a full toolchain that combines job-level dependency construction for data-age constraints, SIL-constrained clustering, multirate task construction, and BSW- and memory-aware mapping to heterogeneous multicore platforms. Using thousands of synthesized systems grounded in the case study profiles, we quantify the impact of BSW overheads and show that SIL/BSW-aware clustering can substantially reduce analysis pessimism compared to less informed variants; additionally, we report that resulting clusterings/mappings are more stable than less informed variants when conservative budgets are refined towards WCET inputs.

The rest of the paper is organized as follows: Section 2 reviews related work; Section 3 defines the system model; Section 4 presents our heuristic framework; Section 5 evaluates it on large generated systems; and Section 6 concludes.

2 Related Work

End-to-end semantics in automotive. The mapping and scheduling of automotive applications have been extensively studied. Kramer et al. [35] characterize an engine management system and highlight the complexity of cause-effect chains (CECs) [1]. Feiertag et al. [21] classify end-to-end delay semantics for such chains, and the WATERS Industrial Challenge 2017 [28] examines communication semantics on multicore platforms.

Communication and precedence modeling. Timing-analysis tools provide end-to-end latency checks for multi-rate CECs [29, 30, 48], typically assuming detailed system information. The Logical Execution Time (LET) paradigm decouples communication from execution and eases integration on multicore platforms [27], but can increase end-to-end latencies compared to implicit communication [6, 39, 47]. Ernst et al. [20] and Gemlau et al. [25] discuss extensions of LET to distributed settings, and several optimization strategies mitigate LET-related latency penalties [37, 50, 26]. Complementarily, precedence-based models capture multi-rate dependencies explicitly: Forget et al. [22] introduce a synchronous data-flow language and propose fixed-priority scheduling without synchronization to realize extended precedence constraints [23]. Becker et al. [5, 7] compute end-to-end latencies via job-level dependencies (JLDs), which Klaus et al. integrate into table-driven scheduling [34] and later into dynamic scheduling via system transformations [33]. Dietrich et al. [16] derive global control flows for whole-system WCET analysis, extended to fuzzing-based dynamic whole-system timing analysis by Berger et al. [8].

Runnable-to-task construction and clustering. Several works examine how the runnable/-task structure affects schedulability and analysis. Monot et al. [43] bundle related runnables into sequencer tasks on multicore platforms. Khenfri et al. [32] investigate runnable-to-task allocation based on the multiframe task model [42], and Becker et al. [4] similarly merge tasks under JLD constraints. These approaches motivate treating *task construction* as a first-class design dimension rather than a fixed input.

Allocation and design flexibility. A broad body of work addresses constrained task-to-processor allocation, often coupled with priority assignment and schedulability analysis [12, 13, 38]. In the automotive context, Zheng et al. [51] and follow-up work (e. g., [52, 11]) optimize allocation/priority choices under end-to-end constraints for distributed architectures. Beyond optimizing a single design point, flexibility- and change-aware allocation has been studied using scenario-based evaluation and robustness objectives, aiming to accommodate future changes with minimal alterations to an existing mapping [3, 19]. Tool-supported generation flows have also been explored for time-triggered multicore systems [24]. Bhat et al. [9] further demonstrate allocation techniques under additional practical constraints in embedded automotive systems. Exact formulations based on (mixed) integer linear programming have also been proposed for constrained mapping and priority assignment problems [17, 44, 49, 51]; however, these approaches typically assume a *given* task set and cost model.

Mixed criticality and safety constraints. Functional safety standards such as ISO 26262 [31] mandate freedom from interference across criticality levels, underscoring the need for SIL-aware allocation and scheduling. Mixed-criticality scheduling [10] represents a natural candidate to combine strong partitioning with efficient resource sharing and is, for example, part of Eclipse's SDV framework. Recent work also considers *precedence constraints*, for example, Socci et al. [46, 45] for precedence-constrained mixed-critical jobs on multiprocessors and Medina et al. [41, 40] for (multi-periodic) mixed-criticality DAG models with mode-dependent static schedules. In principle, such precedence/DAG formalisms could encode parts of SIL/BSW-induced ordering or mode-dependent functionality.

State-of-the-art Gap. Prior work advances cause–effect-chain semantics, precedence modeling, clustering, and allocation. Yet these lines are rarely integrated with (i) SIL-constrained clustering and (ii) an explicit BSW cost model that distinguishes task-local overheads from global platform services and captures their CPU/memory impact. As a result, current approaches either abstract away from BSW and safety-induced separation, which can introduce unnecessary pessimism and obscure the true feasibility and cost trade-offs that practitioners face. Moreover, most allocation techniques assume fixed constraints and task sets, whereas in our setting task construction (i. e., runnable mapping), SIL feasibility, and BSW overheads are coupled: clustering decisions change the task set and directly affect BSW overhead and memory aggregation. This coupling motivates our integrated chain-aware flow that treats SIL and BSW as first-class citizens while remaining compatible with iterative refinement in SDV development.

3 System Model

This section introduces our system model, focusing on information typically available in early design and its relation to implementation-level artifacts.

Safety Integrity Levels. We use the ASIL levels of ISO 26262 [31], ranging from *QM* (*Quality Management*, i.e., functionality without an ASIL assignment) to ASIL D, the highest integrity level considered here. We abstract ASIL as a safety integrity level (SIL) and denote the finite set of supported levels by $\mathcal{A}$.

Application Layer. **Runnables** are the elementary units of execution; we consider a set $\mathcal{R} = \{r_1, \ldots, r_p\}$ that communicates via AUTOSAR implicit communication [2]. At design time, runnable r_j is characterized by an execution-time C_{r_j} (depending on the development phase given either as abstract budget or sound upper bound/WCET) and a period T_{r_j}.

For allocation, runnables are grouped into **software components** (SW-Cs) $\mathcal{S} = \{\Psi_1, \ldots, \Psi_n\}$. Each SW-C Ψ_i contains a runnable subset $\mathcal{R}(\Psi_i) \subseteq \mathcal{R}$ and has a SIL α_{Ψ_i} that is inherited by its runnables: $\forall r_j \in \mathcal{R}(\Psi_i) : \alpha_{r_j} = \alpha_{\Psi_i}$. Its memory requirements are S_i^{ROM} and S_i^{RAM}, hence $\Psi_i = (\mathcal{R}(\Psi_i), \alpha_{\Psi_i}, S_i^{ROM}, S_i^{RAM})$.

Cause-effect chains $\mathcal{Z} = \{\zeta_1, \ldots, \zeta_d\}$ capture functional dependencies. Each chain $\zeta_c \in \mathcal{Z}$ is modeled as a DAG $(\mathcal{V}_c, \mathcal{E}_c)$ over runnables and is subject to a maximum data-age bound μ_c. In mixed-criticality settings, a chain may include QM functionality that is replaced by a substitute in safe mode. We denote such a substitute by $A\{QM\}$: it provides a high-assurance replacement for a QM runnable that is active only in safe mode, while the original QM functionality is used only in regular mode. The notation $A\{QM\}$ is descriptive rather than a new ASIL level: it marks ASIL-constrained substitute functionality for a QM runnable. We model this as an additional runnable with mutually exclusive activation semantics; details are given in Sec. 4.2.

Platform and Scheduling. We consider a platform's lockstep cores used for high-SIL workloads, modeled as $\mathcal{K} = \{\kappa_0, \ldots, \kappa_{n-1}\}$, where κ_k provides capacities M_k^{ROM} and M_k^{RAM}. We assume WCETs to be upper bounds including potential inter-core interference. Consequently, when using early-stage budgets, the result is a conditional architecture-level feasibility assessment that becomes sound once budgets are refined to actual WCETs.

We assume partitioned, preemptive fixed-priority scheduling. Runnables are allocated to periodic tasks (which may execute multiple runnables), and each task is statically mapped to a core and assigned a fixed priority. The specific allocation strategy of different runnable types to tasks, and the determination of task parameters is discussed in Section 4.

Task-local BSW extensions account for service overheads attached to application-task activations, such as RTE interaction, communication handling, mode management, and OS/RTE glue code. We model this overhead by a factor $f^{BSW}(k)$ depending on the number of runnable instances k in a task frame, yielding an execution-time extension $\hat{C}_t^{BSW}$ for task τ_t. The factor is not a communication-edge model but an empirical task-composition proxy.

Global BSW tasks implement central system services (e.g., OS, communication, diagnostics) and are modeled as periodic tasks $\beta_b \in \mathcal{B}$ comprising **up to 200 runnables** each, with parameters $\beta_b = (C_b^{BSW}, T_b^{BSW}, \alpha_{\beta_b})$. Their aggregated ROM footprint on core κ_k is captured by B_k^{ROM}, reducing the available capacity M_k^{ROM}. In early phases, the number and parameters of global BSW tasks are often unknown; we therefore estimate their impact via a conservative utilization/memory envelope (Sec. 4.4).

Requirements.

R1 Each runnable is assigned to exactly one periodic task with implicit deadlines; a task may contain multiple runnables. Tasks are mapped under partitioned FP scheduling.

R2 Clustering/merging must preserve SIL separation and chain context and must respect data-age constraints (modeled later as JLDs).

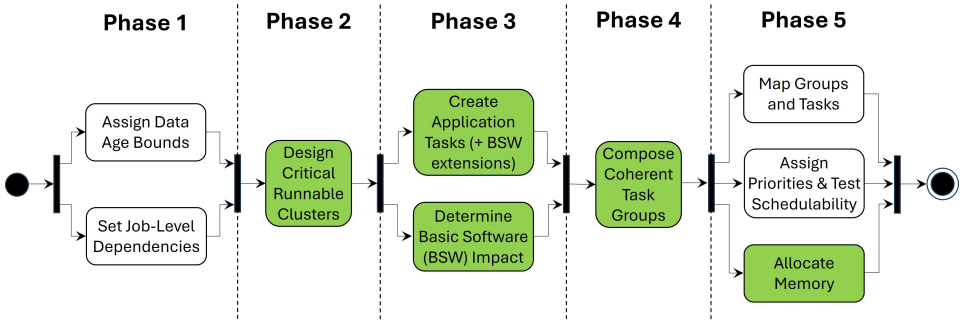


Figure 1 Methodical overview includes the development phases in sequential order. New contributions are shown in green.

R3 All runnables of the same SW-C must be co-located on one core; per-core ROM/RAM capacities must not be exceeded.

R4 BSW overheads are modeled as task-local extensions (via $f^{BSW}(\cdot)$) and as global BSW tasks with per-core memory envelopes.

4 Automotive System Analysis Across Iterative Development Phases

This Section introduces our approach for the iterative development of automotive systems. It consists of the following five main phases, which are also visualized in Figure 1:

1. **Deriving job-level dependencies.** We compute Job-Level Dependencies (JLDs) from the cause–effect chains and their data-age constraints, as described by Becker et al. [5]. These JLDs capture the ordering constraints required to satisfy data-age bounds and form the temporal backbone of our construction (Section 4.1).
2. **Building SIL-aware runnable clusters.** We group runnables of chains into clusters based on safety-criticality, ensuring they share the same SIL and periods. This prepares them for later merging while preserving SIL and chain semantics (Section 4.2).
3. **Creating application tasks with BSW extensions.** Each safety-critical cluster is transformed into a multiframe application task using the Arbitrary Periodic Solution (APS) [32, 4] method, which combines different periods within a single task. Budgets include runnable WCETs and frame-local BSW extensions (Section 4.3).
4. **Estimating global BSW impact.** Based on the task-local BSW extensions, we approximate, if necessary, the remaining global BSW tasks (utilization and memory footprint) using utilization bounds and platform parameters (Section 4.4).
5. **Composing task groups and mapping to cores.** JLDs and SW-C boundaries are used to create task groups that must reside on the same core. All application and BSW tasks are then allocated to cores using a worst-fit heuristic. Per-core schedulability and memory constraints are checked in tandem (Sections 4.5 and 4.6).

Overall, our toolchain takes an architectural model and WCETs (or budgets, depending on the development phase) and returns a schedulability- and memory-validated core mapping, along with the induced task decomposition (tasks, clusters, and remaining precedences). Each step consumes the artifacts of the previous step and produces the next engineering artifact: data-age dependencies, SIL-compatible runnable clusters, multiframe application tasks with local BSW extensions, an estimated global BSW envelope, coherent task groups, and finally a schedulability- and memory-validated core mapping.

Table 1 Symbols and notation used in this work. Indices: i SW-C, j runnable, a runnable instance/job, c chain, q cluster, t task, b global BSW task, g task group, k core, ℓ frame.

Symbol	Description	Symbol	Description
$\kappa_k \in \mathcal{K}$	CPU core κ_k of core set $\mathcal{K}$	$\mathcal{Z}$	set of cause-effect chains
$n = \lvert\mathcal{K}\rvert$	number of cores	ζ_c	cause-effect chain c
$M_k^{[ROM/RAM]}$	ROM/RAM capacity of κ_k	μ_c	data-age bound of ζ_c
$\Psi_i \in \mathcal{S}$	SW-C i of SW-C set $\mathcal{S}$	$chains(r_j)$	chains containing r_j
α_{Ψ_i}	SIL of Ψ_i	$\mathcal{D}$	set of runnable-level JLDs
$S_i^{[ROM/RAM]}$	ROM/RAM budget of Ψ_i	$r_{p,a} \to r_{q,u}$	JLD: inst. a of r_p prec. inst. u of r_q
$\mathcal{R}$ ($\supseteq \mathcal{R}_k$)	set of runnables (on κ_k)	h	hyperperiod for repeating JLDs
r_j / m_j	runnable j / instance of r_j	$\mathcal{G}$ (G)	remaining task-level JLDs
α_{r_j}	SIL inherited by r_j	Δ (Δ)	set of task groups
C_{r_j}, T_{r_j}	WCET/period of r_j	δ_g (δ_g)	task group (must be co-located)
$\tau_t \in \Gamma$ (Γ)	task t of periodic task set Γ	$\mathcal{H}$ (H)	remaining independent tasks
$m = \lvert\Gamma\rvert$	number of tasks	$\mathcal{C}$	set of SIL-aware clusters
π_t	fixed priority of task τ_t	χ_q	SIL-aware cluster q
$\mathcal{T}'$	set of multiframe app. tasks	$\mathcal{R}'$	set of runnable instances in a cluster
τ'_t	multiframe application task t	$\mathcal{D}^{inst}$	set of instance-level dependencies
$\alpha_{\tau'_t}$	SIL of τ'_t	T', η', f'	GCD period / major cycle / #frames
$T_{\tau'_t}$, $D_{\tau'_t}$	period/deadline of τ'_t	$\theta_{j,a}$	frame index of instance a of r_j
$\mathcal{B}$	set of global BSW tasks	$\theta_{j,a}^{min}$, $\theta_{j,a}^{max}$	earliest/latest feasible frame
β_b	global BSW task b	$\rho_{j,a}$	position of $r_{j,a}$ within its frame
C_b^{BSW}, T_b^{BSW}	WCET/period of β_b	$\hat{C}_\ell$	ASW execution time of frame ℓ
α_{β_b}	SIL of β_b	$\hat{C}_\ell^{BSW}$	BSW extension time of frame ℓ
U_{ub}	utilization bound for global BSW	C'_ℓ	total WCET of frame ℓ
B_k^{ROM}	BSW ROM footprint on core κ_k	$jobPre(j,a)$	predecessors of instance (j,a) in $\mathcal{D}^{inst}$
$\mathcal{E}$	set of task-local BSW extensions	$preExt(j,a)$	external predecessor (outside cluster)
ϵ_t	BSW extension attached to τ'_t	$sucExt(j,a)$	external successor (outside cluster)
$\hat{C}_t^{BSW}$	WCET of local BSW extension	b_ℓ	# runnables assigned to frame ℓ
Υ (Υ)	utilization of local BSW ext.	$f^{BSW}(\cdot)$	frame-wise local BSW cost factor

Conventions. All symbols and index conventions in this Section are summarized in Table 1. In particular, SW-Cs are indexed by i, runnables by j (with instances/jobs indexed by a), chains by c, clusters by q, tasks by t, global BSW tasks by b, task groups by g, cores by k, and frames within multiframe tasks by ℓ.

4.1 Meeting Data Age Constraints by Job-Level Dependencies

Constructing tasks from application runnables is challenging when these participate in cause–effect chains with data-age constraints. We address this by transforming end-to-end data-age bounds into JLDs using the heuristic by Becker et al. [5, 7]. Their approach first identifies all propagation paths, independent of scheduling and allocation policy, and then imposes an order on instances such that only those respecting data-age constraints remain.

We apply this heuristic to all chains in $\mathcal{Z}$ and obtain a set of JLDs $\mathcal{D}$. A JLD $r_p \xrightarrow{(a,u)} r_q \in \mathcal{D}$ states that instance a of runnable r_p must execute before instance u of runnable r_q. This constraint repeats with the hyperperiod $h = \mathrm{LCM}(T_{r_p}, T_{r_q})$, i. e., the least common multiple of the two periods. Satisfying all JLDs at runtime is sufficient to guarantee the original data-age constraints of all cause-effect chains. From this point, we therefore operate on $\mathcal{D}$ instead of the original data-age constraints.

4.2 Designing Criticality-Aware Clusters for System Modes

Automotive systems typically distinguish between two operating modes: *regular* and *safe*. Safe operation is entered upon specific events and represents a temporary degradation mode in which all QM-based runnables are inactive. The concrete configuration of cause-effect chains within the application structure depends on functional and non-functional design decisions and must adhere to timing and safety requirements.

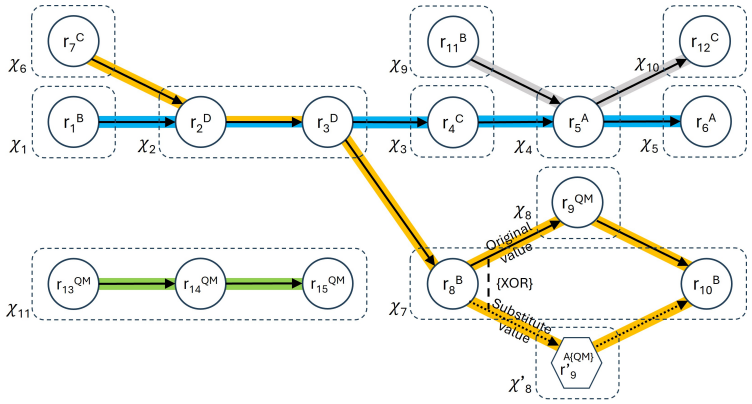


Figure 2 Composition of the safety-critical clusters χ_1 to χ_{11} according to runnable sequences of four different chains in two operating modes that are described by the colored underlays.

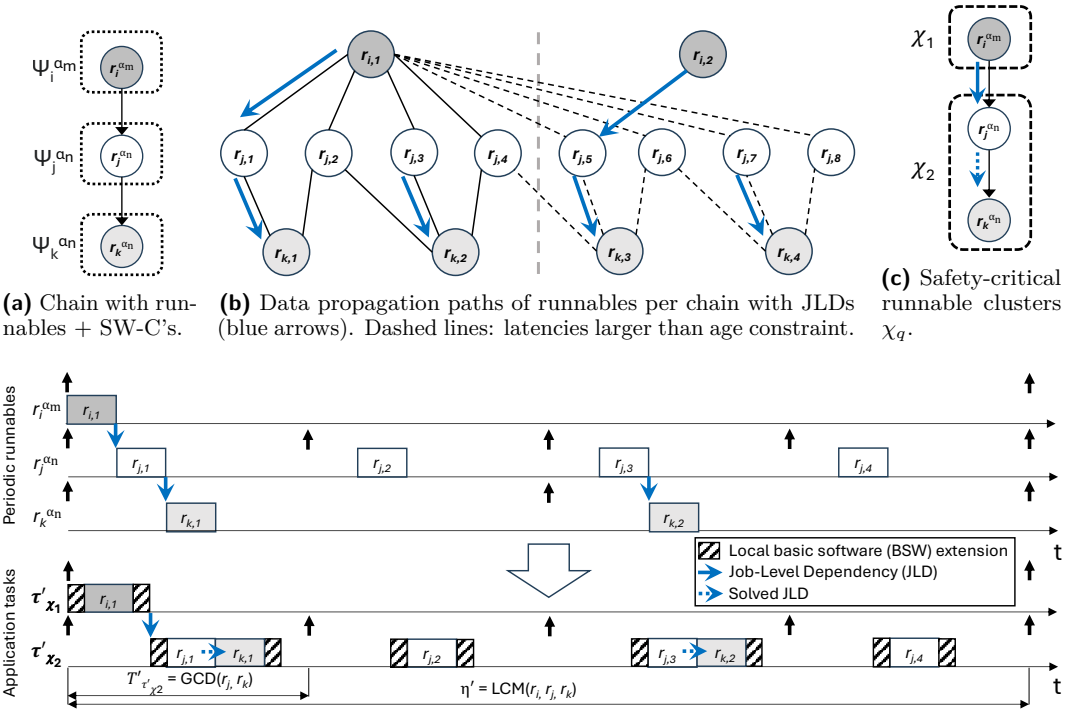
Figure 2 shows four example cause-effect chains (colored underlays) in both modes. A special case occurs in the orange chain: a QM runnable r_9^{QM} is integrated alongside SIL-constrained runnables in regular mode – it serves to improve QoS without having to be managed in accordance with SIL. In safe mode, this functionality is replaced by a high-assurance substitute runnable $r_9'^{A\{QM\}}$, which provides a simple replacement value. The XOR condition indicates that r_9^{QM} and $r_9'^{A\{QM\}}$ are mutually exclusive across operating modes. Conversely, the green chain is composed solely of QM components whose comfort functionality is temporarily deactivated during safe operation.

Cluster Construction Rules. Our goal is to construct a set of criticality-aware runnable clusters $\mathcal{C}$ such that all runnables in a cluster share the same SIL and compatible chain and period properties. These clusters are the units transformed into application tasks in the next phase (Section 4.3). We build $\mathcal{C}$ in three steps: (i) *Per-chain grouping:* for each chain ζ_c, we group runnables of equal SIL into preliminary clusters. (ii) *Conflict resolution:* if a runnable appears in multiple preliminary clusters (chain sharing), we iteratively split/merge affected clusters until each runnable belongs to exactly one cluster while preserving identical chain context $chains(r_j)$. (iii) *Period-cap refinement:* if a cluster contains more than two distinct periods, we keep the two most frequent periods (counted over runnable occurrences in that cluster; ties broken by choosing the shorter period) and split off remaining periods into separate clusters. The cap is not a theoretical limitation. It is an engineering heuristic: adding further periods increases the APS major cycle and the number of frames, which enlarges the CP model and often increases the maximum frame cost C' through sparse runnable placements. We therefore keep at most two periods per cluster to preserve schedulability margins and solver tractability. Formally, each cluster $\chi_q \in \mathcal{C}$ must satisfy:

$$\forall r_p, r_r \in \chi_q : \ \alpha_{r_p} = \alpha_{r_r} \ \wedge \ chains(r_p) = chains(r_r) \ \wedge \ periods(\chi_q) \leq 2. \tag{1}$$

Here, $chains(r_j)$ returns the set of cause-effect chains containing r_j, and $periods(\chi_q)$ returns the number of distinct runnable periods in cluster χ_q.

Figure 2 illustrates the construction by clusters χ_1 to χ_{11} across four chains. We first form per-chain clusters by grouping runnables of equal SIL that appear in the same chain. We then resolve conflicts where a runnable would belong to multiple clusters (e. g., due to chain sharing) by splitting or reassigning clusters until each runnable appears in exactly one cluster (e. g., χ_2 and χ_4). If a chain exhibits more than two periods for the same SIL, we assign the two most



(a) Chain with runnables + SW-C's.

(b) Data propagation paths of runnables per chain with JLDs (blue arrows). Dashed lines: latencies larger than age constraint.

(c) Safety-critical runnable clusters χ_q.

(d) Conversion from runnables to application tasks by restrictions of safety-critical clusters and local basic software extensions. The number of JLDs can be reduced by task merging based on the GCD procedure.

Figure 3 Successive steps to create application tasks from critical runnable clusters that are enhanced by basic software extensions.

frequent periods to the same cluster and keep additional periods separate to avoid excessive utilization growth. Finally, substitute functionality within a SIL-constrained chain is treated as a separate cluster (e. g., χ'_8). The resulting set $\mathcal{C}$ provides SIL/chain/period-consistent clusters that serve as input to the task-creation step in Section 4.3.

4.3 Creating Application Tasks from Safety-Criticality Runnable Clusters with Basic Software Extensions

Before detailing the construction of application tasks, we illustrate the successive steps in Figure 3. Figure 3a shows an example cause-effect chain $\zeta = r_1 \rightarrow r_2 \rightarrow r_3$ with periods $T_{r_1} = 20\,\text{ms}$, $T_{r_2} = 5\,\text{ms}$, and $T_{r_3} = 10\,\text{ms}$. Each software component (SW-C) Ψ_1, Ψ_2, and Ψ_3 contains one runnable r_1, r_2, and r_3, respectively, and the safety integrity level (SIL) α_m of Ψ_1 differs from the two subsequent SW-Cs (Ψ_2 and Ψ_3) which share the same SIL α_n.

Figure 3b then shows the data propagation paths derived from the data-age constraints, as explained in Section 4.1; paths that violate the age bound are removed (dashed lines). Based on these chains and SILs, safety-criticality clusters $\chi_q \in \mathcal{C}$ are formed as described in Section 4.2 (Figure 3c). Finally, Figure 3d summarizes the transformation from periodic runnables to application tasks, combining timing constraints, SIL restrictions, and basic software extensions (in particular for communication services). In the following, we formalize this construction and present the heuristic used to create the corresponding application tasks.

Multi-Frame Tasks. For each critical cluster $\chi_q \in \mathcal{C}$, we create exactly one multiframe application task τ'_t that inherits the cluster's SIL. We follow the Arbitrary Periodic Solution (APS) [32, 4] method, which combines multi-periodic runnables into a single AUTOSAR-compliant multiframe task [42]. A multiframe task is a periodic task with period T' and deadline D', whose f' frames have frame-specific WCETs $\{C'_1, \ldots, C'_{f'}\}$ repeating every f' activations. For a cluster $\chi_q = \{r_1, \ldots, r_n\}$ with runnable periods T_{r_j}, APS defines

$$T' = \operatorname*{gcd}_{r_j \in \chi_q} T_{r_j}, \qquad \eta' = \operatorname*{lcm}_{r_j \in \chi_q} T_{r_j},$$

with $f' = \eta'/T'$ frames over one major cycle η'. Runnable r_j executes $m_j = \eta'/T_{r_j}$ instances per cycle. We denote the a-th instance by $r_{j,a}$ for $a \in \{0, \ldots, m_j - 1\}$, and assign it to exactly one frame $\theta_{j,a} \in \{1, \ldots, f'\}$. Within each frame, the order of runnable instances is given by $\rho_{j,a}$. We assume an implicit deadline $D' = T'$ for the merged task.

To go beyond [4], which focuses on multi-rate patterns and JLDs, we integrate SIL and BSW impact into the task configuration. We formulate the assignment of runnable instances to frames as a Constraint Programming (CP) problem with the objective to minimize C' (Eq. 10), that is, it balances frame loads to reduce worst-case utilization. All runnable instances in χ_q form the instance set

$$\mathcal{R}' = \bigcup_{r_j \in \chi_q} \{r_{j,0}, \ldots, r_{j,m_j-1}\}.$$

Runnable-level JLDs between runnables of χ_q are transformed into precedences between runnable instances, repeated over η'. For example, a JLD $r_p \xrightarrow{(a,u)} r_q$ induces an instance-level precedence $r_{p,a} \to r_{q,u}$ (and corresponding repetitions in subsequent cycles). We denote the resulting set of instance-level dependencies by $\mathcal{D}^{inst}$.

CP Outcome and Failure Handling. For each cluster χ_q we solve the CP model (and, if necessary, retry after conservative cluster refinement) to obtain (a) a frame assignment $\theta_{j,a}$ and within-frame order $\rho_{j,a}$ for all instances in $\mathcal{R}'$, and (b) frame costs C'_ℓ used to derive the periodic abstraction (C', D'). If the CP model is infeasible (e. g., due to tight external predecessors/successors), we conservatively refine the cluster by splitting off the runnable(s) responsible for the violated bounds and retry.

Decision Variables. The main decision variables are: (a) $\theta_{j,a} \in \{1, \ldots, f'\}$, the frame index of runnable instance $r_{j,a}$, and (b) $\rho_{j,a} \in \{1, \ldots, |\mathcal{R}'|\}$, the position of $r_{j,a}$ within its frame. For convenience, we use $\theta_{j,a}(\ell)$ as an indicator that equals 1 if $r_{j,a}$ is assigned to frame ℓ, and 0 otherwise. The exclusive (application-only) execution time of frame ℓ is

$$\hat{C}_\ell = \sum_{r_{j,a} \in \mathcal{R}'} C_{r_j} \cdot \theta_{j,a}(\ell). \tag{2}$$

Let $b_\ell = \sum_{r_{j,a} \in \mathcal{R}'} \theta_{j,a}(\ell)$ denote the number of runnable instances in frame ℓ. We assign a task-local BSW extension factor $f^{BSW}(b_\ell)$ based on the case-study profile: smaller frames incur larger relative BSW overhead, whereas larger frames amortize task-boundary service costs across more runnable instances. This yields the frame-local BSW extension

$$\hat{C}^{BSW}_\ell = f^{BSW}(b_\ell) \cdot \hat{C}_\ell. \tag{3}$$

Thus, $f^{BSW}(\cdot)$ captures empirically observed task-local BSW extension ratios, not the number or topology of explicit runnable-to-runnable communication edges. The total WCET of frame ℓ is $C'_\ell = \hat{C}_\ell + \hat{C}_\ell^{BSW}$. For later schedulability analysis, we abstract the multiframe task as a single periodic task with

$$C' = \max_{\ell \in \{1,\dots,f'\}} C'_\ell, \qquad D' = T'.$$

JLD Handling and Frame Bounds. The execution window of runnables, when assigned to a task τ', is either equal to the task period or smaller. If JLDs are not considered during frame assignment, it can result in assignments that make it impossible to meet an imposed JLD. For example, if a successor runnable is allocated to a frame that must complete before the predecessor runnable is released. External predecessors/successors further restrict feasible frames (Equations 4–5).

Not all original JLDs are fully internal to the cluster; some relate runnables of different SILs and thus span multiple clusters. We capture these via helper functions: (a) $jobPre(j,a)$ returns all predecessors of $r_{j,a}$ in $\mathcal{D}^{inst}$, and (b) $preExt(j,a)$ / $sucExt(j,a)$ return predecessors / successors outside χ_q.

Release times and deadlines restrict the earliest and latest feasible frame for each runnable instance. For $r_{j,a}$ we derive

$$\theta_{j,a}^{min} = \begin{cases} \frac{a \cdot T_{r_j}}{T'} + 1, & \text{if } preExt(j,a) = \emptyset, \\ \left\lceil \frac{\max_{r_{p,u} \in preExt(j,a)} (u+1) \cdot T_{r_p}}{T'} \right\rceil + 1, & \text{otherwise,} \end{cases} \tag{4}$$

$$\theta_{j,a}^{max} = \begin{cases} \frac{(a+1) \cdot T_{r_j}}{T'}, & \text{if } sucExt(j,a) = \emptyset, \\ \left\lfloor \frac{\min_{r_{p,u} \in sucExt(j,a)} u \cdot T_{r_p}}{T'} \right\rfloor, & \text{otherwise,} \end{cases} \tag{5}$$

and enforce

$$\theta_{j,a}^{min} \le \theta_{j,a} \le \theta_{j,a}^{max}. \tag{6}$$

Constraints. To avoid overruns within one activation of the multiframe task, we require the worst-case frame time to fit into one base period:

$$C' \le T'. \tag{7}$$

Within each frame, runnable instances must have a strict order:

$$\text{allDifferent}(\{\rho_{j,a} \mid r_{j,a} \in \mathcal{R}'\}). \tag{8}$$

All internal precedences in $\mathcal{D}^{inst}$ must be honored by either placing predecessors in earlier frames or, if in the same frame, before their successors:

$$(\theta_{p,u} < \theta_{j,a}) \vee (\theta_{p,u} = \theta_{j,a} \wedge \rho_{p,u} < \rho_{j,a}), \tag{9}$$

for all $r_{p,u} \in jobPre(j,a)$.

Objective Function. Since later analysis uses the periodic abstraction with execution time C', we balance frame loads by minimizing the worst-case frame time:

$$minimize\ C'. \tag{10}$$

From runnable-level to task-level JLDs. After solving the CP model, runnable instances are bound to frames, and frames are bound to tasks. JLDs between instances belonging to different clusters induce task-level JLDs between the corresponding multiframe tasks. A JLD between runnables mapped to different tasks is translated into a task-level JLD between these frames, repeating with the hyperperiod of the two tasks. Thus, a JLD $r_{r_i} \xrightarrow{(a,b)} r_{r_j}$, with r_i mapped to τ'_k and r_j mapped to τ'_l, is translated to a new JLD $r_{\tau'_k} \xrightarrow{(\theta_{i,a},\theta_{j,b})} r_{\tau'_l}$. We apply this translation to all cross-cluster dependencies, obtaining a reduced set of JLDs $\mathcal{G}$.

Finally, the created application tasks form the set $\mathcal{T}'$, and the corresponding BSW extensions form $\mathcal{E}$, with $\mathcal{E} \subseteq \mathcal{T}'$. These sets, together with $\mathcal{G}$, are the input to the subsequent steps on global BSW impact, task grouping, and core allocation.

4.4 Determining Basic Software Impact

BSW aggregates the runtime environment, communication stacks, OS services, diagnostics, and related functionality. Its contribution to execution time and memory is substantial and depends on the task structure. We model this impact through two distinct channels.

First, *task-local BSW extensions* account for service work attached to application-task activation boundaries, such as RTE and communication handling. For every multiframe task $\tau'_t \in \mathcal{T}'$, a corresponding extension $\epsilon_t \in \mathcal{E}$ contributes an execution-time extension $\hat{C}_t^{BSW}$. The frame-local factor $f^{BSW}(\cdot)$ is derived from the task-local BSW extension profile in Section 5.1. For each frame, we select the BSW-extension ratio from the range associated with its number of runnable instances and use a conservative value from that range. We additionally enforce the monotone trend observed in the case study: relative BSW overhead decreases as more runnable instances share the same task frame. Thus, $f^{BSW}(\cdot)$ is an empirical early-phase abstraction of task-local service overhead, not a communication-edge model. The cumulative utilization of all task-local extensions is

$$\Upsilon = \sum_{\tau'_t \in \mathcal{T}'} \frac{\hat{C}_t^{BSW}}{T'_t}. \tag{11}$$

Here, Υ summarizes the utilization contribution of all task-local BSW extensions, based on each extension WCET $\hat{C}_t^{BSW}$ and the period T'_t of the corresponding application task.

Beyond these extensions, the system also requires dedicated global BSW tasks $\beta_b \in \mathcal{B}$ for system-wide operations, described by $\beta_b = (C_b^{BSW}, T_b^{BSW}, \alpha_{\beta_b})$. From a BSW perspective, the total number of periodic tasks considered for the utilization envelope is

$$m = |\mathcal{E}| + |\mathcal{B}|. \tag{12}$$

Since every application task has exactly one attached BSW extension, we have $|\mathcal{E}| = |\mathcal{T}'|$. In early development stages, however, engineers typically lack a reliable model of the final global BSW task set $|\mathcal{B}|$, yet feasibility decisions must already be made.

A finer model could distinguish individual RTE calls, sender–receiver labels, and inter-task communication paths. We deliberately use the coarser task-composition proxy because these details are often unavailable in the early architectural phase targeted here, whereas the number of runnable instances per task frame is already determined by clustering.

Second, *global BSW tasks* represent periodic platform services that are scheduled independently of application tasks, such as OS, RTE, communication stacks, and diagnostics. In early development stages, the final set of such tasks is often not yet available, although feasibility decisions must already reserve timing and memory capacity for them. Eq. 13

therefore estimates only the *number* of global BSW tasks used in this early-phase abstraction. It is an empirical surrogate that enables integrated timing/memory analysis despite missing information based on empirical data. For our evaluation, we calibrated it from the abstracted case-study profile, not a schedulability theorem and not a consequence of the utilization bound used below. Once the concrete BSW task set is available, Eq. 13 is replaced by the actual task set $|\mathcal{B}|$ and only the feasibility checks in Equations (14) and (15) remain.

$$|\mathcal{B}| = \left\lceil |\mathcal{T}'| \cdot \begin{cases} \dfrac{n^{\frac{1}{|\mathcal{T}'|}}}{(2^{\frac{n}{|\mathcal{T}'|}}+1)\cdot 2^{\frac{1}{n}}}, & \text{if } \mathcal{E}=\emptyset, \\ \dfrac{(n-\Upsilon)^{\frac{1}{|\mathcal{T}'|}}}{(n+\Upsilon)^{\frac{1}{|\mathcal{T}'|}}\cdot(2^{\frac{1}{n}}+1)^2}, & \text{otherwise.} \end{cases} \right\rceil \tag{13}$$

How it is used. The estimate in Eq. 13 partitions an unavailable final BSW configuration into two modeled early-phase channels. Task-local extensions reserve BSW work that scales with the constructed application tasks; global BSW tasks reserve separate periodic platform services. These channels are not functionally interchangeable. Rather, the surrogate prevents the early model from double-counting BSW load: if task-local extensions already reserve part of the BSW utilization, the remaining global-BSW estimate is reduced accordingly. The parameters of the generated global BSW tasks – periods, WCETs, SIL labels, and memory envelopes – are then sampled from the global BSW profile summarized in Section 5.1.

Eq. 13 determines the size of the generated global BSW task set. Its utilization is constrained separately. To avoid unrealistically large BSW load, we bound the combined task-local and global BSW utilization using the multiprocessor utilization bound of López et al. [36]. For a platform with $n = |\mathcal{K}|$ cores and an admissible BSW task-count range, this yields an upper BSW-utilization envelope $\mathcal{U}_{ub}$. We require:

$$\sum_{\beta_b \in \mathcal{B}} \frac{C_b^{BSW}}{T_b^{BSW}} \leq \mathcal{U}_{ub} - \Upsilon. \tag{14}$$

Equation 14 states that the utilization of global BSW tasks must fit into the remaining BSW envelope after reserving the utilization already accounted for by task-local extensions.

Finally, BSW also consumes ROM. Let B_k^{ROM} denote the ROM footprint associated with BSW on core κ_k (both for global BSW tasks in $\mathcal{B}$ and task-local BSW extensions), and M_k^{ROM} the per-core ROM capacity available to software. We require:

$$\forall \kappa_k \in \mathcal{K}: \quad B_k^{ROM} < M_k^{ROM}. \tag{15}$$

This constraint ensures that the BSW memory footprint remains below the available per-core ROM budget, leaving sufficient space for application tasks $\mathcal{T}'$. The combined utilization and memory constraints for BSW and application tasks are enforced during allocation and schedulability analysis in Section 4.6, where the overall task set $(\mathcal{B} \cup \mathcal{T}') = \Gamma$ is mapped to the multicore platform $\mathcal{K}$.

4.5 Composing Coherent Task Groups

The next step prepares for core allocation by identifying sets of tasks that must be co-located. After the merging procedure in Section 4.3, some job-level dependencies (JLDs) become internal to individual tasks, while others remain between tasks. The latter form the reduced

set of task-level JLDs $\mathcal{G} \subset \mathcal{D}$. In addition, SW-C boundaries impose further co-location requirements: all tasks that contain cyclic runnables of the same SW-C must reside on the same core to respect the SW-C's memory footprint and SIL context.

We capture these constraints by grouping tasks into coherent task groups Δ, where all tasks inside a group must be allocated to the same core. First, each task-level JLD induces a grouping constraint: for any abstracted JLD $\tau_t \xrightarrow{(a,u)} \tau_{t'} \in \mathcal{G}$, the two tasks must be placed in the same group,

$$\forall \tau_t \xrightarrow{(a,u)} \tau_{t'} \in \mathcal{G} : \exists \delta_g \in \Delta \text{ s.t. } \tau_t, \tau_{t'} \in \delta_g. \tag{16}$$

In practice, an empty set Δ is populated iteratively: when a JLD connects two tasks, a new group is created (or existing groups are merged) such that both tasks end up in the same δ_g.

Second, we must respect the structure of software components. For each SW-C $\Psi_i \in \mathcal{S}$, consider its cyclic runnables. These runnables may be mapped either to the same task or to different tasks after merging. All tasks containing cyclic runnables of the same SW-C must, however, be assigned to the same task group to ensure that the SW-C's ROM/RAM footprint is allocated consistently on a single core and to preserve its SIL context. Let $task(r_j)$ denote the task containing runnable r_j, and let $group(\tau_t)$ denote the task group containing task τ_t. We then require

$$\forall \Psi_i \in \mathcal{S} : \forall r_j, r_p \in \Psi_i : group(task(r_j)) = group(task(r_p)). \tag{17}$$

In other words, no SW-C may be split across different groups: if two tasks contain cyclic runnables from the same SW-C, they must belong to the same δ_g. Existing groups are extended accordingly, or new groups are created when needed.

After processing all JLDs and SW-Cs, we obtain: (a) a set of coherent task groups Δ, each of which must be mapped to a single core, and (b) a remaining set of independent tasks $\mathcal{H}$, defined as the tasks not contained in any group, $\mathcal{H} = \Gamma \setminus \bigcup_{\delta_g \in \Delta} \delta_g$.

In the subsequent allocation step (Section 4.6), each task group in Δ and each individual task in $\mathcal{H}$ is treated as an indivisible unit that must be placed on exactly one core in $\mathcal{K}$, while satisfying both schedulability and memory constraints.

4.6 Orchestrating Task Abstraction, Mapping and Scheduling

This final step brings together the previous phases. As a result of runnable merging, multiframe application tasks are represented by periodic abstractions. Some precedence constraints may still remain between application tasks if they could not be resolved internally during merging; these constitute the reduced task-level JLD set $\mathcal{G}$.

Similarly, we represent the global share of basic software by periodic BSW tasks $\mathcal{B}$. Hence, at this stage both application and global BSW tasks are abstracted and treated uniformly for scheduling, i. e., $\Gamma = \mathcal{T}' \cup \mathcal{B}$.

For memory allocation, however, we must distinguish between the two types: global BSW contributes a pre-accounted per-core ROM envelope (Eq. 15), whereas application memory costs arise from the software components (SW-Cs) containing the mapped runnables. In particular, all tasks containing runnables of a given SW-C must be mapped to the same core, as enforced by the task-group construction in Section 4.5.

To test schedulability under JLD constraints, we adopt the approach of Forget et al. [23]: for the set of tasks mapped to a given core, we synthesize offsets and deadlines that satisfy the extended precedences and then assign fixed priorities (via a modified Audsley-style procedure) such that the resulting task set is schedulable. We use partitioned scheduling and apply this per core; by construction of Δ (Section 4.5), all tasks connected by task-level JLDs are co-located on the same core.

Orchestration loop. We treat each task group $\delta_g \in \Delta$ as an indivisible unit, map groups first (worst-fit by utilization), and then place remaining tasks in $\mathcal{H}$. After each placement, we (i) run the per-core schedulability test described above and (ii) check per-core ROM/RAM feasibility and the BSW ROM envelope. If a placement causes infeasibility, we backtrack locally by selecting the next-best core (worst-fit order) for the affected unit.

Let $\mathcal{R}_k \subseteq \mathcal{R}$ denote the set of runnables mapped to core κ_k after allocation, and let $\mathcal{S}_k = \{\Psi_i \in \mathcal{S} \mid \exists r_j \in \mathcal{R}_k : r_j \in \Psi_i\}$ be the induced set of SW-Cs placed on core κ_k. Memory feasibility is then checked per core as

$$\forall \kappa_k \in \mathcal{K} : \quad \sum_{\Psi_i \in \mathcal{S}_k} S_i^{ROM} + B_k^{ROM} \leq M_k^{ROM}, \qquad \sum_{\Psi_i \in \mathcal{S}_k} S_i^{RAM} \leq M_k^{RAM}. \tag{18}$$

This accounts for the ROM/RAM footprint of a SW-C's runnables assigned to a core.

Assuming that all timing used by our analysis are sound upper bounds (i. e., include any potentially unmodeled interference) mapping accepted by our orchestration satisfies: (i) data-age constraints; (ii) SIL separation as modeled; (iii) ROM/RAM and (iv) BSW accounting by construction of extensions and enforcement of the utilization/ROM envelopes.

5 Evaluation

Our primary goal is to quantify the practical impact of treating SIL constraints, BSW costs, and memory as first-class citizens. Thus, we investigated how progressively richer modeling and clustering decisions affect feasibility margins and resource efficiency. In addition, we included an ablation that transitions from conservative budgets to WCETs to assess how much of the resulting clustering and mapping structure is retained in refinement.

Note that a direct comparison against external baselines was not straightforward for two reasons. First, most related allocation and precedence-aware scheduling approaches assume that the *task decomposition is given* and therefore do not cover the coupled design space we target: SIL-constrained runnable clustering and multirate task construction together with explicit modeling of task-local BSW extensions and global BSW tasks. Second, there are currently no readily available public benchmarks that reflect automotive systems at our granularity – particularly in distinguishing BSW extensions from global BSW tasks and enforcing SIL constraints at the component/runnable level. This benchmark gap is the motivation for our case study and the derived system generator used throughout this evaluation. We therefore compare against progressively less informed variants of our own flow to isolate the contribution of individual modeling and optimization steps.

5.1 Industrial Case Study

Our evaluation is grounded in an abstracted profile of a production-grade automotive motion/drive controller [15]. Due to IP restrictions, we cannot disclose the original architecture, functionality, or exact per-task timing values. Instead, the companion case study extracts correlation-preserving ranges and distributions for the properties that affect our optimization problem: SIL composition, runnable timing, cause-effect-chain structure, task-local BSW extensions, global BSW services, and memory envelopes. The following profile parameters are used directly by our benchmark generator; similar to TASKers [18], the generator constructs instances under multiple interacting structural and timing constraints.

Application structure and SIL mix. Software components (SW-Cs) are the architectural containers that carry SIL labels and memory footprints; their runnables inherit the corresponding SIL. The abstracted SIL distribution is QM/ASIL A/B/C/D = 41 %/7 %/14 %/10 %/28 %. SIL-specific memory ranges span 6 kB to 255 kB ROM and 0.5 kB to 29 kB RAM. The SW-C profile has a correlation factor of 0.5174 with the industrial baseline.

Table 2 Data and profiles from our industrial case-study [15], used for benchmark synthesis.

Property	Abstracted values used in synthesis
SIL distribution	QM: 41 %; ASIL A: 7 %; ASIL B: 14 %; ASIL C: 10 %; ASIL D: 28 %
SW-C memory	ROM: 6–255 kB; RAM: 0.5–29 kB; sampled by SIL class
Runnable periods	1, 5, 10, 20, 50, 100, 200, 500, 1000 ms
Dominant app. periods	5 ms: 31 %; 10 ms: 25 %; 20 ms: 12 %; 100 ms: 10 %
Fast runnable budgets	1 ms: 15–290 μs; 5 ms: 10–725 μs; 10 ms: 22–1218 μs
Slow runnable budgets	100 ms: 228–2447 μs; 500 ms: 403–6176 μs; 1000 ms: 1209–9200 μs
Cause-effect chains	1–3 activation patterns; 2–18 runnables per chain; age bound: 1.8–4.9× hyperp.
Task-local BSW ext.	1 runnable: 24–57 %; 2: 19–48 %; 3: 15–39 %; ≥6: 8–14 %
Global BSW periods	1, 2, 5, 10, 20, 50, 100, 200 ms
Dominant BSW periods	5 ms: 31 %; 10 ms: 34 %
Global BSW budgets	14–1698 μs over all BSW periods; period-specific ranges used for sampling
BSW memory envelope	634–1258 kB ROM reserved per lockstep core
BSW system scale	approx. 6000 system-related BSW runnables; up to 200 runnables per BSW task

Runnable timing. Cyclic runnables use automotive periods from 1 ms to 1000 ms. The dominant application periods are 5 ms (31 %) and 10 ms (25 %); additional mass appears at 20 ms (12 %) and 100 ms (10 %). Period-specific timing budgets range from 10 µs to 9200 µs; the runnable timing profile has a correlation factor of 0.4322 with the baseline.

Cause-effect chains. Chains contain one to three activation patterns and between 2 and 18 runnables. Their data-age bounds are specified relative to the chain hyperperiod, with factors between 1.8 and 4.9. Runnables may occur in multiple chains, which is why the generator must preserve chain context when constructing SIL-compatible clusters.

Task-local BSW extensions. Task-local BSW work models services attached to application-task activations, such as RTE and communication handling. The case study shows that this overhead depends strongly on the number of runnable instances in a task frame. The abstracted BSW ratio decreases with task size: 24 % to 57 % for single-runnable frames, 19 % to 48 % for two runnables, 15 % to 39 % for three runnables, and 8 % to 14 % for frames with at least six runnables. The corresponding correlation factor is 0.5042.

Global BSW tasks. Separate periodic BSW tasks model global platform services such as OS, RTE, communication stacks, and diagnostics. Their periods range from 1 ms to 200 ms, with dominant shares at 5 ms (31 %) and 10 ms (34 %); period-specific budgets range from 14 µs to 1698 µs. The global BSW task profile has a correlation factor of 0.4189. The underlying industrial BSW set contains approximately 6000 system-related BSW runnables, with up to 200 runnables per BSW task.

BSW memory envelope. Global BSW memory is accounted for separately from application memory. The abstracted per-core BSW ROM reservation ranges from 634 kB to 1258 kB and reduces the ROM capacity available for application SW-Cs on that core.

The synthesized benchmarks should therefore be read as application slices sampled from this industrial profile, not as complete replicas of the deployed ECU. This distinction is important for interpreting the scale of the experiments: the generated instances vary the number of application runnables to obtain many independent optimization problems, whereas the task-local and global BSW abstractions remain anchored in the system-wide BSW profile. Table 2 details those numbers further.

5.2 Experimental Setup

Benchmark synthesis. Based on the case-study profile [15], we implemented the system and task set generator *Driverator* [14]. A random seed, an application-size parameter, and a configuration variant identify each generated instance. The application-size parameter

controls the number of generated application runnables (10 to 70); these instances are application slices sampled from the industrial profile, not complete replicas of the deployed ECU. The much larger figure of approximately 6000 runnables refers to the system-wide BSW profile from which global BSW task parameters are derived.

For each seed and size, Driverator first samples SW-Cs, assigns SIL labels, and draws SIL-specific ROM/RAM footprints. It then creates cyclic application runnables, binds them to SW-Cs so that runnables inherit the SW-C SIL, and samples periods and timing budgets from the period-specific case-study ranges. Next, it constructs cause-effect chains by sampling the number of activation patterns, the number of runnables per chain, and the corresponding data-age bound. Finally, it derives task-local BSW extension factors from the task-composition profile and generates global BSW tasks from the global BSW profile. The resulting base instance is then analyzed under the configuration variants in Table 3.

Filtering and counted instances. We distinguish three outcomes. A generated instance is *attempted* if the generator produced a complete application/BSW model for a given seed and size. It is *schedulable* if the corresponding variant can construct tasks, derive JLD-compatible timing constraints, and pass the schedulability test. It is *allocatable* if, in addition, the resulting task set can be mapped to cores while satisfying the SIL and memory constraints enabled for that variant. Thus, the columns *Sched. Systems* and *Allocatable* in Table 3 report post-analysis outcomes, not independent benchmark pools. Since each configuration variant enables different constraints, the counts are intentionally variant-specific. The total row summarizes the number of completed analyses across all variants; it should not be interpreted as the number of distinct generated base systems.

Platform model and implementation. As a representative allocation target, we selected the AURIX™ TC39x family: six cores, with four lockstep cores serving high-SIL workloads and dedicated RAM. We excluded the two performance cores as they may not run SIL runnables. We did not model interference and assumed it to be either prevented by platform configuration or covered by conservative WCETs. We assumed partitioned, preemptive fixed-priority scheduling (Section 3). All experiments ran on a 64-core AMD CPU at 3.1 GHz with 384 GB RAM; IBM CPLEX Optimizer 22.1.1 solves the CP model in Section 4.3.

Configuration variants. To expose the effect of modeling choices and clustering decisions, we evaluated five configuration variants (Table 3, Figures 4 and 5). The column *Basic Software* distinguishes whether a variant accounts only for task-local *BSW extensions* (*Exts*), only for periodic *global BSW tasks* (*Tasks*), or for both (*Both*). Variants ① and ② are diagnostic, requirement-incomplete boundary cases: they deliberately combine BSW-channel selection with simplified constraint settings and are therefore not clean single-factor ablations. We retain them to show, in compact form, how partial BSW accounting and omitted constraints affect the resulting utilization. Variants ③–⑤ form the feasible comparison set: they enable both data-age and SIL constraints and progressively introduce clustering/merging under full BSW accounting. The additional BSW reference Ⓑ in Figure 5a is not a configuration variant; it disables SIL and data-age constraints only to expose the complete BSW approximation, that is the split between task-local extensions and global BSW tasks on budgets.

① **NoJLD+BSWTasks (non-compliant).** We disabled data-age constraints (no JLDs / no age bounds) while keeping SIL constraints enabled. Application runnables remained unmerged (one runnable per task), and we accounted only for *global BSW tasks* while omitting task-local BSW extensions. This variant is a diagnostic boundary case, not

Table 3 Overview of our evaluation scenarios by variant, properties, and costs.

Configuration Variant	Sched. Systems	Alloc. Systems	Data Age	SIL	Task Merging	Basic Software	Analysis Time $[min, max]$
① NoJLD+BSWTasks[1] (non-compl.)	3,386	3,248	No	Yes	No	Tasks	[1 s, 1.4 h]
② NoSIL+BSWExts[2] (non-compl.)	2,477	2,212	Yes	No	Yes	Exts	[1 s, 1.2 h]
③ NoCluster+BSWFull (**Baseline**)	2,865	2,701	Yes	Yes	No	Both	[2 s, 2.1 h]
④ SeqCluster+BSWFull	2,594	2,449	Yes	Yes	Yes	Both	[1 s, 2.7 h]
⑤ SILCluster+BSWFull	2,503	2,363	Yes	Yes	Yes	Both	[1 s, 2.9 h]
Total	**13,825**	**12,973**	–	–	–	–	[1 s, 2.9 h]

[1] Only global BSW tasks are considered. [2] Only task-local BSW extensions are considered.

a feasible architecture: it combines missing chain semantics with global-only BSW accounting. It is included to show the scale of global BSW tasks within a conservative, unmerged task structure and to provide a lower-dimensional reference for later full-accounting variants.

② **NoSIL+BSWExts (non-compliant).** We disabled SIL constraints while keeping data-age constraints enabled and allowing task merging. We accounted only for task-local *BSW extensions* while omitting global BSW tasks. This variant is again a diagnostic boundary case, not a feasible architecture: it combines missing safety partitioning with local-only BSW accounting. It is included to show how JLD-constrained task construction interacts with task-local BSW extensions when SIL-induced splitting is removed.

③ **NoCluster+BSWFull (baseline).** We enabled both data-age and SIL constraints and account for *both* BSW extensions and global BSW tasks. Runnables remain unmerged (one runnable per task). This establishes a feasible baseline with full BSW accounting, against which merging/clustering decisions are evaluated.

④ **SeqCluster+BSWFull.** Starting from ③, we enabled JLD-constrained task merging (data-age aware) under SIL constraints and full BSW accounting. Within each chain, consecutive runnables sharing the same SIL are clustered, preserving the original runnable order; runnables shared across chains are treated separately. This captures the first stage of reducing pessimism through merging while preserving chain semantics.

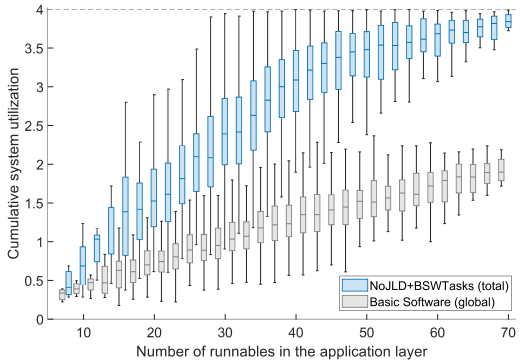
⑤ **SILCluster+BSWFull.** Our full approach (Section 4.2–4.6). We applied SIL-aware cluster construction, JLD-aware merging (even when non-consecutive), and full BSW accounting to minimize pessimism while preserving both safety partitioning and chain semantics.

Table 3 summarizes the evaluation scale and runtimes. Across all variant-specific analyses, 13,825 runs produced schedulable task sets and 12,973 of those also satisfied the enabled memory-allocation constraints. The table also lists which constraints were enforced (data age / SIL / task merging) and which BSW accounting mode was used.

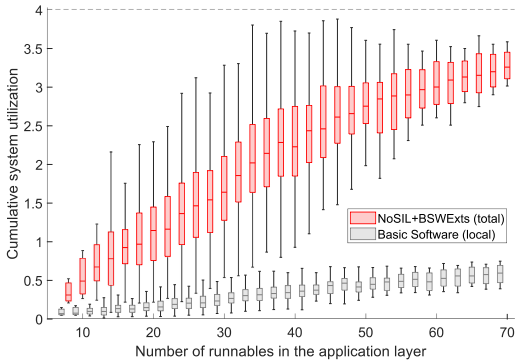
5.3 Feasibility, Efficiency, and Practical Impact

The following results were obtained from *synthesized systems* produced by our generator, using *budgets only*. Global BSW tasks were estimated via Eq. 13 and constrained by Eqs. 14–15. The *Reality Check* applies the full approach to the industrial controller configuration.

Analysis Overhead. Across all variants and synthesized systems, end-to-end runtimes range from 1 s to 2.9 h and vary substantially with system structure (and thus analyzability), yet remain within a practical range for iterative engineering workflows. The dominant costs stem from application task construction (CP-based multiframe configuration) and the subsequent timing analysis; other steps contribute only minor overhead in comparison.

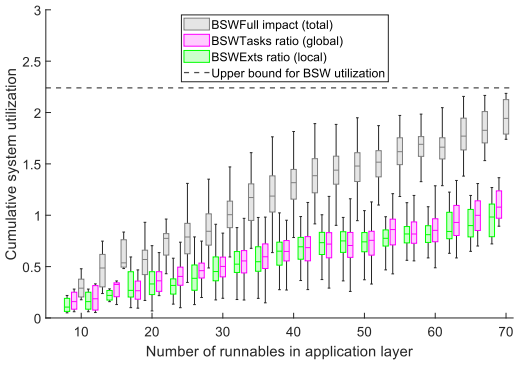


① NoJLD+BSWTasks (non-compliant).

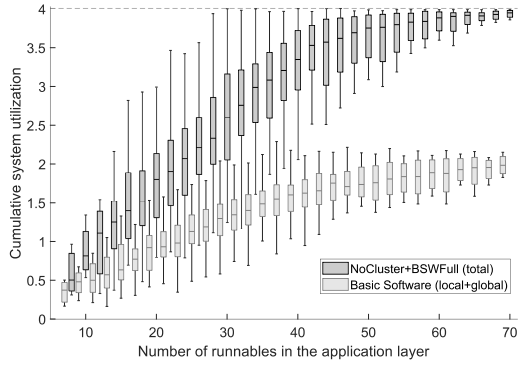


② NoSIL+BSWExts (non-compliant).

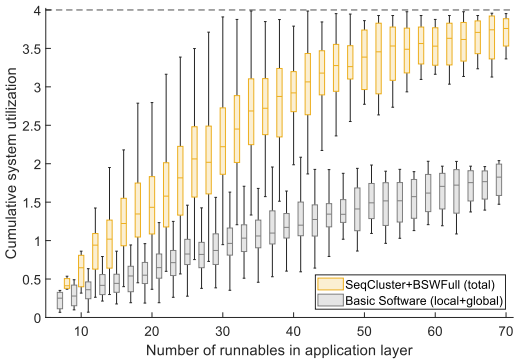
Figure 4 Utilization for diagnostic extremes ① and ② (non-compliant by construction). ① accounts only for global BSW tasks, while ② accounts only for task-local BSW extensions; hence, ②'s total utilization excludes global BSW tasks and is not directly comparable to full-accounting variants.



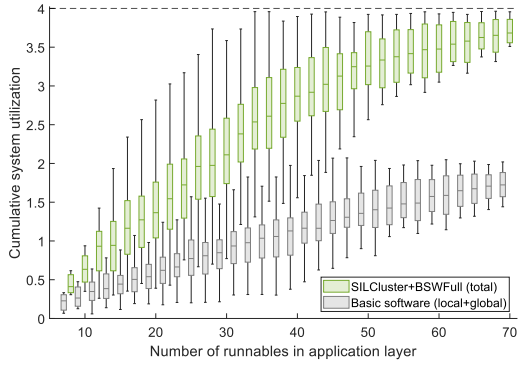
Ⓑ BSW overheads (NoSIL+NoJLD).



③ NoCluster+BSWFull (Baseline).

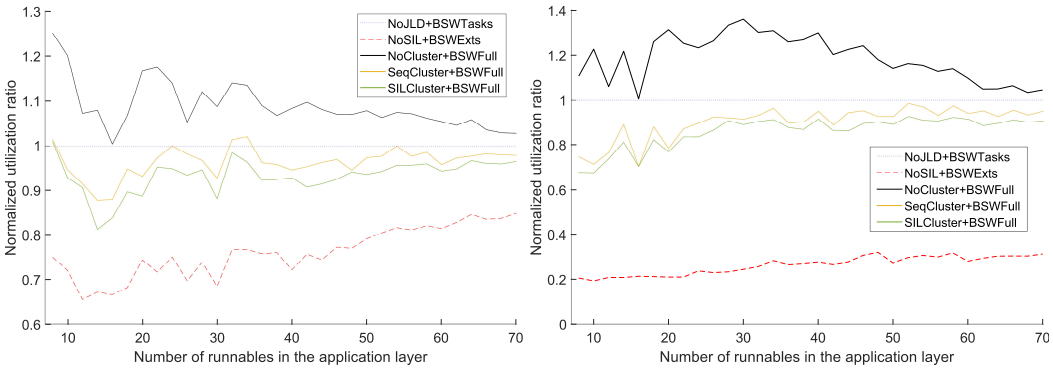


④ SeqCluster+BSWFull.



⑤ SILCluster+BSWFull.

Figure 5 Utilization for the BSW reference Ⓑ and feasible variants ③–⑤. Ⓑ disables SIL and data-age constraints to expose the complete generated BSW surrogate: task-local BSW extensions plus global BSW-task utilization, bounded by Eq. 14. ③ is the no-clustering baseline, while ④ and ⑤ progressively reduce pessimism via sequence-based clustering and SIL-aware clustering, respectively.



(a) Normalized Total Utilization. **(b)** Normalized BSW Utilization.

Figure 6 Utilization ratios normalized to variant ① (NoJLD+BSWTasks).

From Constraints to Reduced Pessimism. Our initial claim was that explicit BSW modeling and SIL/JLD-aware clustering reduce analysis pessimism and increase feasibility margins. How to read the plots: The absolute utilizations in Figures 4 and 5 are informative but dense. Variants ① and ② are diagnostic boundary cases, whereas ③–⑤ form the feasible full-accounting comparison set. The normalized ratios in Figure 6 provide the clearest view of relative gains and should be consulted alongside the absolute plots.

Global BSW tasks are a dominant driver: Variant ① combines global-only BSW accounting with disabled data-age enforcement and unmerged application tasks. In this diagnostic setting, global BSW tasks account for about 42 % utilization on average, with utilization increasing quickly as system size grows. This does not constitute a clean single-factor ablation, but it shows that global BSW is a first-order design concern. Since ① disables JLDs, it violates chain semantics by construction and serves only as a non-compliant diagnostic reference.

Task-local BSW extensions equally matter, but depend on runnable mapping: Variant ② quantifies task-local BSW extensions under JLD-enforced data age and merging, but it omits global BSW tasks and disables SIL constraints. Hence, its absolute utilization is *not* directly comparable to full-accounting variants; one must conceptually add the missing global BSW share exposed by ①. Nevertheless, ② shows that extensions remain non-negligible and that enforcing data-age constraints contributes substantially to overall utilization. The variant violates safety partitioning by construction and is included only as a non-compliant diagnostic boundary case.

Our BSW approximation remains bounded: Since ① and ② expose only one BSW channel each, Figure 5a reports the unconstrained BSW reference Ⓑ. It shows the complete generated BSW utilization, split into task-local extensions and global BSW tasks. The combined BSW utilization stays below the Lopez upper-utilization envelope by construction: Eq. 14 first reserves the task-local contribution Υ and then restricts global BSW tasks to the residual budget $\mathcal{U}_{ub} - \Upsilon$. Thus, Ⓑ does not independently validate the bound but makes the calibration of the early-phase BSW model transparent.

Full constraints and full BSW accounting increase pessimism: Variant ③ enables both SIL and data-age constraints and accounts for both BSW types, but keeps runnables one-per-task (no clustering). This represents a feasible yet pessimistic baseline and yields the highest total utilization among the feasible variants; the disproportionate growth in BSW utilization is most clearly visible in Figure 6b.

SIL/BSW-aware clustering progressively reduces pessimism: Variants ④ (SeqCluster+BSWFull) and ⑤ (SILCluster+BSWFull) progressively reduce pessimism by clustering/merging under full constraints and full BSW accounting. Figure 6 shows a monotone reduction

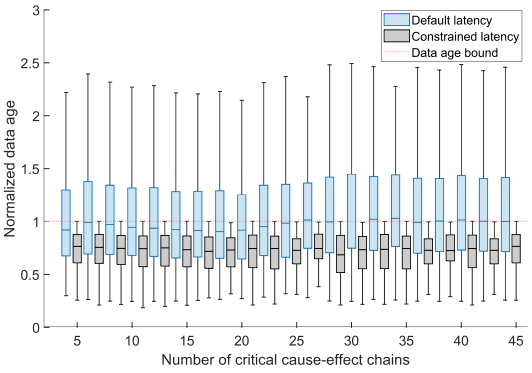


Figure 7 Normalized data ages across critical cause-effect chains (feasible variants).

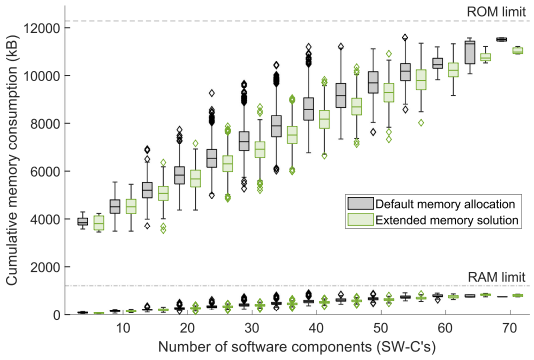


Figure 8 Cumulative ROM/RAM consumption induced by SW-Cs (per core).

in both total and BSW utilization from ③ to ④ to ⑤, with improvements becoming more pronounced as system size grows. Notably, the full approach (⑤) can achieve utilization below the diagnostic configurations that ignore either JLDs (①) or SIL constraints / global BSW tasks (②), indicating that the gains stem from improved task construction under realistic constraints rather than from relaxing the constraint set.

From this set of experiments, we conclude that BSW is a primary utilization driver and must be modeled explicitly; within that realistic setting, our SIL/BSW-aware clustering effectively reduces pessimism and recovers feasibility margins.

Data Age Impact. As a sanity check against established cause-effect chain analysis practice, Figure 7 plots, for each cause-effect chain, the ratio of computed end-to-end latency to its data-age bound (ratios ≤ 1 satisfy the constraint). In ① (NoJLD+BSWTasks), latencies are evaluated *without* enforcing JLDs; only about half of the chains meet their bounds, confirming that ① violates data-age requirements by construction and thus serves only as a diagnostic extreme. In contrast, all variants that enable JLDs (②–⑤) satisfy the data-age bounds by construction; we illustrate this with the feasible baseline ③ (NoCluster+BSWFull) in Figure 7. This confirms that our implementation of JLD-based enforcement behaves as expected, and that the utilization results discussed above are obtained under data-age compliant configurations.

Allocation and Memory Consumption. We propose that feasibility-relevant memory constraints must be considered alongside timing. Therefore, our experiments in Figure 8 report cumulative ROM/RAM footprints induced by SW-Cs, where each SW-C aggregates the budgets of its constituent runnables. We compared a default SW-C partitioning (one cyclic runnable per SW-C) against an extended partitioning (fewer SW-Cs hosting up to two runnables each), which improved memory efficiency by reducing per-component overheads. As the number and size of SW-Cs grew, both schemes approached the per-core resource limits of the target platform (Section 5.2). Global BSW memory was accounted for separately via the per-core envelope in Eq. 15. We reason that even when clustering reduces utilization, memory remains critical and must be enforced jointly with schedulability.

Reality Check. To assess practical effectiveness, we applied our full approach (⑤ SILCluster+BSWFull) to our real-world motion and drive controller in its early-design configuration (all constraints enabled). The end-to-end analysis completed in 2.43 h and produced a feasible mapping that respects SIL separation, data-age constraints, and per-core memory limits.

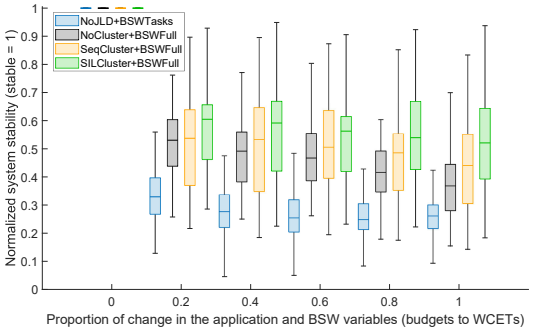


Figure 9 Assignment stability (0–100% budgets → WCETs) for variants ① and ③–⑤. Subset switched to WCETs is re-sampled independently.

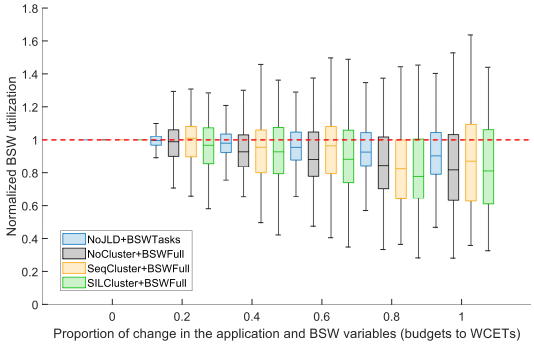


Figure 10 Normalized total utilization of variants ① and ③–⑤. Lower utilization implies larger headroom and less analysis pessimism.

We report results as ratios normalized to the controller's deployed configuration (deployed baseline = 1.0). In terms of *utilization*, our approach reduces total CPU load to 0.9325 (≈7 % improvement) and BSW utilization to 0.9278 (≈7 % improvement), indicating that SIL-aware clustering together with explicit BSW modeling can translate into tangible headroom in real-world development cycles. This check does not validate Eq. 13 as a universal model, but it confirms that the early-phase BSW abstraction yields a total BSW load in the same range as the deployed industrial configuration. In terms of *memory*, all cores remain close to capacity; ROM improves slightly to 0.9893, while RAM remains essentially unchanged at 0.9954. The framework yields meaningful utilization gains on a highly loaded production-grade automotive system, suggesting that it can provide actionable guidance when revisiting architectures across iterative integration and update cycles.

5.4 Stability and Architectural Churn

We assess how much of the *structural* allocation produced by our SIL/BSW-aware flow is retained when execution-time inputs are refined from early budgets toward WCETs.

Stability indicators. We quantify *architectural churn* using overlap-based *stability scores* (not a metric in the strict mathematical sense). A score of 1 means that the same tasks remain co-located on equivalent cores; a score of 0 means that no such co-location structure is retained. The underlying engineering notion is that the costly decision is the *co-location structure* of periodic tasks; reordering within a core (priorities) or swapping equivalent lockstep cores is comparatively cheap. Accordingly, we measure how well the original per-core task grouping is preserved. Let $\Gamma = \mathcal{T}' \cup \mathcal{B}$ denote the full periodic task set (application multiframe tasks and global BSW tasks), and let $\mathcal{K}$ be the set of lockstep cores. For a reference solution (at the initial budget stage), let $P_k \subseteq \Gamma$ be the set of tasks allocated to core κ_k; after refinement, let P'_k denote the corresponding set. We first compute a directional *fixed-label retention* score $S_{\text{fix}} = \frac{1}{|\mathcal{K}|} \sum_{\kappa_k \in \mathcal{K}} \frac{|P_k \cap P'_k|}{|P_k|}$, which answers: *how much of the initial allocation is retained on the same labeled core?* Since lockstep cores are interchangeable in our model, we additionally allow a relabeling of cores and compute the best matching between the per-core task sets: $S_{\text{perm}} = \max_\pi \frac{1}{|\mathcal{K}|} \sum_{\kappa_k \in \mathcal{K}} \frac{|P_k \cap P'_{\pi(k)}|}{|P_k|}$, where π ranges over permutations of the lockstep cores. In practice, we keep any unchanged cores with $P_k = P'_k$ fixed (they contribute a value of 1 in the average) and solve the remaining assignment as a maximum-overlap matching.

We report overall stability as $S = S_{\text{perm}}$ and define churn as $1 - S$. Thus, we do *not* penalize priority changes or core label swaps, but we *do* penalize changes in the co-located task sets that drive integration/configuration effort.

Refinement experiment. For each synthesized system, we start from conservative execution-time *budgets* (as in the previous experiments) and compute a reference allocation. We then progressively replace budget values by WCET samples for increasing fractions of execution-time parameters: $0\%, 20\%, 40\%, 60\%, 80\%$, and 100% (budgets $\rightarrow$ WCETs), and re-run the toolchain to obtain a refined allocation. Importantly, for each refinement level we re-sample *which* parameters are switched independently (e.g., the 20% subset is not necessarily contained in the 40% subset). This yields a deliberately conservative estimate of stability: unlike a realistic iterative workflow where measured values accumulate over time, our setup injects additional variation and thus tends to *maximize* observed churn.

Results. Figure 9 reports stability distributions over several thousand generated systems for variants ① and ③–⑤. Across all refinement levels, the feasible variants with full BSW accounting and clustering (③–⑤) consistently achieve higher stability than the diagnostic variant ①. In particular, ⑤ (*SILCluster+BSWFull*) shows the highest median stability and the tightest spread, indicating that SIL/BSW-aware clustering reduces the need for structural remapping as timing inputs change. Compared to ①, stability improves by roughly 20–30 percentage points (depending on refinement level), despite the conservative re-sampling.

Figure 10 complements this view by reporting normalized total utilization for the same refinement experiment. Values below 1 indicate that early budgets were conservative and that refining toward WCETs reduces execution-time pessimism; conversely, higher values imply less headroom. Variants ③–⑤ exhibit lower normalized utilization than ①, with ⑤ lowest overall. This additional headroom helps absorb timing refinements without forcing structural changes, which is consistent with the stability trends in Figure 9.

Taken together, these results support using our toolchain as a recurring architectural analysis step during budget-to-WCET refinement. The full SIL/BSW-aware variant does not eliminate architectural churn, but it consistently retains more of the initial co-location structure than less-informed variants even under aggressive resampling. We therefore interpret the stability experiment as evidence for the relative robustness of the structural decisions.

The absolute stability values should not be read as evidence that architectural churn disappears. Values around 0.5–0.6 still indicate substantial structural change under aggressive changes – we expect better results in actual use, since measured WCET information typically accumulates monotonically rather than being re-replaced by budgets. We therefore use the experiment primarily as a *relative* robustness test: under identical refinement and resampling conditions, SIL/BSW-aware clustering retains more of the initial co-location structure than less-informed variants.

6 Discussion and Conclusion

We presented a chain-aware analysis and optimization flow for automotive ECUs that treats SIL constraints, Basic Software (BSW) costs (task-local extensions and global BSW tasks), and memory as first-class citizens. Grounded in a real-world motion and drive controller, our case study revealed that SIL constraints and BSW overheads are tightly coupled to structural architectural decisions (task construction, clustering, and core allocation) and can dominate feasibility margins. Based on these empirical profiles, we derived a system generator

and implemented a heuristic toolchain that combines SIL-aware clustering, multirate task construction, and BSW- and memory-aware multicore allocation, with feasibility checks under precedence constraints.

Our evaluation demonstrates three key takeaways. First, explicit BSW accounting is essential: both global BSW tasks and task-local extensions materially influence utilization and thus feasibility margins, and treating BSW merely as a fixed margin can be misleading. Second, SIL/JLD-aware clustering reduces pessimism in analysis under full constraint sets, recovering substantial headroom compared to unclustered baselines. Third, beyond single-shot feasibility, our toolchain can be used repeatedly as timing information is refined: in the industrial configuration, our approach yields a 7 % utilization reduction, and our refinement study (budgets → WCETs) shows that the full SIL/BSW-aware variant preserves more of the initial co-location structure than less informed variants. Taken together, these results support using the approach not only for early architectural exploration, but also as a recurring decision-support step in iterative SDV development when timing parameters evolve more frequently than safety- and BSW-relevant architectural structure.

The companion case study [15] and the Driverator benchmark generator [14] are available as supplemental material.

References

1 AUTOSAR. Specification of Timing Extensions. R22-11, `https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_TPS_TimingExtensions.pdf`, 2022.

2 AUTOSAR. Specification of RTE Software. R24-11, `https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_RTE.pdf`, 2024.

3 I. Bate and P. Emberson. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, pages 221–230, 2006. `doi:10.1109/RTAS.2006.21`.

4 Matthias Becker. Meeting job-level dependencies by task merging. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC 2024)*, ASPDAC '24, pages 792–798. IEEE Press, 2024. `doi:10.1109/ASP-DAC58780.2024.10473901`.

5 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proceedings of the 22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2016)*, pages 159–169, 2016. `doi:10.1109/RTCSA.2016.41`.

6 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, October 2017. `doi:10.1016/j.sysarc.2017.09.004`.

7 Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte. A generic framework facilitating early analysis of data propagation delays in multi-rate systems (invited paper). In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017)*, pages 1–11. IEEE Computer Society, 2017. `doi:10.1109/RTCSA.2017.8046323`.

8 Alwin Berger, Simon Schuster, Peter Wägemann, and Peter Ulbrich. Dynamic Fuzzing-Based Whole-System Timing Analysis. In *Proceedings of the 46th IEEE Real-Time Systems Symposium (RTSS 2025)*, pages 420–433, December 2025. `doi:10.1109/RTSS66672.2025.00041`.

9 Anand Bhat, Soheil Samii, and Ragunathan Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. *Real-Time Systems*, 55(4):889–924, October 2019. `doi:10.1007/s11241-019-09339-7`.

10 Alan Burns and Robert Davis. Mixed criticality systems - a review : (13th edition, february 2022). Technical report, University of York, February 2022. Scholarly edition, 97 pages.

11 Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th Annual Design Automation Conference (DAC 2007)*, DAC '07, pages 278–283, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1278480.1278553.

12 R. Davis. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43, October 2011. doi:10.1145/1978802.1978814.

13 Robert Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of Systems Architecture*, 65, April 2016. doi:10.1016/j.sysarc.2016.04.002.

14 Tobias Denzinger, Matthias Becker, and Peter Ulbrich. Driverator: Generating Realistic Automotive Real-Time Task Sets with SIL and BSW Effects, 2026. doi:10.17877/TUDODATA-2026-MOR12ARE.

15 Tobias Denzinger, Matthias Becker, and Peter Ulbrich. Shedding light onto safety integrity level and basic software constraints in a real-world automotive application: Case study with driverator framework, 2026. arXiv:2605.04837.

16 Christian Dietrich, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper). In Gabriel Parmer, editor, *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2017)*, pages 37–48. IEEE Computer Society, 2017. doi:10.1109/RTAS.2017.37.

17 Alessandro Druetto, Enrico Bini, Andrea Grosso, Stefano Puri, Silvio Bacci, Marco Di Natale, and Francesco Paladino. Task and memory mapping of large size embedded applications over NUMA architecture. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems (RTNS 2023)*, RTNS '23, pages 166–176, Dortmund, Germany, 2023. Association for Computing Machinery. doi:10.1145/3575757.3593650.

18 Christian Eichler, Tobias Distler, Peter Ulbrich, Peter Wägemann, and Wolfgang Schröder-Preikschat. TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses. In Florian Brandner, editor, *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 6:1–6:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2018.6.

19 Paul Emberson. *Searching for flexible solutions to task allocation problems.* PhD thesis, University of York, 2009.

20 Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports*, 8:122–149, 2018. doi:10.4230/DagRep.8.2.122.

21 Nico Feiertag, Kai Richter, J. Eric Nordlander, and Jan Åke Jönsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Proceedings of the IEEE Real-Time Systems Symposium Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (RTSS Workshop 2008)*. IEEE, 2008.

22 Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, SAC '10, Sierre, Switzerland, 2010. doi:10.1145/1774088.1774196.

23 Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*, pages 301–310, 2010. doi:10.1109/RTAS.2010.26.

24 Florian Franzmann, Tobias Klaus, Peter Ulbrich, Patrick Deinhardt, Benjamin Steffes, Fabian Scheler, and Wolfgang Schröder-Preikschat. From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors. In *Proceedings of the 19th IEEE International Symposium on Real-Time Distributed Computing (ISORC 2016)*, pages 134–141, Washington, DC, USA, May 2016. IEEE. doi:10.1109/ISORC.2016.27.

25 Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Trans. Cyber-Phys. Syst.*, 5(2), January 2021. doi:10.1145/3381847.

26 Mario Günzel and Matthias Becker. Optimal task phasing for end-to-end latency in harmonic and semi-harmonic automotive systems. In *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2025)*, pages 164–176. IEEE, 2025. doi:10.1109/RTAS65571.2025.00026.

27 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.10.

28 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. Waters industrial challenge 2017. In *Proceedings of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS 2017)*, 2017. [Online].

29 Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005. doi:10.1049/ip-cdt:20045088.

30 INCHRON AG. chronsuite. https://www.inchron.com/chronsuite/.

31 International Organization for Standardization. ISO 26262-1:2018 Road vehicles – Functional safety – Part 1: Vocabulary. https://www.iso.org/standard/68383.html, 2018.

32 Fouad Khenfri, Khaled Chaaban, and Maryline Chetto. Efficient mapping of runnables to tasks for embedded autosar applications. *Journal of Systems Architecture*, 110:101800, 2020. doi:10.1016/j.sysarc.2020.101800.

33 Tobias Klaus, Matthias Becker, Wolfgang Schroder-Preikschat, and Peter Ulbrich. Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*, pages 66–79, May 2021. doi:10.1109/RTAS52030.2021.00014.

34 Tobias Klaus, Florian Franzmann, Matthias Becker, and Peter Ulbrich. Data propagation delay constraints in multi-rate systems: Deadlines vs. job-level dependencies. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS 2018)*, pages 93–103, 2018. doi:10.1145/3273905.3273923.

35 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS 2015)*, volume 130, 2015.

36 Jose M. López, M. García, Jose L. Díaz, and Daniel F. García. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, 24:5–28, 2003. doi:10.1023/A:1021749005009.

37 Luiz Maia and Gerhard Fohler. Reducing End-to-End Latencies of Multi-Rate Cause-Effect Chains in Safety Critical Embedded Systems. In *Proceedings of the 12th European Congress on Embedded Real Time Software and Systems (ERTS 2024)*, Toulouse, France, June 2024.

38 Claire Maiza, Hamza Rihani, Juan Rivas, Joël Goossens, Sebastian Altmeyer, and Robert Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys*, 52:1–38, June 2019. doi:10.1145/3323212.

39 Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. End-to-end latency characterization of task communication models for automotive systems. *Real-Time Systems*, 56:315–347, 2020. doi:10.1007/s11241-020-09350-3.

40 Roberto Medina, Etienne Borde, and Laurent Pautet. Generalized mixed-criticality static scheduling for periodic directed acyclic graphs on multi-core processors. *IEEE Transactions on Computers*, 70(3):457–470, March 2021. doi:10.1109/TC.2020.2990229.

41 Roberto Medina, Étienne Borde, and Laurent Pautet. Scheduling multi-periodic mixed-criticality dags on multi-core architectures. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 254–264, December 2018. doi:10.1109/RTSS.2018.00042.

42 A.K. Mok and D. Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS 1996)*, pages 22–29, 1996. doi:10.1109/REAL.1996.563696.

43 Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multisource software on multicore automotive ecus—combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, 2012. doi:10.1109/TIE.2012.2185913.

44 Johannes Schlatow, Mischa Mostl, Sebastian Tobuschat, Tasuku Ishigooka, and Rolf Ernst. Data-age analysis and optimisation for cause-effect chains in automotive control systems. In *Proceedings of the 13th IEEE International Symposium on Industrial Embedded Systems (SIES 2018)*, pages 1–9, 2018. doi:10.1109/SIES.2018.8442077.

45 Dario Socci. *Scheduling of certifiable mixed-criticality systems*. Theses, Université Grenoble Alpes, March 2016.

46 Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In *Proceedings of the 18th IEEE International Symposium on Real-Time Distributed Computing (ISORC 2015)*, pages 198–207, April 2015. doi:10.1109/ISORC.2015.18.

47 Yue Tang, Xu Jiang, Nan Guan, Dong Ji, Xiantong Luo, and Wang Yi. Comparing communication paradigms in cause-effect chains. *IEEE Transactions on Computers*, 72(1):82–96, 2023. doi:10.1109/TC.2022.3197082.

48 Vector Informatik GmbH. Ta tool suite. https://www.vector.com/int/en/products/products-a-z/software/ta-tool-suite/.

49 Alexander Wieder and Björn B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *Proceedings of the 8th IEEE Int. Symp. on Industrial Embedded Systems (SIES 2013)*, pages 49–58, 2013. doi:10.1109/SIES.2013.6601470.

50 Risheng Xu, Marvin Kühl, Hermann Von Hasseln, and Dirk Nowotka. Reducing overall path latency in automotive logical execution time scheduling via reinforcement learning. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems (RTNS 2023)*, RTNS '23, pages 212–223, Dortmund, Germany, 2023. doi:10.1145/3575757.3593658.

51 Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 161–170, 2007. doi:10.1109/RTSS.2007.40.

52 Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems*, 11(4), January 2013. doi:10.1145/2362336.2362352.