

# Managing Real-Time Constraints through Monitoring and Analysis-Driven Edge Orchestration<sup>★</sup>

Daniel Casini<sup>a,b</sup>, Paolo Pazzaglia<sup>c</sup> and Matthias Becker<sup>d</sup>

<sup>a</sup>TeCIP Institute, Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56124 Pisa (PI), Italy

<sup>b</sup>Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, Pisa 56127, Italy.

<sup>c</sup>Robert Bosch GmbH, Robert-Bosch-Campus 1, 71272 Renningen, Germany

<sup>d</sup>EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

## ARTICLE INFO

### Keywords:

Real-time systems  
Design-Space Exploration  
QNX  
Distributed Systems  
Edge Computing

## Abstract

Emerging real-time applications are increasingly moving to distributed heterogeneous platforms, under the promise of more powerful and flexible resource capabilities. This shift inevitably brings new challenges. The design space to deploy chains of threads is more complex, and sound estimates of worst-case execution times are harder to obtain. Additionally, the environment is more dynamic, requiring additional runtime flexibility on the part of the application itself. In this paper, we present an optimization-based approach to this problem. First, we present a model and real-time analysis for modern distributed edge applications. Second, we propose a design-time optimization problem to show how to set the main parameters characterizing such applications from a time-predictability perspective. Then, we present an orchestration and runtime decision-making mechanism that monitors execution times and allows for runtime reconfigurations, spanning from graceful degradation policies to re-distributions of workload. A prototypical implementation of the proposed approach based on the QNX RTOS and its evaluation on a realistic case study based on an edge-based valet parking application conclude the paper.

## 1. Introduction

The landscape of applications of interest for real-time systems is rapidly evolving. Some of today's emerging scenarios differ considerably from what the research community worked on in the past. First, the multi-core revolution caused a paradigm shift from single-core platforms, introducing considerable challenges, such as quantifying memory-contention delays at DRAM, bus, and cache level [36, 57], which need to be considered even when applications are partitioned to cores mimicking single-core scheduling. Second, real-time applications typically run on complex Commercial-Off-The-Shelf (COTS) platforms, featuring heterogeneous cores with different speed profiles, powerful hardware accelerators, and complex memory hierarchies that strongly enhance the average-case performance. However, executions occurring on this hardware are largely

not predictable, and typically employ scheduling policies not disclosed by the vendors [2, 36].

These aspects point out a remarkable difference with respect to classical single-core real-time systems: the (almost) impossibility of relying on sound worst-case execution time (WCET) estimates. The availability of sound WCETs is an ordinary key assumption in single-core systems, that allows for decoupling the scheduling effects – typically accounted for with the worst-case response time (WCRT) – from the execution time experienced by a certain computational activity (thread) in isolation. WCETs can be precisely bounded for application-specific platforms (e.g., in avionics) by leveraging static analysis tools [35], but it is undoubtedly much harder in COTS platforms and, nowadays, is not standard practice for many application scenarios. In many cases, pseudo-WCETs – better called *worst-observed execution times* (WOETs) – are obtained by profiling the application in different conditions and possibly inflating the measured execution time by some engineered safety percentage. But estimates can be inaccurate, especially for complex real-time applications interconnected in highly distributed systems, such as those in the increasingly popular IoT-to-Edge-to-Cloud Compute Continuum, in which the software can be incredibly complex and possibly subject to software vulnerabilities. The causes of inaccuracies in execution time estimates may be multiple, including the application program being originally tested in a slightly different platform, or the presence of data-dependent code paths that have not been covered during testing, or rare boundary conditions in iterative optimization problems (like the ones required for Model Predictive Control applications), as well as bugs and cyber attacks. The frequency, in terms of the number of

<sup>★</sup>The work has been partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU and the European Union's Horizon Europe Framework Programme project NANCY under grant agreement No. 101096456 and by the Swedish Research Council (VR) under the project nr. 2023-04773, by Sweden's Innovation Agency via the Advanced and innovative digitalization project 2021-02484 EARLY BIRD Seamless System Design from Concept Phase to Implementation and by Digital Futures via the project GPARSE. This research was also supported by the Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, Pisa 56127, Italy. We thank BlackBerry QNX for supplying relevant software.

\*All authors are corresponding.

 daniel.casini@santannapisa.it (D. Casini);

paolo.pazzaglia@de.bosch.com (P. Pazzaglia); mabecker@kth.se (M. Becker)

ORCID(s): 0000-0003-4719-3631 (D. Casini); 0000-0003-0377-3327 (P. Pazzaglia); 0000-0002-1276-3609 (M. Becker)

Paper	Chains	Distributed	MultiMode	Degr./Adapt.	Monitoring	Optimization/Orchestration	Target OS/Context
Zhu et al. [70]	YES	Distr. ECUs	NO	NO	NO	Design-time Optimization	AUTOSAR Classic
Bate et al. [7]	YES	Distr. ECUs	NO	Scenario-based	NO	Heuristics	Automated Design
Struhar et al. [64]	NO	Edge/Fog	NO	Reservation Parameters	YES	Static Opt. + Orchestration	Linux
Kritikakou et al. [42]	NO	Distr. ECUs	Dual	Susp. Low Crit.	YES	NO	Bare-metal
Guo et al. [33], Yang et al. [67]	NO	NO	Dual	Reduce Perf. Low Crit.	YES	NO	Theory-Mixed-Crit.
Gifford and Phan [29]	NO	NO	Arbitrary	NO	YES	Task Redistribution	Multi-core
Lakshmanan et al. [44]	NO	Distr. ECUs	Dual	NO	NO	Design-time Optimization	Mixed-criticality
Monaco et al. [48]	NO	Edge/Cloud	NO	Reduce Alloc. Resources	YES	Orchestration only	Kubernetes
Durrieu et al. [27]	NO	Distr. ECUs	Arbitrary	Adaptation Tables	YES	Reconfiguration tables	Avionics
Stankovic et al. [63]	NO	Compatible	NO	NO	NO	NO	Theory-Control
Papadopoulos et al. [53]	NO	NO	Dual	Control and Reservations	YES	NO	Control-VestalMCS
Burns and Baruah [13]	NO	NO	Dual	Param. Change Low Crit.	YES	NO	Control-VestalMCS
Sinha et al. [62]	NO	NO	Dual	Reservation Budget	YES	NO	Vestal-LitmusRT
Kritikakou and Skalistis [43]	NO	NO	Dual	Drop Low. Crit.	YES	NO	Vestal
Zou et al. [71]	YES	NO	Dual	Drop Low. Crit.	YES	NO	Functional Dependencies
Peeck et al. [56]	YES	NO	NO	Not discussed	YES	NO	Automotive
Schlatow et al. [59]	YES	NO	NO	Budget Enforcement	YES	NO	Genode
Kampmann et al. [40]	YES	NO	NO	Not discussed	YES	Orchestration	Service-oriented
<i>This Paper</i>	<b>YES</b>	<b>Edge</b>	<b>Arbitrary</b>	<b>Modular + Degr. Order</b>	<b>YES</b>	<b>Static Opt. + Orchestration</b>	<b>RTOS (POSIX-QNX)</b>

**Table 1**

Comparison of a selection of related papers.

affected jobs in a time interval, of such WOET overshoots is hard to predict and can be harmful even in case they are transitory, as they might nonetheless last long enough to disrupt the functional behavior, especially during critical working conditions. Furthermore, the unreliability of threads' WCET estimates also makes the thread-to-cores (and thus threads-to-platforms) allocation problem harder, as it can only rely on initial estimates that may turn out to be inaccurate or wrong at runtime.

To face this problem in the context of *soft* real-time systems –considered hereafter– we propose to manage these sources of unpredictability by *monitoring* execution times and providing *analysis-driven runtime decision-making and orchestration* mechanisms. This is used to re-distribute the workload in case of core or platform overloads, i.e., if the system becomes unschedulable when running the analysis again with an updated (larger) execution time estimate. Furthermore, orchestration can also integrate the need to manage the dynamic and open nature of such systems, where new threads and new computing platforms may join (or leave) the system at runtime [17].

When the workload increases, it is however possible that the available platforms can simply be not enough to accommodate it while meeting timing requirements. Then, a key question arises: how to enforce a *graceful degradation* of performance for some threads to guarantee the timing constraints of the overall thread set? For example, it could be allowed to increase some thread periods and deadlines, switch some threads to a faster –but less accurate– operating mode, or kill some low-critical threads. Another dilemma concerns the order in which threads are to be degraded. These questions necessarily involve both the design and orchestration aspects of the application. The problem becomes even more complex for chains of communicating threads, where degrading a thread may have unforeseen consequences on the chains it belongs to.

**Contributions.** These issues have no universal solution since the preferred choice can change case by case. Therefore, this paper proposes to tackle them in a modular and

configurable manner by means of a set of (tunable) *reaction policies* to cope with overloads while guaranteeing that the degradation process follows a *degradation order* specified by the designer.

We pursue this goal through the following steps, which establish the contribution of our paper: **(i)** we provide a model for a distributed system where communicating threads form processing chains, which is amenable to real-time analysis and includes multiple operating modes to support graceful degradation; **(ii)** we derive a response-time analysis for the presented model; **(iii)** we design an optimization problem for the initial allocation of the distributed application; **(iv)** leveraging (i), (ii), and (iii), we design the reaction policies based on modified versions of the optimization problem at (iii). *Importantly, the designer can configure our reaction policies by specifying a degradation order that the runtime-decision-making algorithm must follow.* To show the viability of our approach, we further **(v)** implement a prototype for a small-scale edge system, considering the QNX real-time operating system (RTOS), which seamlessly allows following the proposed model by leveraging *automatic code generation* from a modified Amalthea APP4MC model [39], and **(vi)** evaluate the proposed approach experimentally.

## 2. Related Work

The problem tackled in this paper spans multiple popular research topics, which we briefly cover in the following.

Firstly, the subject of optimal partitioning of task chains onto multiple heterogeneous platforms has sparked many research publications. Since the problem is known to be NP-hard [5], most works approach it either using heuristics or (M)ILP formulations (see, e.g., [70]). Another topic related to this paper regards mixed-criticality real-time tasks. The seminal work is due to Steve Vestal [66]. According to this model, the system can execute in two criticality modes: at the beginning, the lowest criticality mode is selected. Each criticality mode is characterized by an expected execution time for each task: if any task exceeds that estimate, the

system transitions to the higher criticality mode, in which all low criticality tasks do not execute. Many works targeted this design, extending it in several directions, see, e.g., [14, 42].

Burns and Baruah [13] presented several degradation options to make Vestal’s model more practical for mixed criticality [66], focusing on dual criticality systems. They investigate the change in priority, reduction of execution time budget, increment of period, and capacity inheritance, for low criticality tasks. However, their focus is on single-processor systems only and degradation policies only penalize low criticality threads. Davis et al. [25] proposed the Compensating Adaptive Mixed Criticality (C-AMC) scheduling scheme, which allows executing degraded versions of low-criticality tasks, characterized by a shorter execution time, to overcome the increase of load due to higher-criticality tasks executing more than expected, while still providing some essential functional behavior for low-criticality tasks. Sinha et al. [62] proposed a monitoring-based approach for Vestal’s dual criticality systems [6, 66] to avoid mode switches by monitoring the progress of each thread and dynamically changing its budget. An implementation in Litmus-RT is also provided. Similar approaches are also proposed by Kritikakou and Skalistis [43] to avoid mode switches by using slack time. Zou et al. [71] presented an approach for Vestal’s dual criticality systems that considers the functional dependencies among threads to derive a degradation process that discards computational load at the task level (rather than dropping all low-criticality tasks simultaneously).

Other works in literature attempted to provide graceful degradation to lower criticality tasks even when the system switches to the high criticality execution mode. Relevant works on this topic, such as [33, 67], however, do not consider chains of tasks and distributed systems and support only two modes (dual-mode). Other approaches for graceful degradation employ adaptive mechanism [31, 61], mainly focusing either on fault tolerance [31] or control performance [61]. Examples of adaptations of real-time systems subject to temporal violations are also rate adaptation [16], the elastic model [15], and the feedback-scheduling of control tasks [63], which however do not consider multi-mode tasks or runtime optimization/reallocation strategies.

Providing adaptability to distributed systems is a topic common to several works in literature. Bate et al. [7] considered improving the flexibility of distributed real-time systems by predicting how the system’s state can evolve, leveraging a set of predefined scenarios. Durrieu et al. [27] considered multi-moded avionics applications proposing the usage of adaptation tables, which list the applications to be interrupted in case of an overload. Reallocations can also occur at runtime in case of a permanent core failure, based on fixed reconfiguration tables. Gifford and Phan [29] considered multi-mode applications in the context of multi-core systems, proposing resource allocation algorithms that support dynamic cache and memory bandwidth allocation, together with a schedulability analysis for the mode transition. Struhar et al. targeted the orchestration of containers on Linux [64]. This work differs from our proposal since

they do not provide a guaranteed degradation order and since we focus on small-scale edge systems running an RTOS, which are broadly different from large-scale systems running general-purpose OSes. Papadopolous et al. [53] presented a control-theoretic method to achieve resilience in a dual criticality system using a resource reservation mechanism coupled with a feedback control loop that monitors execution time overruns. Different from this paper, the optimization of deadlines, modes, and allocation is not addressed.

Another branch of related papers has been published over the recent years under the name of self-awareness, targeting the unpredictability of execution times in modern high-performance platforms addressed by monitoring and autonomous orchestration [50, 51]. One of the first works [51] describes the idea of using self-aware resource managers in the Internet-of-Things to gain benefits in terms of application accuracy and power consumption. This concept is elaborated further in [50] presenting the controlling current change - CCC - and information processing factory - IPF - approaches (coming from corresponding research projects): the first follows an analytical approach with contracting and formal system analysis, and the latter uses incremental changes and feedback control. However, both works just discuss the issues and solutions at a high level of abstraction. Later, Mostl et al. [49] extended the concepts provided in CCC from the safety-related systems engineering point of view. Schlatow et al. [60] further expanded the concept by presenting a framework to enable modeling and automating integration activities, which include automated decision-making. However, decision-making policies are not discussed in detail.

Along this research direction, the closest to our work are [40, 56, 59]. Peeck et al. [56] presented a decentralized monitoring concept supporting the supervision of thread chains at runtime, which allows for the detection of latency-bound violations at runtime. However, it detects violations by directly measuring latencies and does not use real-time analysis. Furthermore, it does not consider multiple operating modes. Also, reaction policies and optimization are not addressed. Schlatow et al. [59] leveraged the concepts of self-aware computing for model-implementation incoherence of CPU scheduling. To this end, a new reservation policy with an event-based replenishment policy is proposed. The reservation policy works with a budget and overhead monitoring system. Optimization and orchestration are not addressed. Kampmann et al. [40] presented a concept for a service-oriented architecture allowing dynamic runtime reconfiguration with a corresponding implementation stack and web-based architecture design tool. In this work, the architecture includes an orchestrator to control the interaction among services at runtime. However, orchestration policies are not discussed in detail because the focus is more on architecture definition and implementation.

Still related to this paper, other works targeting the optimization of real-time systems are [7, 19–21, 69, 70].

A selection of related works is compared in Table 1, which classifies each paper accounting for: **(i)** the consideration of chains of communication threads, **(ii)** the type of distributed system (automotive distributed Electronic Control Units - ECUs, edge, etc.), **(iii)** the consideration of multi-moded threads, **(iv)** the degradation and adaptation policies, **(v)** the presence of monitoring mechanisms, **(vi)** the consideration of design-time optimization or runtime orchestration strategies, and **(vii)** the considered OS or context.

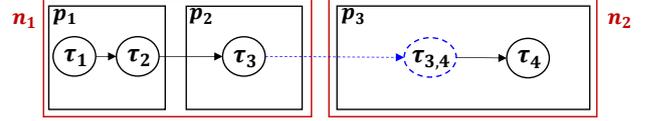
Overall, none of them targeted the optimization of a distributed multi-mode system while providing a modular set of reaction policies with the key property of guaranteeing to satisfy a provided degradation order.

### 3. System Model

**Platform and Threads.** This paper considers a distributed system with a set  $\mathcal{N}$  of interconnected nodes, i.e., computing platforms. Each node  $n_x \in \mathcal{N}$  consists of a heterogeneous multicore platform. Each platform comprises a set  $\mathcal{P}_x$  of cores, where each core  $p_k \in \mathcal{P}_x$  can have a different speed profile. The set of all cores is denoted hereafter as  $\mathcal{P} = \bigcup_{n_x \in \mathcal{N}} \mathcal{P}_x$ .

The system runs a workload composed of a set  $\mathcal{T}$  of real-time threads. For each core  $p_k \in \mathcal{P}$ , the set  $\mathcal{T}_k \subseteq \mathcal{T}$  denotes the threads allocated to core  $p_k$ . Each thread  $\tau_i \in \mathcal{T}_k$  has  $\mathcal{M}_i = \{1, \dots, d, \dots\}$  execution modes. A thread  $\tau_i$  adopting its  $d$ -th execution mode is denoted with  $\tau_i^d$ . When the execution mode is not relevant or clear from the context, we simply use  $\tau_i$ . An orchestrator selects the currently active mode for each thread. In each mode, threads can be characterized by different parameters, such as execution times, deadlines, and periods, modeled in the following. A lower mode index identifies an operational mode that provides a more accurate result for some metrics of interest. Each  $\tau_i^d$  is characterized by a worst-observed execution time (WOET)  $e_{i,k}^d$  that depends on the core  $p_k$ . Threads are periodic: each thread releases a potentially infinite sequence of instances (called jobs), with period  $T_i^d$ . Each thread is characterized by a constrained deadline  $D_i^d \leq T_i^d$ , meaning that each of its instances needs to be completed within  $D_i^d$  time units from its release. We consider a discrete model of time, where a time unit is an integer multiple of some basic units (e.g., a processor cycle).

On each core, threads are scheduled according to a *partitioned fixed-priority scheduling policy*, where each thread is assigned a unique priority  $\pi_i$  and is allocated to only one core. For simplicity, the priority is assumed to be independent of the mode. Each thread  $\tau_i$  is also associated with a unique *criticality index*  $c_i$  (the lower, the more critical). This index is an integer that identifies a degradation order specified by the system designer, which the orchestrator follows in its graceful degradation decisions (a thread with a high value of  $c_i$  will be picked as a candidate for being degraded before another thread with a lower value). Unlike most state-of-the-art works that target only dual-criticality threads, our model allows obtaining a more refined order of



**Figure 1:** A chain  $\gamma = (\tau_1, \dots, \tau_4)$  spanning two nodes and three cores. The copy-in thread  $\tau_{(3,4)}$  is introduced to restore the shared-memory communication across nodes.

criticality levels. Also, in this formulation the criticality does not pose any constraint on the assigned priority.

**Communication Model.** Threads are characterized by communication dependencies modeled by means of a direct acyclic graph (DAG), where vertexes encode threads and edges encode communication dependencies among them. Threads allocated to the same platform communicate using a shared-memory buffer. Communication between threads allocated to different platforms occurs by means of message-passing primitives in a data-driven manner involving the network. When using data-driven communication, the so-called *sender thread* (i.e., the producer) calls a function to send the data to the consumer thread through the network stack. Under this paradigm, the reception of a message from a thread involves the release of a corresponding instance.

To maintain consistent shared-memory communication across all threads and reduce jitter propagation effects [37], the data-driven reception of data is managed by a dedicated *copy-in* thread, which is in charge of receiving (e.g., by issuing a blocking system call) the data and copying it into a portion of memory, to make it available to the actual receiving thread. An example is shown in Fig. 1. In this way, functional threads always read data according to a full shared-memory communication paradigm, with the advantage of avoiding the introduction of computationally intensive additional interfering instances of functional threads in the analysis (which contribute to increasing the pessimism [37]). Such an approach is also easily extensible to the Logical Execution Time (LET) [38] paradigm (which is left to future work).

The set of copy-in threads allocated on a core  $p_k$  is denoted with  $\mathcal{T}_k^{\text{cp}}$ , and  $\mathcal{T}^{\text{cp}}$  denotes the set of all copy-in threads, irrespective of the core in which they are allocated to (with  $\mathcal{T}^{\text{cp}} \cap \mathcal{T} = \emptyset$ ). For simplicity, we map each pair of threads communicating this way to a distinct copy-in thread. When it is relevant to denote the producer thread  $\tau_i^d$  and the corresponding consumer thread  $\tau_j^q$  related to a given copy-in thread, we denote the copy thread as  $\tau_{(i,j)}^d \in \mathcal{T}_k^{\text{cp}}$ , using a tuple for the thread index; we simply use a single index (e.g.,  $\tau_f^d \in \mathcal{T}_k^{\text{cp}}$ ) when the relation to the producer and consumer is irrelevant or clear from the context. Each copy-in thread  $\tau_{(i,j)}^d \in \mathcal{T}_k^{\text{cp}}$  is characterized by the execution time  $e_{(i,j),k}$ , priority  $\pi_{(i,j)}$ , and period  $T_{(i,j)}^d = T_i^d$ . The execution time  $e_{(i,j),k}$  of the copy-in thread corresponds to the time required to copy the data to the memory of the receiving thread. We reasonably assume that the amount of data exchanged among threads is independent of the mode; hence, also,

$e_{(i,j),k}$  does not depend on the mode. Each copy-in thread inherits the mode index of the corresponding producer thread since it is activated by messages sent by it and inherits the same (mode-dependent) period. However, despite being still characterized by a period, the data-driven activation introduces an activation jitter that must be accounted for in the analysis [37], as discussed next.

Threads in DAG without in-/out-going edges are named *source* and *sink*, respectively. A chain  $\gamma_x = (\tau_f, \dots, \tau_l)$  is a path in the DAG starting with a source thread and ending with a sink. Each chain is characterized by an end-to-end deadline  $D_{\gamma_x}$ . The set of all chains is denoted as  $\Gamma = \{\gamma_1, \dots, \gamma_a\}$ . Depending on the thread-to-node allocation, the chain can contain copy-in threads. The subset of computational threads and copy-in threads of a chain  $\gamma_x$  are referred to as  $\gamma_x^{\text{exe}}$  and  $\gamma_x^{\text{cp}}$ , respectively. The symbol  $\mathcal{PC}$  denotes the set containing all the pairs  $(\tau_i, \tau_j)$  of communicating threads. Communicating threads on different nodes are further characterized by a worst-observed communication delay  $\lambda_{(i,j)}^{x,y}$  from  $\tau_i \in \mathcal{T}$  to  $\tau_{(i,j)} \in \mathcal{T}^{\text{cp}}$ , when  $\tau_i$  and  $\tau_j$  are allocated on nodes  $n_x \in \mathcal{N}$  and  $n_y \in \mathcal{N} \setminus n_x$ , respectively. Communication delays (e.g., due to accesses to memory) are included in WOETs for communicating threads in the same node.

**Definitions.** Before proceeding, we recall some useful definitions. A job is said to be *pending*, from the time instant when it is released to when it completes. The worst-case response time (WCRT) of an arbitrary thread is defined as the longest time span elapsed between the release and the completion of any of its jobs for any possible schedule that complies with the system model. The same thread in different modes can have different WCRTs:  $R_i^d$  is then a WCRT upper bound for  $\tau_i^d$ .<sup>1</sup> The schedulability analysis presented in the following consider each thread to be in a fixed mode during the analysis interval. The symbol  $\bar{m}$  denotes the vector of modes adopted by each thread during the analysis interval. The thread set is said to be schedulable if the condition  $R_i^d \leq D_i^d$  holds for all threads  $\tau_i^d \in \mathcal{T}$  considering modes in  $\bar{m}$ . The worst-case end-to-end latency  $L_{\gamma_x}$  of a chain  $\gamma_x = (\tau_f, \dots, \tau_l)$  is the longest timespan elapsed between the release of any job of  $\tau_f$  and the completion of the job of  $\tau_l$  that consumes the output propagated from that job of  $\tau_f$ , for any possible schedule that complies with the system model. The chain set is schedulable if  $L_{\gamma_x} \leq D_{\gamma_x}$  for all  $\gamma_x \in \Gamma$ . The entire system is said to be schedulable if all threads and chains are schedulable.

The set of computational (i.e., non-copy-in) and copy-in threads with a priority higher than any given thread  $\tau_i$  on core  $p_k$  are denoted with  $\text{hp}_k^{\bar{m}}(\tau_i)$  and  $\text{hp}_k^{\bar{m},\text{cp}}(\tau_i)$ , respectively.  $\text{hp}(\tau_i)$  and  $\text{hp}^{\text{cp}}(\tau_i)$  denote the sets of higher-priority threads irrespective of the allocation.

**Mode Transitions.** To keep the analysis simple, we do not consider the transient effects due to mode changes [58, 65], which are generally negligible for soft real-time systems and

<sup>1</sup>The WCRT also depends on the modes of the interfering threads and on the core  $p_k$ : for brevity, we omit the corresponding indexes from  $R_i^d$ .

Symbol	Description
$\mathcal{N}$	Set of interconnected nodes
$n_x$	A specific node in $\mathcal{N}$
$\mathcal{P}$	Set of all cores across all nodes
$\mathcal{P}_x$	Set of cores in node $n_x$
$p_k$	A specific core in $\mathcal{P}_x$
$\mathcal{T}$	Set of real-time threads in the system
$\mathcal{T}_k$	Set of threads allocated to core $p_k$
$\mathcal{T}^{\text{cp}}$	Set of all copy-in threads across all cores
$\mathcal{T}_k^{\text{cp}}$	Set of copy-in threads allocated on core $p_k$
$\mathcal{PC}$	Set of all pairs of communicating threads
$\tau_i$	A specific thread in $\mathcal{T}_k$
$\pi_i$	Priority of thread $\tau_i$
$c_i$	Criticality index of thread $\tau_i$
$\mathcal{M}_i$	Set of execution modes for thread $\tau_i$
$\tau_i^d$	Thread $\tau_i$ in its $d$ -th execution mode
$e_{i,k}^d$	WOET of thread $\tau_i^d$ on core $p_k$
$T_i^d$	Period of thread $\tau_i^d$
$D_i^d$	Deadline of thread $\tau_i^d$
$R_i^d$	WCRT of thread $\tau_i^d$
$\tau_{(i,j)}^d$	Copy-in thread for prod. $\tau_i^d$ and cons. $\tau_j^d$ , $d \neq q$
$e_{(i,j),k}^d$	WOET of copy-in thread $\tau_{(i,j)}^d$ on core $p_k$
$T_{(i,j)}^d$	Period of copy-in thread $\tau_{(i,j)}^d$
$\Gamma$	Set of all chains in the system
$\gamma_x$	A specific chain in $\Gamma$
$\gamma_x^{\text{exe}}$	Subset of computational threads in chain $\gamma_x$
$\gamma_x^{\text{cp}}$	Subset of copy-in threads in chain $\gamma_x$
$\lambda_{(i,j)}^{x,y}$	Worst-observed comm. delay from $\tau_i$ to $\tau_{(i,j)}$
$L_{\gamma_x}$	Worst-case end-to-end latency of chain $\gamma_x$
$D_{\gamma_x}$	End-to-end deadline of chain $\gamma_x$
$\bar{m}$	Threads' modes in the analysis interval
$\text{hp}_k^{\bar{m}}(\tau_i)$	Set of HP threads than $\tau_i$ on $p_k$
$\text{hp}_k^{\bar{m},\text{cp}}(\tau_i)$	Set of HP copy-in threads than $\tau_i$ on core $p_k$
$\text{hp}(\tau_i)$	Set of HP threads irrespective of allocation
$\text{hp}^{\text{cp}}(\tau_i)$	Set of HP copy-in threads irres. of allocation

**Table 2**

Symbols presented in the system model.

that can be accounted for with state-of-the-art methods [4, 58]. Instead, we focus on steady-state schedulability.

**Summary of Symbols.** Table 2 summarizes the symbols introduced in the system model.

## 4. Proposed Approach

The proposed analysis-driven allocation and runtime decision-making scheme consists of three main phases.

**1. Initial design.** In the initial phase, we perform the design of the distributed application. A *mixed-integer linear programming* (MILP) formulation, presented in Section 6, is used to satisfy timing constraints (for both threads and chains), while assigning threads to cores (and hence to nodes) and deciding the operational mode of each thread based on the analysis in Section 5. The MILP relies on initial estimates of the WOETs to provide an initial allocation.

**2. Monitoring.** In our architecture, shown in Fig. 2, computing nodes report the execution times of monitored threads to

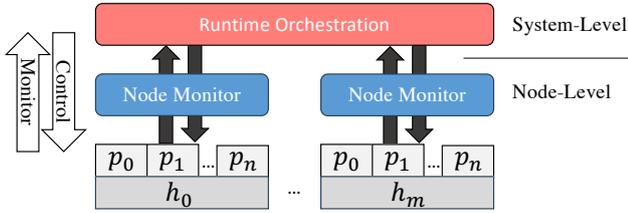


Figure 2: Monitoring and decision-making infrastructure.

the *orchestrator*, a separate node in charge of the decision-making process to guarantee timing constraints. Individual non-orchestrator nodes maintain a record of the current WOET value for each thread and core type. Every time a thread instance executes, the node keeps track of the observed execution time (discounting preemptions) and, at the end of the instance, compares it with the previous WOET record. The orchestrator is then informed about any violations of the WOET detected by the nodes. The orchestrator maintains a global WOET record that is used in reallocation decisions.

**3. Orchestration and runtime decision-making.** The initial allocation can be (partially) modified at runtime. Our orchestration and runtime decision-making architecture consists of a set of *trigger events* and *reaction policies*, as shown in Fig. 3. Trigger events give rise to the need for a new configuration produced by the orchestration and runtime decision-making module. For example, the notification of a WOET update, the arrival/exit of a thread, and horizontal scaling (i.e., the availability of a newly connected platform) are important events for triggering changes in the configuration. This paper *focuses on WOET updates* but remains compatible with the others. Since applications managed by an edge-distributed system can be very diverse, there is no universal solution on how to react to such events. Therefore, we propose a set of *modular* reaction policies: *mode relaxation*, *deadline inflation*, *reallocation*, and *thread discarding*. The designer can enable an arbitrary subset of such policies and specify an order among them: for example, if some new WOET updates make the current parameter assignment infeasible (when running the schedulability test in Section 5), in some cases it could be preferred to first try changing the allocation, while in others to restore schedulability by inflating deadlines, or by switching some threads to a degraded execution mode.

The specification of the reaction policies is detailed in Section 7. Among them, deadline inflation, mode relaxation, and thread discarding imply a (possibly temporary) *degradation* of the current service. The key property of our orchestration and decision-making process consists of ensuring a well-defined *degradation order* that the designer can configure in the orchestrator, by setting which policies should be adopted first and the order in which the threads are considered in the decisions. This is done, for example, by leveraging the threads' criticalities, or interleaving the threads and policies order, in such a way that the designer

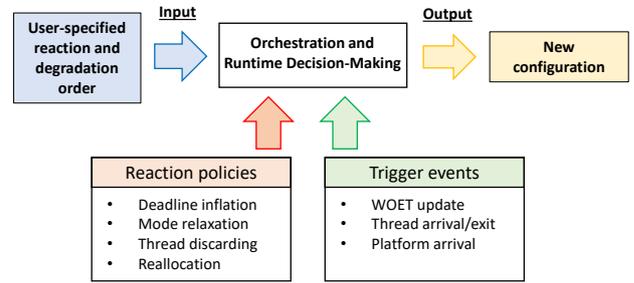


Figure 3: Orchestration and runtime decision-making.

can specify which are the threads to be sacrificed (and how) to favor highly critical ones.

## 5. Schedulability Analysis

The approach of our paper relies on a proper response-time analysis for constrained-deadline threads, developed for the system model of Section 3. We consider an arbitrary thread  $\tau_i$  allocated to a core  $p_k \in \mathcal{P}$  under partitioned fixed-priority scheduling. The classical schedulability test [45] under these assumptions checks the following condition:

$$\exists t \in [0, D_i] \mid W_i(t) \leq t \quad (1)$$

where  $W_i(t)$  is a function upper-bounding the overall processing time required by a thread  $\tau_i$  and all the threads that can possibly interfere with  $\tau_i$  in any window of length  $t$ . If Eq. (1) holds for a given time  $t'$  in the interval, then  $W_i(t')$  is a valid WCRT bound for  $\tau_i$ . The schedulability test of Eq. (1) is general and can be instantiated under different scheduling schemes providing an appropriate instance of  $W_i(t)$ .

In this paper, threads are characterized by multiple operating modes. Considering an arbitrary thread  $\tau_i^d$  (in mode  $d$ ), the worst-case response time can be bounded as:

$$R_i^d = \min_{t \geq 0} \left\{ e_{i,k}^d + I_{i,k}^{\bar{m}}(t) \mid e_{i,k}^d + I_{i,k}^{\bar{m}}(t) \leq t \right\}, \quad (2)$$

where  $e_{i,k}^d$  is the WOET of thread  $\tau_i^d$  executing on  $p_k$  in mode  $d$ , and  $I_{i,k}^{\bar{m}}(t)$  represents an upper-bound on the amount of interference on the execution of  $\tau_i^d$  from threads in core  $p_k$  with priority higher than  $\pi_i$ , released in an arbitrary window of length  $t$ , and with modes  $\bar{m}$ .  $I_{i,k}^{\bar{m}}(t)$  is computed as:

$$I_{i,k}^{\bar{m}}(t) = \sum_{\tau_h^q \in \text{hp}_k^{\bar{m}}(\tau_i)} \left\lceil \frac{t}{T_h^q} \right\rceil e_{h,k}^q + \sum_{\tau_{(f,g)}^s \in \text{hp}_k^{\bar{m}, \text{cp}}(\tau_i)} \left( \left\lceil \frac{t}{T_f^s} \right\rceil + 1 \right) e_{(f,g),k} \quad (3)$$

The first addend of Eq. (3) represents the interference from the computational threads with priority higher than  $\pi_i$ , while the second addend is the interference from the copy threads. Regarding the second term, since the activation of copy threads is triggered by the completion of the preceding thread in the chain, the number of activations in the window

is safely inflated by 1 to account for the data-driven activation delay (caused by the execution of the sender thread), if the WCRT of the sender thread  $\tau_f^s$  plus the communication delay  $\lambda_{(f,g)}^{x,y}$  is less than or equal to  $T_{(f,g)}^s = T_f^s$ . This is proven in Lemma 1.

**Lemma 1.** *Consider a sender thread  $\tau_i^s \in \mathcal{T}$  allocated on  $n_x \in \mathcal{N}$ , a consumer thread  $\tau_j^c \in \mathcal{T}$  on  $n_y \in \mathcal{N} \setminus n_x$ , and its copy-in thread  $\tau_{(i,j)}^s \in \mathcal{T}^{\text{cp}}$  (on  $n_y$ ). If  $R_i^s + \lambda_{(i,j)}^{x,y} \leq T_i^s$ , then the number of pending jobs of  $\tau_{(i,j)}^s$  in an arbitrary time window  $[t', t' + t)$  is bounded by  $(\lceil t/T_i^s \rceil + 1)$ .*

*Proof.* First note that if the number of pending jobs of  $\tau_{(i,j)}^s$  is bounded by  $\lceil (t + R_i^s + \lambda_{(i,j)}^{x,y})/T_{(i,j)}^s \rceil$ , then it is also bounded by  $(\lceil t/T_i^s \rceil + 1)$ , since  $R_i^s + \lambda_{(i,j)}^{x,y} \leq T_i^s$  (by assumption),  $T_{(i,j)}^s = T_{(i,j)}^s$ , and  $\lceil (t + T_i^s)/T_i^s \rceil \leq \lceil t/T_i^s \rceil + 1$  hold. The lemma follows by noting that there cannot be more than  $\lceil (t + R_i^s + \lambda_{(i,j)}^{x,y})/T_{(i,j)}^s \rceil$  pending jobs in  $[t', t' + t)$ , since the copy-in thread  $\tau_{(i,j)}^s$  behaves as a normal thread subject to a data-driven activation jitter [37], which is due to the response time of the sender thread  $R_i^s$  and the communication delay  $\lambda_{(i,j)}^{x,y}$ , yielding an overall jitter of  $R_i^s + \lambda_{(i,j)}^{x,y}$ .  $\square$

Consequently, the schedulability test in Eq. (1) becomes:

$$\exists t \in [0, D_i^d] \mid e_{i,k}^d + I_{i,k}^{\bar{m}}(t) \leq t, \quad (4)$$

with  $I_{i,k}^{\bar{m}}(t)$  as in Eq. (3).

Furthermore, the previous analysis assumes that each subchain composed of a sender thread and a copy-in thread has at most one pending instance at a time. Analogously to the classical case of periodic/sporadic threads with constrained deadlines [45], this can be achieved by guaranteeing that the whole subchain of events, composed of the producer, the network propagation, and the copy-in thread, completes within the producer thread's period  $T_i^d$ , i.e., ensuring the condition:

$$R_i^d + \lambda_{(i,j)}^{x,y} + R_{(i,j)}^d \leq T_i^d, \quad \forall (\tau_i, \tau_j) \in \mathcal{PC} \quad (5)$$

which, if satisfied, implies the condition required for Lemma 1.

Finally, following state-of-the-art results from Davare et al. [24] for communications occurring in shared memory, and from Henia et al. [37] for communications occurring in a data-driven manner, adding the communication delay  $\lambda_{(i,j)}^{x,y}$  to the latter, and considering the external event triggering the chain arriving synchronously with the release of the first thread of the chain  $\tau_f^d$ , the end-to-end latency of an arbitrary chain  $\gamma_x$  can then be easily bounded as follows:

$$L_{\gamma_x} = \sum_{\tau_i^d \in \gamma_x^{\text{exe}}} (T_i^d + R_i^d) + \sum_{\tau_{(i,j)}^d \in \gamma_x^{\text{cp}}} (R_{(i,j)}^d + \lambda_{(i,j)}^{x,y}) - T_f^d. \quad (6)$$

**Limiting the number of checkpoints.** Directly implementing the schedulability test of Eq. (4) in a linear programming

formulation is highly impractical, since it requires writing a separate inequality for each  $t$  in the interval.

In literature, several works attempted to ease this problem by reducing the number of points to test. Lehoczy et al. [45] showed that an exact schedulability test requires checking only the points corresponding to the activation instants of interfering threads plus the deadline of the thread under analysis. Other works [11, 68] further reduced such sets under different conditions. Anyway, the resulting number of constraints is still exponential with the thread set size in the worst case, and any further reduction requires relying on sufficient-only tests. Nonetheless, a sufficient-only test with high accuracy can provide significant benefits for the runtime of the optimization problem, especially if executed online, as proposed next in Section 7. Park and Park [54] showed that under implicit deadline with rate monotonic scheduling, restricting the checkpoints to the last activation of the interfering threads in  $[0, D_i]$ , plus the point  $D_i$ , provides a sufficient-only test with extremely high accuracy. Building upon this result, Pazzaglia et al. [55] extended this result to general thread set models with similar accuracy (drop in the order of 1%), and with a set of checkpoints polynomial in the number of threads.

For the case under analysis here, the application consists of a combination of purely periodic threads and threads activated with jitter by a periodic source (i.e., the copy-in threads). The formulation of Eq. (2) coincides with an initial (critical) release instant where all periodic threads are synchronously released at 0, while all interfering copy threads have the first activation released with maximum jitter (equal to their period) at 0, and all successive instances released periodically with no jitter [3]. Thus, the relative alignment of the activations resembles that of purely periodic threads with constrained deadlines, as the first two activations of each copy thread coincide. Building upon the corresponding case of periodic task set model with jitter in [55], the check in Eq. (4) can be restricted to  $\exists t \in \mathcal{V}_i^{\bar{m}}$ , with:

$$\mathcal{V}_i^{\bar{m}} := \bigcup_{\tau_h^q \in (\text{hp}_k^{\bar{m}}(\tau_i^d) \cup \text{hp}_k^{\bar{m},\text{cp}}(\tau_i^d))} \left\{ \left\lfloor \frac{D_i^d}{T_h^q} \right\rfloor \cdot T_h^q \right\} \cup \{D_i^d\}. \quad (7)$$

## 6. Design-Time Optimization

**MILP Inputs.** Next, we list the constants that are provided in input to the optimization problem.

- I1** For each pair of sender thread  $\tau_i$  and receiver thread  $\tau_j$ , i.e.,  $(\tau_i, \tau_j) \in \mathcal{PC}$ , a copy-in thread  $\tau_{(i,j)}$  is added in the set  $\mathcal{T}^{\text{cp}}$ . The optimizer only enables the thread if  $\tau_i$  and  $\tau_j$  are in different nodes.
- I2** The priority and criticality assignment of each thread are constant and user-defined.

Other placement constraints can be provided as input, such as groups of threads to be allocated in the same or

in different cores if required by the designer, which can be easily added to the optimization problem as additional constraints.

Before proceeding with the presentation of the optimization problem, we extend the set of points in Eq. (7) to account for all possible points for  $\tau_i$  in all combinations of modes. This extension is required because modes are decision variables in the MILP, but having a linear formulation requires checkpoints to be independent of the optimization variables.

The new set of checkpoints  $\mathcal{V}_i$  is hence defined as:

$$\mathcal{V}_i := \bigcup_{d \in \mathcal{M}_i} \bigcup_{\tau_h \in \mathcal{T}^*} \bigcup_{q \in \mathcal{M}_h} \left\{ \left[ \frac{D_i^d}{T_h^q} \right] \cdot T_h^q \right\} \cup \{D_i^d\}, \quad (8)$$

with  $\mathcal{T}^* = \text{hp}(\tau_i) \cup \text{hp}^{\text{cp}}(\tau_i)$ .

**Main MILP Variables.** The optimization problem is characterized by the following sets of variables:

- Thread assignment to Core (TC): For each thread  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , for each core  $p_k \in \mathcal{P}$ ,  $\text{TC}_{i,k} \in \{0, 1\}$ , is a binary variable set to 1 if  $\tau_i$  is assigned to  $p_k$ ; 0 otherwise.
- Thread assignment to Node (TN): For each thread  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , and for each node  $n_x \in \mathcal{N}$ ,  $\text{TN}_{i,x} \in \{0, 1\}$  is a binary variable set to 1 if  $\tau_i$  is assigned to  $n_x$ ; 0 otherwise.
- Threads in Same Core (SC): For each thread pair  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ ,  $\tau_j \in \mathcal{T} \cup \mathcal{T}^{\text{cp}} \setminus \tau_i$ ,  $\text{SC}_{i,j} \in \{0, 1\}$  is set to 1 if  $\tau_i$  is assigned on the same core as  $\tau_j$ ; 0 otherwise.
- Threads in Same Node (SN): For each pair of communicating threads  $(\tau_i, \tau_j) \in \mathcal{PC}$ ,  $\text{SN}_{i,j} \in \{0, 1\}$  is set to 1 if  $\tau_i$  is assigned to the same node as  $\tau_j$ ; 0 otherwise.
- Thread Mode (TM): For each thread  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , for each mode  $d \in \mathcal{M}_i$ ,  $\text{TM}_{i,d} \in \{0, 1\}$  is equal to 1 if thread  $\tau_i$  is set to mode  $d$ ; 0 otherwise.
- Mapping of Communicating threads (MC): For each pair of communicating threads  $(\tau_i, \tau_j) \in \mathcal{PC}$ , for each node pair  $n_x \in \mathcal{N}$ ,  $n_y \in \mathcal{N} \setminus n_x$ ,  $\text{MC}_{x,y}^{i,j} \in \{0, 1\}$  is equal to 1 if  $\tau_i$  is in node  $n_x$  and  $\tau_j$  is in node  $n_y$ ; 0 otherwise.
- WOET of Thread (ET): For each thread  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , for each mode  $d \in \mathcal{M}_i$ ,  $\text{ET}_i^d \in \mathbb{R}^{\geq 0}$  is the WOET of  $\tau_i$ , which depends on the core in which it is allocated to and the mode of the thread.
- WOET of an Interfering thread (EI): For each thread pair  $\tau_j \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ ,  $\tau_i \in \text{hp}(\tau_j) \cup \text{hp}^{\text{cp}}(\tau_j)$  for each mode  $d \in \mathcal{M}_i$ ,  $\text{EI}_{i,j}^d \in \mathbb{R}^{\geq 0}$  is equal to the WOET of  $\tau_i$  (on the core in which it is allocated) if it can interfere with  $\tau_j$  and  $\tau_i$  is in mode  $d$ ; 0 otherwise.
- Response time Candidate of a thread (RC): For each  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , for each  $v_{i,g} \in \mathcal{V}_i$ ,  $\text{RC}_{i,g} \in \mathbb{R}^{\geq 0}$  is a candidate WCRT bound for  $\tau_i$ .

Var.	Description	Type
$\text{TC}_{i,k}$	Set if thread $\tau_i$ is assigned to $p_k$	Binary
$\text{TN}_{i,x}$	Set if thread $\tau_i$ is assigned to $n_x$	Binary
$\text{SC}_{i,j}$	Set if $\tau_i$ and $\tau_j$ share the same core	Binary
$\text{SN}_{i,j}$	Set if $\tau_i$ and $\tau_j$ share the same node	Binary
$\text{MC}_{x,y}^{i,j}$	Set if $\tau_i$ is in node $n_x$ and $\tau_j$ in $n_y$	Binary
$\text{TM}_{i,d}$	Set if thread $\tau_i$ is set to mode $d$	Binary
$\text{ET}_i^d$	WOET of thread $\tau_i$ in mode $d$	Real
$\text{EI}_{i,j}^d$	WOET of interf. $\tau_i$ in mode $d$	Real
$\text{RC}_{i,g}$	Candidate WCRT for thread $\tau_i$	Real
$\text{SV}_{i,g}$	Selector variable for WCRT at $v_{i,g}$	Binary
$\text{RT}_i$	WCRT bound of thread $\tau_i$	Real
$\text{SCC}_{i,j,k}$	Set if $\tau_i$ and $\tau_j$ are both on $p_k$	Binary
$\text{SNN}_{i,j,x}$	Set if $\tau_i$ and $\tau_j$ are both on $n_x$	Binary
Sym.	Description	
$\text{TV}_i$	Period of $\tau_i$ in the selected mode	-
$\text{DV}_i$	Deadline of $\tau_i$ in the selected mode	-
$\text{CD}_{i,j}$	Comm. delay between $\tau_i$ and $\tau_j$	-

**Table 3**

Variables and Symbols used in Section 6.

- Selector Variable for response time candidate of a thread (SV): For each  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , for each  $v_{i,g} \in \mathcal{V}_i$ ,  $\text{SV}_{i,g} \in \{0, 1\}$  is set to 1 if the WCRT is selected for point  $v_{i,g}$ .
- Response Time of a thread (RT): For each  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ ,  $\text{RT}_i \in \mathbb{R}^{\geq 0}$  is the WCRT bound of  $\tau_i$ .

The variables and symbols used in this section are also summarized in Table 3.

**Constraints Overview.** We now present the constraints of our optimization problem. For some constraints, we use the so-called *big-M* formulation [32], where  $M$  is a large constant, representing infinity. In the following, all constraints that are not straightforward to derive are accompanied by a corresponding proof.

Constraints are organized as follows. Constraints 1-2 enforce the properties related to the thread-to-node, thread-to-core and thread-to-mode assignment. Constraints 3-4 enforce the formulations related to execution times. Constraint 3 ensures the execution time accounts for the selected thread-to-core and thread-to-core assignment. Constraint 4 focuses instead on the execution time of a thread if it interferes with another thread. Constraints 5-8 compute the response time bounds and ensure schedulability of both threads and chains.

**MILP Constraints.** First, we enforce a group of basic constraints on thread assignments to cores and to nodes, as well as thread modes. Constraint 1 (a) and (b) enforce that each thread is assigned to only one core and mode, respectively; Constraint 1(c) enforces that a copy-in thread is mapped to the same node as the receiving thread; Constraint 1(d) enforces that a copy-in thread has the same mode of the sender thread.

**Constraint 1** (Basic Constraints).

$$\text{For each } \tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}} : \quad \sum_{p_k \in \mathcal{P}} TC_{i,k} = 1 \quad (9a)$$

$$\text{For each } \tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}} : \quad \sum_{d \in \mathcal{M}_i} TM_{i,d} = 1 \quad (9b)$$

$$\text{For each } \tau_{(i,j)} \in \mathcal{T}^{\text{cp}}, n_x \in \mathcal{N} : \quad TN_{(i,j),x} = TN_{j,x} \quad (9c)$$

$$\text{For each } \tau_{(i,j)} \in \mathcal{T}^{\text{cp}}, d \in \mathcal{M}_i : \quad TM_{i,d} = TM_{(i,j),d} \quad (9d)$$

Constraint 2 defines the thread-to-node mapping starting from the thread-to-core mapping.

**Constraint 2** (Threads to nodes mapping). *For each thread*  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , *and for each node*  $n_x \in \mathcal{N}$ ,

$$TN_{i,x} = \sum_{p_k \in \mathcal{P}_x} TC_{i,k} \quad (10)$$

*Proof.* The constraint follows by noting that, by definition of  $TN_{i,x}$ ,  $TN_{i,x}$  is set to 1 if thread  $\tau_i$  is assigned to one of its cores  $p_k \in \mathcal{P}_x$ , and is 0 otherwise.  $\square$

Next, we enforce the definition of variables  $SC_{i,j}$  (threads  $\tau_i$  and  $\tau_j$  in same core) and  $SN_{i,j}$  (threads  $\tau_i$  and  $\tau_j$  in same node). This can be achieved by standard techniques to implement linearly AND ( $\wedge$ ) and OR ( $\vee$ ) constraints in optimization problems [32] by defining a set of auxiliary boolean variables  $SCC_{i,j,k}$  and  $SNN_{i,j,x}$  to denote whether threads  $\tau_i$  and  $\tau_j$  are allocated on the same specific core  $p_k$  or specific node  $n_x$ , respectively. This is performed by enforcing the constraints

$$SCC_{i,j,k} = TC_{i,k} \wedge TC_{j,k} \quad \text{and} \quad SC_{i,j} = \bigvee_{p_k \in \mathcal{P}} SCC_{i,j,k},$$

and analogously for nodes

$$SNN_{i,j,x} = TN_{i,x} \wedge TN_{j,x} \quad \text{and} \quad SN_{i,j} = \bigvee_{n_x \in \mathcal{N}} SNN_{i,j,x}.$$

The definition of variables  $MC_{x,y}^{i,j}$  (mapping of communicating threads  $(\tau_i, \tau_j)$ , in nodes  $n_x, n_y$ ) relies on the same techniques, but just by enforcing

$$MC_{x,y}^{i,j} = TN_{i,x} \wedge TN_{j,y}.$$

Next, we enforce the definition of  $ET_i^d$ , that is, the WOET of a thread  $\tau_i^d$ . For computational efficiency, Constraints 3-5 define only lower bounds of the variables: the solver will select the smaller values that satisfy the real-time constraints (defined later in Constraints 6-8).

**Constraint 3** (WOET of a thread). *For each thread*  $\tau_i \in \mathcal{T}$ , *for each mode*  $d \in \mathcal{M}_i$ ,

$$ET_i^d \geq \sum_{p_k \in \mathcal{P}} e_{i,k}^d \cdot TC_{i,k}. \quad (11a)$$

*For each thread*  $\tau_{(i,j)} \in \mathcal{T}^{\text{cp}}$ , *for each mode*  $d \in \mathcal{M}_i$ ,

$$ET_{(i,j)}^d \geq \sum_{p_k \in \mathcal{P}} e_{(i,j),k}^d \cdot TC_{(i,j),k} - SN_{i,j} \cdot M. \quad (11b)$$

*Proof.* The first inequality follows by noting that the only term greater than zero is associated with the core in which  $TC_{i,k} = 1$  (for Constraint 1(a) only one such a core exists). The second inequality for copy-in threads is similar, but it also disables the constraint ( $ET_{(i,j)}^d \geq -\infty$ ) if  $SN_{i,j} = 1$ , i.e., if the producer and consumer are in the same node (thus, no copy-in thread is needed).  $\square$

Constraint 4 enforces the definition of variable  $EI_{i,j}^d$ , i.e., the WOET of  $\tau_i^d$  if it can interfere with  $\tau_j$ .

**Constraint 4** (Interfering WOET). *For each pair of threads*  $\tau_j \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ ,  $\tau_i \in hp(\tau_j) \cup hp^{\text{cp}}(\tau_j)$ , *for each*  $d \in \mathcal{M}_i$ ,

$$EI_{i,j}^d \geq ET_i^d - M \cdot (2 - SC_{i,j} - TM_{i,d}). \quad (12)$$

*Proof.* The constraint enforces the definition of  $EI_{i,j}^d$ , being  $EI_{i,j}^d \geq ET_i^d$  only if  $\tau_i$  and  $\tau_j$  are in the same core ( $SC_{i,j} = 1$ ) and only if  $\tau_i$  is in mode  $d$  ( $TM_{i,d} = 1$ ); otherwise the constraint is disabled:  $EI_{i,j}^d \geq -\infty$ .  $\square$

Next, Constraint 5 enforces the conditions required for  $RT_i$  to correctly encode a response time bound for  $\tau_i$ . Before proceeding, we define the auxiliary variable  $ET_i$  to represent the WOET of the thread  $\tau_i$  under analysis (independent of the mode), enforcing the constraint

$$ET_i \geq \sum_{d \in \mathcal{M}_i} ET_i^d - (1 - TM_{i,d}) \cdot M.$$

For conciseness in the presentation of the next constraint, we also introduce the sets  $hp_{\text{am}}(\tau_i)$ ,  $hp_{\text{am}}^{\text{cp}}(\tau_i)$ , which are analogous to  $hp(\tau_i)$  and  $hp^{\text{cp}}(\tau_i)$  but they contain multiple occurrences of all threads, one for each mode, in such a way that  $\sum_{\tau_h^q \in hp_{\text{am}}(\tau_i)} (\cdot)$  is equivalent to  $\sum_{\tau_h \in hp(\tau_i)} \sum_{q \in \mathcal{M}_h} (\cdot)$ .

**Constraint 5** (WCRT bound candidate). *For each thread*  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ , *and for each*  $v_{i,g} \in \mathcal{V}_i$  *obtained with Equation (8),*

$$RC_{i,g} \geq ET_i + \sum_{\tau_h^q \in hp_{\text{am}}(\tau_i)} \left[ \frac{v_{i,g}}{T_h^q} \right] EI_{h,i}^q + \dots \\ + \sum_{\tau_h^q \in hp_{\text{am}}^{\text{cp}}(\tau_i)} \left( \left[ \frac{v_{i,g}}{T_h^q} \right] + 1 \right) EI_{h,i}^q \quad (13a)$$

$$RC_{i,g} \leq v_{i,g} + (1 - SV_{i,g}) \cdot M, \quad (13b)$$

$$RT_i \geq RC_{i,g} - (1 - SV_{i,g}) \cdot M. \quad (13c)$$

*Additionally, for each*  $\tau_i \in \mathcal{T} \cup \mathcal{T}^{\text{cp}}$ ,

$$\sum_{v_{i,g} \in \mathcal{V}_i} SV_{i,g} = 1. \quad (13d)$$

*Proof.* Given an arbitrary checkpoint  $v_{i,g} \in \mathcal{V}_i$ , Eq. (13a) enforces its corresponding response time candidate  $RC_{i,g}$  to be greater than or equal to its computational requirement  $ET_i$  plus the interference in the core to which it is allocated,

computed as in Eq. (3) for  $t = v_{i,g}$ . Then, Eq. (13b) enforces  $RC_{i,g}$  to be smaller than the time that is provided by the core up to  $t = v_{i,g}$  if  $SV_{i,g} = 1$ , otherwise it has no effect ( $RC_{i,g} \leq \infty$ ). Using Eq. (13d) and the set of boolean variables  $SV_{i,g}$ , the solver enforces the second inequality (13b) only for one of the points in set  $\mathcal{V}_i$ : in other words, it is sufficient it exists one checkpoint, namely  $v_{i,g^*} \in \mathcal{V}_i$ , in which the condition (13b) is true, thus implementing the schedulability test in Eq. (4). For that checkpoint  $v_{i,g^*}$  it follows that  $SV_{i,g^*} = 1$ . Finally, Eq. (13c) selects such a checkpoint  $v_{i,g^*}$  as the response-time bound for  $\tau_i$ .  $\square$

For the next constraints, we first define the auxiliary symbols  $TV_i = \sum_{d \in \mathcal{M}_i} T_i^d \cdot TM_{i,d}$  and  $DV_i = \sum_{d \in \mathcal{M}_i} D_i^d \cdot TM_{i,d}$  that denote the period and deadline of an arbitrary thread  $\tau_i$  in the mode that the solver selected, respectively (which easily holds by recalling Eq. (9b), i.e., only one value of  $TM_{i,d}$  is equal to 1). We also define the symbol  $CD_{i,j} = \sum_{n_x \in \mathcal{N}} \sum_{n_y \in \mathcal{N} \setminus n_x} \lambda_{(x,y)}^{i,j} \cdot MC_{x,y}^{i,j}$  to denote the communication delay experienced in the communication of threads  $\tau_i$  and  $\tau_j$ .

Leveraging the auxiliary variables introduced above, the last constraints of the formulation can be compactly defined. Constraint 6 enforces the condition in Eq. (5), while Constraints 7 and 8 enforce the timing constraints.

**Constraint 6** (Analysis Condition). *For each thread pair  $(\tau_i, \tau_j) \in \mathcal{PC}$ ,*

$$RT_i + RT_{(i,j)} + CD_{i,j} \leq TV_i. \quad (14)$$

**Constraint 7** (Schedulability). *For each  $\tau_i \in \mathcal{T}$ ,*

$$RT_i \leq DV_i. \quad (15)$$

**Constraint 8** (Chain Deadline). *For each  $\gamma_x = (\tau_f, \dots, \tau_l) \in \Gamma$*

$$\sum_{\tau_i \in \gamma_x} (RT_i + TV_i) + \sum_{\tau_i \in \gamma_x \setminus \tau_l} (RT_{(i,j)} + CD_{i,j}) - TV_f \leq D_{\gamma_x}, \quad (16)$$

where, for each  $\tau_i \in \gamma_x \setminus \tau_l$ ,  $\tau_j$  is its successor in the  $\gamma_x$ .

*Proof.* The constraint implements the latency bound of Eq. (6) and enforces it to be smaller than or equal to the deadline.  $\square$

**Objective function(s).** We consider *three* different (alternative) objective functions.

1. **MODE-WEIGHTED** minimizes the weighted ratio of the selected thread mode multiplied by a function of the criticality of the thread to maximize the accuracy of the functional results by privileging thread with higher criticalities,

$$\min \sum_{\tau_i \in \mathcal{T}} \sum_{d \in \mathcal{M}_i} (d \cdot TM_{i,d} \cdot w(c_i)).$$

The objective function is explained as follows. Due to Eq. (9b) of Constraint 1, the solver must set the values

of variables  $TM_{i,d}$  such that only one term of the summation is different from zero for each thread  $\tau_i \in \mathcal{T}$ . Each value in the summation is weighted by  $w(c_i)$ , which is a function that maps each criticality index to a corresponding weight to be used in the objective function. As discussed in Section 3, a lower mode index identifies an operational mode that provides a more accurate result for some metrics of interest. Thus, for the same value of  $w(c_i)$ , the optimizer will try to select the best accurate execution modes whenever possible, since  $TM_{i,d}$  is multiplied by the corresponding index  $d$ . In this way, it is possible to tune the mode/criticality trade-off in the objective function.

For example, consider a system with three threads  $\tau_1, \tau_2$ , and  $\tau_3$ , with  $c_i = i$  (for simplicity), and all being characterized by two modes 1 and 2, which are the values of  $d$  in the objective function. By setting  $w(c_1) = 1000$ ,  $w(c_2) = 400$ , and  $w(c_3) = 1$ , for  $d = 1$  the product  $d \cdot w(c_i)$  results to 1000, 400, and 1, respectively; for  $d = 2$ ,  $d \cdot w(c_i)$  results to 2000, 800; and 2. Therefore, the solver will try to privilege higher criticality threads to be assigned to more accurate modes, followed by medium and low criticality threads, because this will also minimize the ratio  $d \cdot w(c_i)$ : assigning the highest criticality thread  $\tau_1$  to mode 1 will be considered more important because it would just add 1000 to the objective function; on the contrary, if  $\tau_1$  is set to mode 2 the payback would be 2000 (the difference between the two is 1000). Conversely, assigning the medium criticality thread  $\tau_2$  to mode 1 instead of mode 2 will provide a difference in the objective function equal to 400 because the product  $d \cdot w(c_2)$  evaluates to 400 and 800, respectively.

2. **RD-MAX** aims to provide a robust assignment that preserves the schedulability of threads through the changes in parameters, by minimizing the maximum ratio between the response time and the deadline. It requires defining a single additional variable  $R_{\text{MAX}}$ , subject to the following constraint: for each  $\tau_i \in \mathcal{T}$ , for each  $d \in \mathcal{M}_i$ ,

$$R_{\text{MAX}} \geq (RT_i / D_i^d) - (1 - TM_{i,d}) \cdot M,$$

and thus the objective function being **min**  $R_{\text{MAX}}$ .

3. **LT-MAX** minimizes the maximum latency of the chains. It requires defining one variable for each chain to store its latency, computed as the left-hand expression of Eq. (16), and another one  $L_{\text{MAX}}$  for the maximum among them, then solving **min**  $L_{\text{MAX}}$ .

**Complexity and Scalability.** Table 4 lists the number of constraints and number of variables per constraint of the MILP formulation proposed above. In the table, sets with index “\*” are a short notation for the sets with maximum size among their peers, e.g.,  $|\mathcal{M}_*| = \max_{\tau_i \in \mathcal{T}} |\mathcal{M}_i|$ . Recalling Equation (8), we can also find the upper-bound for the set of

Cnstr.	Eq.	# of constraints	# of variables per constraint
Cnstr. 1	(9a)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}}) )$	$\mathcal{O}( \mathcal{P} )$
Cnstr. 1	(9b)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}}) )$	$\mathcal{O}( \mathcal{M}_* )$
Cnstr. 1	(9c)	$\mathcal{O}( \mathcal{T}^{\text{cp}}  \times  \mathcal{N}' )$	$\mathcal{O}(1)$
Cnstr. 1	(9d)	$\mathcal{O}( \mathcal{T}^{\text{cp}}  \times  \mathcal{M}_* )$	$\mathcal{O}(1)$
Cnstr. 2	(10)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}})  \times  \mathcal{N}' )$	$\mathcal{O}( \mathcal{P}_* )$
Cnstr. 3	(11a)	$\mathcal{O}( \mathcal{T}  \times  \mathcal{M}_* )$	$\mathcal{O}( \mathcal{P} )$
Cnstr. 3	(11b)	$\mathcal{O}( \mathcal{T}^{\text{cp}}  \times  \mathcal{M}_* )$	$\mathcal{O}( \mathcal{P} )$
Cnstr. 4	(12)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}}) ^2 \times  \mathcal{M}_* )$	$\mathcal{O}(1)$
Cnstr. 5	(13a)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}})  \times  \mathcal{V}_* )$	$\mathcal{O}( \mathcal{T} \cup \mathcal{T}^{\text{cp}} )$
Cnstr. 5	(13b)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}})  \times  \mathcal{V}_* )$	$\mathcal{O}(1)$
Cnstr. 5	(13c)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}})  \times  \mathcal{V}_* )$	$\mathcal{O}(1)$
Cnstr. 5	(13d)	$\mathcal{O}( (\mathcal{T} \cup \mathcal{T}^{\text{cp}}) )$	$\mathcal{O}( \mathcal{V}_* )$
Cnstr. 6	(14)	$\mathcal{O}( \mathcal{T} ^2)$	$\mathcal{O}(1)$
Cnstr. 7	(15)	$\mathcal{O}( \mathcal{T} )$	$\mathcal{O}(1)$
Cnstr. 8	(16)	$\mathcal{O}( \Gamma )$	$\mathcal{O}( \mathcal{T} )$
Cnstr. 9	(17a)	$\mathcal{O}( \mathcal{T} )$	$\mathcal{O}( \mathcal{M}_* )$
Cnstr. 9	(17b)	$\mathcal{O}( \mathcal{T} )$	$\mathcal{O}( \mathcal{M}_* )$
Cnstr. 10	(18a)	$\mathcal{O}( \mathcal{T} )$	$\mathcal{O}(1)$
Cnstr. 10	(18b)	$\mathcal{O}( \mathcal{T} )$	$\mathcal{O}(1)$

**Table 4**

Complexity of the MILP formulation in terms of the number of constraints and number of variables per constraint.

schedulability points being  $|\mathcal{V}_*| \leq |\mathcal{M}_*|^2 \times |\mathcal{T}|$ . The total number of variables for the proposed MILP formulation is then  $\mathcal{O}(|\mathcal{P}_*| \times |\mathcal{M}_*|^2 \times |(\mathcal{T} \cup \mathcal{T}^{\text{cp}})|)$ , while the total number of constraints is  $\mathcal{O}(|(\mathcal{T} \cup \mathcal{T}^{\text{cp}})|^2 \times |\mathcal{N}'| \times |\mathcal{M}_*|^2 \times |\Gamma|)$ . In other words, the MILP formulation complexity is affected in particular (quadratically) by the total number of threads (including the copy-in threads), and also by the number of modes per thread. While being potentially computationally intensive for realistically sized systems, we remark that such full-system optimization is required to be performed offline, only once, at design stage. In the next section we cover the aspects of the additional formulation for runtime re-optimization, which often involves only a subset of the variables and constraints introduced here.

## 7. Orchestration and Runtime Decision-Making

As described in Section 4, reaction policies for orchestration and runtime decision-making are triggered by the arrival of a new trigger event (with a focus on WOET updates) to the orchestrator. When this happens, the orchestrator runs a modified version of the optimization problem presented in Section 6, assigning fixed values to the variables of threads that will remain unaffected by the event, thus re-optimizing only a subset. This approach has multiple advantages: it avoids changing the current deployment too much, avoids affecting components that are not influenced by a particular trigger event, and speeds up the optimization procedure, which is essential for runtime reconfiguration and to enhance

Var.	Description	Type
$\text{bM}_i$	Set if the mode of $\tau_i$ can be changed	Binary
$\text{bD}_i$	Set if the deadline of $\tau_i$ is fully inflated	Binary
$D\varepsilon_i$	Deadline inflation coefficient for thread $\tau_i$	Real

**Table 5**

Additional variables introduced in Section 7.

scalability. In the following, we also present the computational complexity of the approach under the different reaction policies, by discussing the corresponding simplified optimization problem.

In the following, we refer to a generic thread set  $\mathcal{T}_{\text{rc}}$  that includes only the threads affected by the reconfiguration according to a particular instantiation of the runtime decision-making policy. The threads to be considered in set  $\mathcal{T}_{\text{rc}}$  depend on both the enabled reaction policies and the specific use case: its definition for a specific application is thus left to the designer. We show some examples in the evaluation (Section 9).

The additional variables required by the methods of this section are summarized in Table 5 and also presented in detail in the following.

**Mode relaxation.** With mode relaxation, the orchestrator tries to switch one or more threads to a degraded execution mode, characterized by a smaller execution time requirement and/or a larger period and deadline, to meet timing requirements at the expense of reduced functional performance. Again, it is fundamental for the designer to have control over the guaranteed degradation order since the impact of a mode degradation can be different depending on the specific thread. To this end, we rely on an optimization problem based on a subset of constraints from the formulation of Sec. 6. For the mode relaxation, we are interested in re-using only those constraints that are influenced by the variable  $\text{TM}_{i,d}$ . Moreover, all constraints related to threads that are not in  $\mathcal{T}_{\text{rc}}$  can be effectively removed by fixing the corresponding variables to the values of the current configuration. For example, the variables in (9a) and (9c) are substituted by the values found in the offline optimization of the setup. On the other hand, we require an additional set of constraints to guide the solver towards reconfiguring the mode degradation. To this end, we first introduce the following variable:

- *Mode degradation order enforcer:* For each thread  $\tau_i \in \mathcal{T}_{\text{rc}}$ ,  $\text{bM}_i \in \{0, 1\}$  is equal to 1 if and only if the solver is allowed to change the mode of  $\tau_i$ .

The following constraints enforce the relationship between  $\text{bM}_i$  and the order dictated by the thread's criticality  $c_i$ .

**Constraint 9.** For each  $\tau_i \in \mathcal{T}_{\text{rc}}$  with current mode  $tm_i$ ,

$$\sum_{d \in \mathcal{M}_i} (d \cdot \text{TM}_{i,d}) \leq tm_i + \text{bM}_i \cdot M, \quad (17a)$$

$$\sum_{d \in \mathcal{M}_i} (d \cdot \text{TM}_{i,d}) \geq tm_i - \text{bM}_i \cdot M. \quad (17b)$$

For each thread  $\tau_i \in \mathcal{T}_{rc}$ , given the thread  $\tau_j \in \mathcal{T}_{rc}$  with immediately lower criticality,

$$bM_i \leq bM_j \cdot M. \quad (17c)$$

*Proof.* When the thread with immediately lower criticality  $\tau_j$  has  $bM_j = 0$ , then Eq. (17c) enforces that also  $bM_i = 0$ . Thus, from Eq. (17a) and Eq. (17b) the mode of  $\tau_i$  in the optimization problem is fixed to the value  $m_i$  of the current solution. Otherwise if  $bM_j = 1$  no constraint is enforced (Eq. (17a) and Eq. (17b) become  $-\infty \leq MM_i \leq \infty$ ), allowing the solver to change the mode freely.  $\square$

The objective function is set to minimize the sum of the boolean variables  $bM_i$  so that only the minimum number of threads (in order of criticality) are affected by the mode relaxation, i.e.,  $\min \sum_{\tau_i \in \mathcal{T}_{rc}} bM_i$ . Due to effectively fixing all variables related to cores and nodes assignment, and considering a restricted subset of threads  $\mathcal{T}_{rc}$  affected by the reconfiguration, the final complexity of such reaction policy is  $\mathcal{O}(|\mathcal{M}_*|^2 \times |\mathcal{T}_{rc}|)$ , while the total number of constraints is  $\mathcal{O}(|\mathcal{T}_{rc}|^2 \times |\mathcal{M}_*|^2 \times |\Gamma|)$ .

**Deadline inflation.** The deadline inflation strategy uses an *inflation coefficient*, defined for each overrunning thread  $\tau_i$ , to decide the amount of inflation of the deadline. The coefficients are introduced as variables of the MILP formulation as follows:

- *Deadline inflation:* For each thread  $\tau_i \in \mathcal{T}_{rc}$ ,  $D\epsilon_i \in \mathbb{R}^{\geq 0}$ , is the amount of which the deadline of the thread is inflated.

These variables need to be inserted in a new MILP formulation, so that the solver can optimize their values.

As for the mode relaxation, all the variables that are not involved in the deadline inflation and the corresponding schedulability test need to be assigned to constant values given by the current solution deployed on the distributed system. Then, Constraint 7 is substituted by the following formulation:

**Constraint 10** (Schedulability under deadline inflation). For each  $\tau_i \in \mathcal{T}_{rc}$ ,

$$RT_i \leq DV_i + D\epsilon_i, \quad (18a)$$

$$DV_i + D\epsilon_i \leq TV_i. \quad (18b)$$

Since our analysis relies on the assumption of constrained deadlines  $D_i^d \leq T_i^d$ , the corresponding constraint  $DV_i + D\epsilon_i \leq TV_i$  of Eq. (18b) is added, and the set of points in Eq. (8) needs to be updated to add point  $T_i^d$ , which is the new maximum deadline allowed to the thread. In this reaction policy, all assignments of cores, nodes and modes can be kept fixed. The constraints related to the response time analysis are nonetheless to be re-checked, since the deadlines can be modified. The final complexity of such policy is  $\mathcal{O}(|\mathcal{T}_{rc}|)$ , while the total number of constraints is  $\mathcal{O}(|\mathcal{T}_{rc}|^2 \times |\Gamma|)$

Designers could configure the deadline inflation to work together with mode relaxation, thus requiring the definition

of an ordering across the two degradation policies. Indeed, the orchestrator could react to an unschedulability condition for a thread  $\tau_i$ , detected by running the response-time analysis of Section 5, by either inflating the deadline of  $\tau_i$ , by changing its mode, or that of other threads that can interfere with  $\tau_i$ . This can be done by adding another MILP variable to account for the order of degradation, as follows:

- *Deadline degradation order:* For each thread  $\tau_i \in \mathcal{T}_{rc}$ ,  $bD_i \in \{0, 1\}$  is equal to 1 if and only if the solver is allowed to inflate the deadline of thread  $\tau_i$ .

The variable can be used to add a set of constraints for each thread  $\tau_i \in \mathcal{T}$ , and given the thread  $\tau_j \in \mathcal{T}$  with immediately lower degradation order (e.g., with immediately lower criticality), expressed as either  $D\epsilon_i \leq bD_j \cdot M$  or  $D\epsilon_i \leq bM_j \cdot M$  depending on whether the deadline inflation of  $\tau_i$  is preceded by either a mode relaxation or deadline inflation for  $\tau_j$  in the degradation order. Then, the objective function is also changed to minimize the sum of variables  $bM_i$  and  $bD_i$ .

Thread degradation achieved with mode relaxation and deadline inflation does not create a danger to the timing constraints of a communication chain. Indeed, both options rely on solving a simplified version of the optimization problem for initial allocation with all the parameters (thread-to-core allocations, thread modes, etc.) that are not affected by the degradation policy remaining constants, as explained above. However, Constraint 8 is still present (with possibly some of the elements in the summations being constants instead of variables, if not involved in the degradation strategy configuration) and it ensures that the end-to-end deadline is respected. Therefore, the MILP solver needs to find a degraded solution that still meets the chain deadline.

**Orchestration.** Another possible degree of freedom to restore schedulability is orchestration, i.e., moving threads from one core to another or even between different nodes. This can be done by using the optimization problem of Sec. 6, enforcing it to change the assignment of only a subset of threads or cores. Possible orchestration policies are: *local*, which reallocates a thread only within the same node; *global*, which reallocates a thread in any node of the edge system; *clustered*, which reallocated a thread only in a cluster of cores selected by the designer (e.g., only some cores with lowest overall load). The MILP formulation can be flexibly adapted to many different application scenarios and be configured by the application designer according to the specific needs. The resulting complexity of this policy has a similar expression to the one of the full offline MILP, substituting the thread set with  $\mathcal{T}_{rc}$  as well as considering the correspondingly affected nodes and cores. Similarly to mode relaxation and deadline inflation, orchestration does not harm the end-to-end deadline constraints of chains since Constraint 8 is still active.

**Thread discarding.** If schedulability cannot be recovered with the previous techniques, discarding some threads to leave room for the others can be a powerful ultimate solution.

Nevertheless, it is likely to be a symptom of a poorly designed system; hence, discarding threads should be avoided since it might impact the functional behavior of the system. Furthermore, it makes the optimization much more complex for the general case; however, some simpler but common cases can still be managed. For example, a possible way of integrating thread discarding in the optimization problem is to provide, for some low-criticality threads that are not involved in communication chains, an additional mode in which the execution time is equal to 0. More difficult is instead discarding threads that are part of a chain since the absence of a thread in the chain clearly invalidates the functional behavior of the chain itself due to input-output relations between threads.

A possible solution to use discarding for threads involved in chains in which no thread receives/sends data from/to more than one producer/consumer is to individuate low-criticality chains, which can be, for example, related to a non-safety related system service (e.g., infotainment) and remove the functionality as a whole. Again, the approach of modeling the discarding as an additional mode with zero execution time can be used, e.g., for the source node of the chain: then, additional constraints need to be added to specify that, for each chain, if the source node is discarded (and hence set to the zero execution time functional mode), then also the other threads in the chain need to be discarded. This results in just adding an “*if-then-else*” (which can easily be implemented in linear form with the techniques already discussed in the paper) constraint to state that if the source thread is set to the “*discard*” mode, then also all the other threads in the chain should be in the same mode.

General chains with threads receiving or sending data from/to more than one producer/consumer may consider such threads as “*forking/joining junctions*”, which give rise to multiple chains in which the threads preceding the junction contribute to multiple chains. Hence, it can be decided to consider discardable only threads following/preceding one branch after the junction, with threads contributing to only one low-criticality chain.

Another solution that does not involve modifying the optimization problem is to run multiple instances of the optimization problem by providing different thread sets in input by iteratively removing one or more threads that have been marked as “*rejectable*” by the designer: these threads need to be carefully selected to preserve the consistency of event chains. Again, this can be done by assigning criticality values to threads, following a degradation order decided by the designer. For example, the corresponding algorithm can include solving the optimization problem again after first removing all the low-criticality threads not involved in chains and seeking a solution. If no solution is found, the linear sequence of threads in low-criticality chains (not involved in forks/join) can be removed, too. While this option is worth mentioning as an ultimate solution to recover from an unexpected timing misbehavior, we do not consider it in detail since its usage should be kept at a minimum, and

if triggered, the designer should consider revising the system design.

**Restoring the performance.** The previous techniques allow for restoring schedulability at the expense of performance degradation. Then, if possible, the system should be scaled horizontally (i.e., adding more computing platforms), enabling the restoration of the non-degraded mode. Low-criticality soft real-time systems can also consider some observed overload conditions as temporary and restore the original configuration if no further violations are detected within a timeout limit, or considering the WOET in a moving time window. Nevertheless, how often a WOET overshoot may occur is hard to predict, making it hard to set a safe timeout limit or length of such time window. Hence, considering the new WOET estimate as a consequence of an overrun as the new permanent WOET estimate is often a safer solution. Furthermore, in case WOET overruns are just transients, e.g., when a memory-intensive thread suffers an overrun due to excessive memory interference coming from another memory-intensive thread allocated to another core of the same node, *orchestration* offers a valuable solution to overcome this issue by reallocating the interfered or interfering thread to another node to avoid mutual interference between the two. This also offers a more resilient solution in case the transient exceedance occurs again. A detailed study of these techniques is left as future work.

## 8. Prototype Implementation

This section describes a prototype implementation of the runtime environment for our proposed approach, targeting low-end embedded systems and the QNX RTOS. An overview of the implemented system is shown in Fig. 2.

QNX is a POSIX-based RTOS and ISO26262 certified to the highest assurance level (ASIL-D) [22]. Because of this, QNX is the preferred base operating system for high-performance platforms of many car manufacturers. While our implementation focuses on QNX, where possible, the standard POSIX API is used to allow cross-OS compatibility. The proposed implementation is based on the QNX 7.1 Software Development Platform (SDP) and considers multiple compute nodes that are interconnected by an Ethernet network.

**Code Generation.** The initial allocation is provided by the optimizer in the form of an Amalthea APP4MC model [39]. Amalthea is an open-source data model that allows the modeling of non-functional aspects of complex hardware and software systems. An automated process is implemented to generate the code for each QNX node in such a way as to fully abstract away the execution and communication details based on copy-in threads presented in Section 3. The automatic code generation of the functional behavior is out of the scope of this paper. Still, state-of-the-art techniques can be used [52], or the system integrator can easily plug manually coded functions within the generated code.

Code generation is implemented as part of APP4MC [39], the Eclipse-based tooling platform for Amalthea models.

The existing infrastructure provides parsing functionality to be used in Java projects in APP4MC. Template-based code generation is implemented via Xtend [10] to generate the necessary configuration and code (in the *C* programming language) for the application on each QNX node. Our code generation mechanism extends the one in [8]. In contrast to the application characteristics considered here, the work in [8] considers APS partitions [23] to provide timing isolation, and communication is based on the QNX synchronous message passing mechanism [9], while runtime monitoring and orchestration have not been explored. Instead, we focus on a more cross-compatible approach leveraging mechanisms available on POSIX operating systems.

The framework generates only minimal files representing the data exchanged among threads, the registration of all application threads, and stubs for user implementation of each thread’s application logic. This information is provided through the Amalthea model. For testing purposes, the thread’s application logic can also be generated so that execution time is emulated with a configurable uncertainty compared to the specified execution time value.

**Execution.** Each QNX node hosts the code of all application threads in the system. This avoids the need for expensive migration operations [47] (which can take up to several seconds [1]) between nodes during a reconfiguration, which is suitable for small-scale edge systems with nodes running an RTOS and subject to stringent timing constraints, as those considered in this paper. All application threads are based on the same thread template and provide the necessary logic to realize all possible application states. The state variable of a thread is used to indicate the current operating state of the thread and is only modified by the node runtime manager. If a thread is in EXECUTE state, the user logic is executed periodically. This includes reading global input data to thread local copies and writing thread local output data to their global counterparts. This process is hidden from the user code and allows the user code to be agnostic of any framework knowledge. Threads in the COPY-IN state decode the data received through the network and write the updated values to the memory of the node. Finally, a thread is INACTIVE on a node when it is in mode EXECUTE on one of the other nodes, and no thread on the same node consumes the data it produces. The INACTIVE state is realized with a dedicated semaphore of the thread that is only posted by the manager thread of the node when the thread state changes again.

**Communication.** Several forms of communication are realized in accordance with the system model of Section 3.

Communication within a single node is realized using shared memory that is accessed by the communicating threads. A reading thread creates local copies of the input data at the start of its execution while writing threads update the global variables at the end of their execution. Thus, the implicit communication model is used [41].

Communication across node boundaries is always data-driven and realized via messages over the network. If two threads communicate across node boundaries, we use a

copy-in thread on the receiver node, as discussed in Section 3.

**Monitoring and Orchestration.** To realize the orchestration, each QNX node implements a manager component that collects runtime statistics and is able to reconfigure the application at runtime (see Fig. 2). A central manager node in the network periodically collects the runtime statistics of the QNX nodes and triggers the runtime decision-making procedure if needed. Any system reconfiguration is then communicated to the manager components on the QNX nodes that reconfigure the user application accordingly. The system manager may reside on any QNX node but can also be implemented on a separate node (e.g., running Linux, to leverage a vast amount of compatible optimization tools). To collect the runtime statistics, each thread records the execution time for each job. The largest per-mode observed execution time is stored.

**Implementation Extensions.** The prototype implementation discussed in this section is a solid baseline for evaluating the monitoring-based runtime decision-making approach advocated by this paper. Nonetheless, there is room for extending it in future work. The prototype assumes all the threads are stateless; however, the discussed thread structure is compatible with extensions to stateful applications, e.g., implementing job-checkpointing [46] for migration that could also allow for the local state of each thread. Support for migration operations can also be included [47], e.g., in case the footprint of the distributed application is too large compared to the memory size of each node. However, we show next in the evaluation that this is not the case for possible reference applications. Finally, possible future extensions may include the monitoring of network delays and devising reaction policies to cope with network-dependent overruns that are, however, not the focus of this paper. All these extensions of the implementation remain compatible with our approach.

## 9. Evaluation

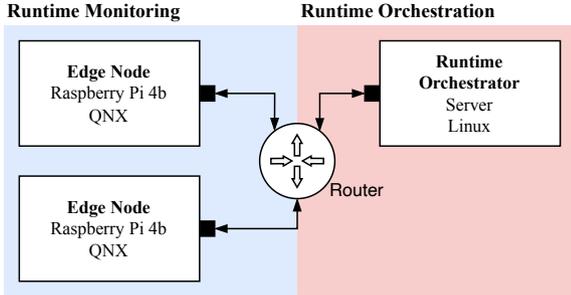
We evaluate the proposed approach considering a target distributed system with two Raspberry Pi 4B with 4 cores and a Gigabit Ethernet network card, using the QNX 7.1 Software Development Platform (SDP), and an orchestrator node consisting of an Intel Xeon Silver 4114 processor with 10 cores (20 threads) at 2.2 Ghz and 32GB RAM, running Linux. The three platforms can be representative of a realistic edge architecture, composed of more resource-constrained embedded systems (the Raspberry Pi 4B) and a more powerful Xeon node, which is used in state-of-the-art edge server solutions (e.g., the Dell PowerEdge solution [26]). The optimization problem has been implemented in C++ and solved with IBM CPLEX, an industry-grade optimization toolkit, on the Linux node. The orchestrator node runs Linux on an Intel architecture to ensure compatibility with CPLEX because it has limited compatibility with ARM platforms and is not compatible with QNX. Thus, for the experiment, the proposed monitoring and decision-making

ID	Name	$T_i^0$	$e_i$	$c'_i$	MODE-WEIGHTED			RD-MAX				LT-MAX				
					$p_k$	$n_x$	$R_i$	mode	$p_k$	$n_x$	$R_i$	mode	$p_k$	$n_x$	$R_i$	mode
$\tau_0$	Lidar Detection 1	33	11	H	1	0	11	0	0	1	11	1	1	0	11	1
$\tau_1$	Lidar Detection 2	33	11	H	0	0	31	0	0	1	22	1	2	0	11	1
$\tau_2$	Park. Detection 1	66	35	H	3	1	35.1	0	0	0	35	1	0	0	40	1
$\tau_3$	Park. Detection 2	66	35	H	0	1	63	0	1	0	35	0	3	1	90	1
$\tau_4$	Camera Detection 1	200	120	H	2	0	192	0	2	0	120	1	0	1	184	0
$\tau_5$	Camera Detection 2	200	120	H	1	0	186	0	3	1	120.06	1	1	1	190	0
$\tau_6$	Communication	10	1	M	2	0	1	0	2	1	2.63	1	0	0	1	0
$\tau_7$	EKF	15	4	H	0	1	4	0	2	1	6.63	1	0	1	4	0
$\tau_8$	Planner	15	12	H	3	0	12	0	1	1	12	1	1	1	12	0
$\tau_9$	SFM	33	15	M	1	1	15	0	3	0	25.71	1	2	0	37	2
$\tau_{10}$	Localization	400	139	H	0	0	216.32	0	0	1	271	1	3	0	139	0
$\tau_{11}$	Data Logging	300	100	L	1	1	205	0	3	0	160	0	1	1	532	3
$\tau_{12}$	Customer Notification	80	30	L	3	1	65.08	0	2	1	48.63	2	2	1	30.14	1
$\tau_{13}$	Park Reserv. Mgmt.	110	40	L	2	1	103	0	0	0	110	3	1	0	84	3
$\tau_{14}$	System Updates	110	20	L	2	1	80	0	2	1	98.64	3	2	1	130.57	3

**Table 6**

Case study parameters and solutions of the initial allocation problem. Times are in milliseconds.

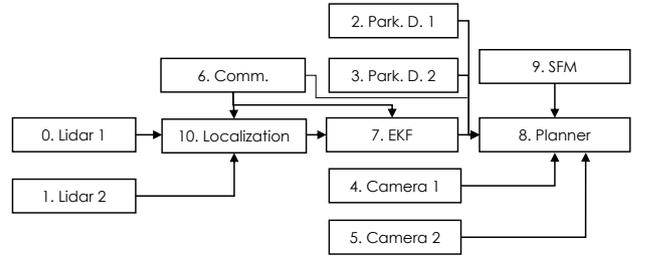
infrastructure, as shown in Fig. 2, is instantiated by the hardware topology shown in Fig. 4. As discussed in Section 2, the proposed framework brings together a unique set of properties that have not been addressed together by related work. Hence, in the following, we evaluate the proposed framework without explicit comparison to other approaches.



**Figure 4:** Hardware setup used for the evaluations.

For the application, we envision as a practical use case for the approaches of this paper an automated valet parking [12] edge system, which we prototype by hybridizing two related publicly available case studies: the WATERS 2019 Challenge by Bosch [34] and the TACLe Benchmark [28].

The WATERS 2019 contains an application for autonomous driving. The automated valet parking application shares most of the needed functionalities with the WATERS 2019, such as path planning, localization, and detection. It has been enriched to duplicate the detection-related threads (to mimic the fact that an automated parking area leverages multiple and diverse detection sources) and to add data logging, customer notification, parking reservation management, and system updates, borrowing the parameters from threads of the TACLe Benchmark. Event chains are shown in Fig. 5. Only threads from the WATERS 2019 are included in chains, while those coming from TACLe are considered as additional workloads for optional functionalities and behave as independent threads.



**Figure 5:** Chains in the case study.

The TACLe Benchmark [28] does not report periods; therefore, we used the same periods that have been associated with TACLe benchmark threads in [18]. Additionally, the case studies do not provide deadlines, priorities, and multiple modes, which are set as follows. Priorities are set according to the rate-monotonic assignment, and deadlines are set as  $D_i^d = e_{i,k}^d + \beta \cdot (T_i^d - e_{i,k}^d)$ , with  $\beta \in [0, 1]$ . In this way, if  $\beta = 1$ ,  $D_i^d = T_i^d$ , and if  $\beta = 0$ ,  $D_i^d = e_{i,k}^d$ . All threads are provided with an additional degraded mode (not shown in the table) in which the execution times remain the same, but periods are multiplied by a factor  $\alpha = 1.5$ ; Medium and low criticality threads have an additional degraded mode with  $\alpha = 2$ ; Low criticality threads have a final degraded mode with  $\alpha = 2.5$ . WOETs are kept unaltered. The application structure and values are reported in Table 6 (first four columns). From the coarse-grained criticalities  $c'_i$  in Table 6, classified as high (H), medium (M), and low (L), the criticality  $c_i$  is derived by resolving ties in rate-monotonic order.

**Evaluation of copy-in threads execution times.** The WOETs of copy-in threads are not included in the WATERS 2019 data since their presence refers to a design choice of this paper. However, the WATERS 2019 model reports the size of the data exchanged among threads, which are also used to determine the communication delays  $\lambda_{(i,j)}^{x,y}$  according to the network data rate. We performed a set of experiments on the Raspberry Pi platforms to evaluate such execution

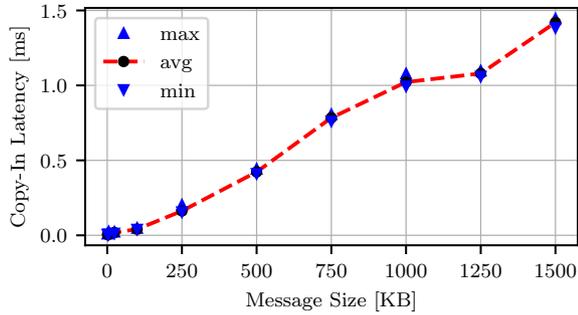


Figure 6: WOET of the copy-in thread with varying data size.

Table 7

Footprint of the application, in byte.

Application	text	data	bss	total
Case Study	52037	3884	18149336	18205257

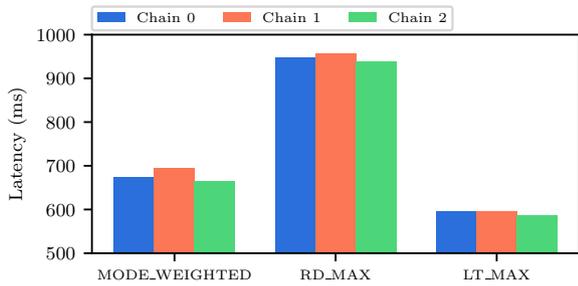


Figure 7: Chain latencies with different obj. functions.

times based on the specified data sizes. Fig. 6 reports the minimum, maximum, and average execution time of the copy-in thread for varying data sizes. For each data point, 500 samples are collected.

Fig. 6 corroborates one of the design choices of this paper: the usage of time-driven communication, with the only exception of copy-in threads, to minimize additional interfering activations in the response-time analysis due to data-driven communications [37] (e.g., in the second summation of Eq. (3)). Indeed, execution times of copy-in threads are much shorter (on average) than those of the other threads (reported in Table 6), thus making it easier to satisfy the schedulability.

**Memory Size.** Table 7 reports the memory footprint of the case study application. On each node, 18.2 MB of memory is required. This includes all global communication buffers, the task local communication buffers, as well as the complete code. For the case study application, the memory size is negligible compared to the total memory available on the used platform (8 GB of RAM), which is representative of the type of platforms the approach targets.

**Initial Design Experiments.** Table 6 reports the allocation of each thread to cores (col.  $p_k$ ) and nodes ( $n_x$ ), its WCRT ( $R_i$ ), and the selected operating mode for each of the three

considered objective functions (see Section 6), considering  $\beta = 0.9$ . All optimization objectives have advantages and disadvantages. For example, with `MODE_WEIGHTED`, the solver is able to find a solution with all threads at their best functional mode (i.e., mode 0). However, it can be generally less robust than `RD_MAX`, which tries to minimize the ratio between WCRT bound and the deadline to avoid that misestimations of the WOET can lead to an unschedulable condition. On the other hand, with `RD_MAX` some threads run in a less accurate mode (modes 1, 2, or 3). Finally, `LT_MAX` privileges mostly threads involved in thread chains, especially in computationally expensive ones. Other threads (e.g.,  $\tau_{11}$ ,  $\tau_{13}$ , and  $\tau_{14}$ ) are assigned to a less accurate mode. Fig. 7 shows the latencies of the three longest chains of Fig. 5. Chain 0, 1, and 2 in the figure correspond to the  $\gamma_0 = (\tau_0, \tau_{10}, \tau_7, \tau_8)$ ,  $\gamma_1 = (\tau_1, \tau_{10}, \tau_7, \tau_8)$ , and  $\gamma_3 = (\tau_6, \tau_{10}, \tau_7, \tau_8)$ , respectively. As expected, `LT_MAX` provides the best (shortest) latencies. Different objective functions are also characterized by different runtimes: `MODE_WEIGHTED`, `RD_MAX`, and `LT_MAX` find the optimal solutions within 1223.13, 37.82, and 55.29 seconds, respectively. Furthermore, the solver is able to find a feasible solution within less than 3 seconds. All runtimes are largely compatible with typical offline design activities.

**Runtime Decision-Making Experiments.** For this experiment, threads execute for a fixed amount of time specified by the WOET. All experiments are performed over a runtime of 30 s. The monitoring period of the central orchestrator is set to 100 ms. We report on five different variants of the decision-making policies that a designer can apply. For all settings, the solution with objective `MODE_WEIGHTED` is the starting configuration. The cases assume threads to overrun their WOET estimate: as previously discussed, the new WOET is assumed to be permanent since it is likely due to the thread being subject to a different working condition that was not experienced during testing, or due to new data-dependent code paths that were never executed.

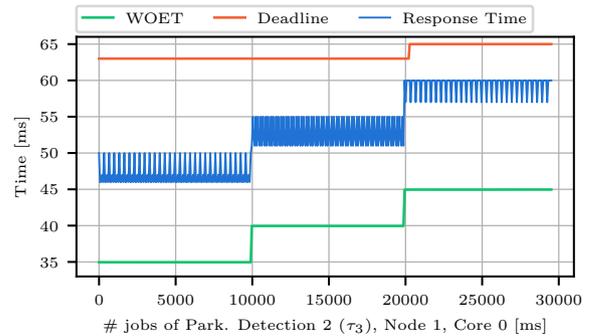
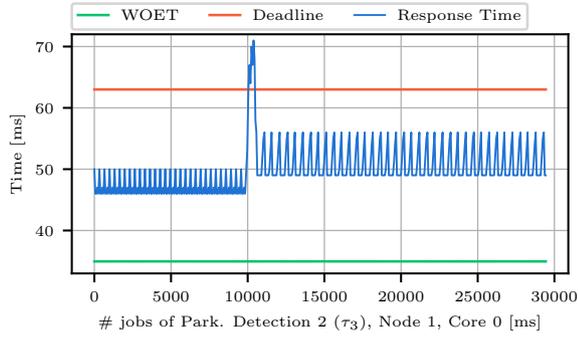
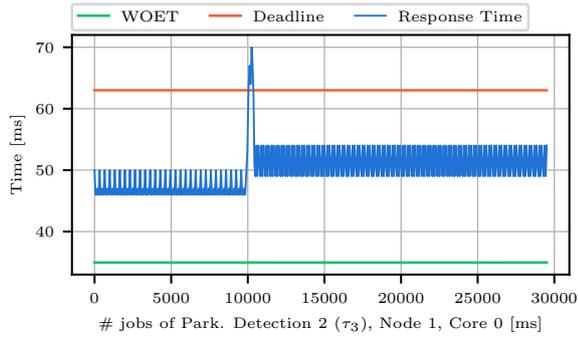


Figure 8: Task Parking Detection ( $\tau_3$ ) with runtime decision-making measurements and deadline inflation.

Fig. 8 considers the deadline inflation strategy. The WOET of  $\tau_3$  first reaches 40 ms (from 35) at  $t = 10$  s and then 45 ms at  $t = 20$  s.  $\tau_3$  is allocated to  $p_0, n_1$ , together with  $\tau_7$ , which has a higher priority. When the WOET reaches



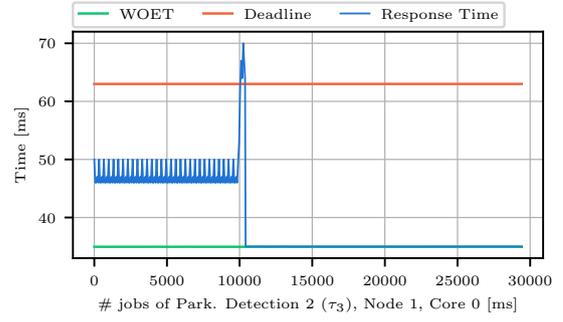
**Figure 9:** Task Parking Detection ( $\tau_3$ ) with runtime decision-making measurements and mode relaxation (period/deadline).



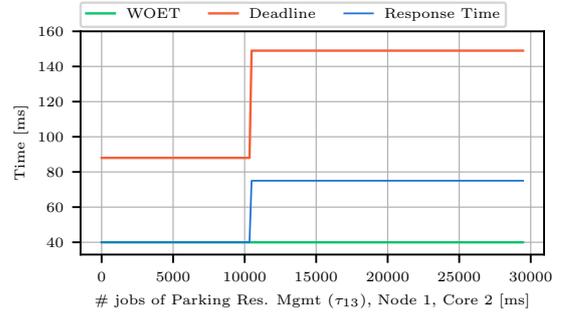
**Figure 10:** Task Parking Detection ( $\tau_3$ ) with runtime decision-making measurements and mode relaxation (execution time).

40 ms, the solver finds a better WCRT candidate that still satisfies the current deadline. Indeed, due to the formulation of Constraint 5(d), the solver finds any WCRT candidate that satisfies the constraints, not necessarily the smallest one. In this case, when the solver runs again with a bigger WOET for  $\tau_3$ , the WCRT candidate selected in the previous run becomes invalid, and another (still schedulable) candidate, corresponding to a different checkpoint  $v_{i,g}$ , is instead selected. When the WOET reaches 45 ms, schedulability requires a deadline inflation for  $\tau_3$ .

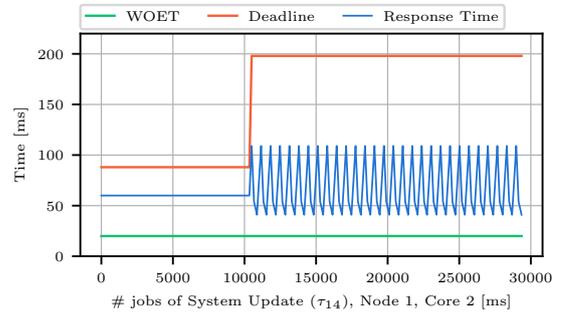
In all other settings, the WOET of thread  $\tau_7$  changes from 4 ms to 7 ms at  $t = 10$  s, which leads to  $\tau_3$ 's unschedulability, and different runtime decision-making strategies are compared. Deadline inflation alone does not lead to a schedulable solution in this case. In Fig. 9, the solver then applies mode relaxation and finds a solution where the mode of  $\tau_7$  is changed from 0 to 1. This is visible in the recorded response time values. After the mode switch of  $\tau_7$ , the interference faced by  $\tau_3$  is different due to the different period of  $\tau_7$ . In this setting,  $\tau_3$  misses six deadlines before reconfiguration. Fig. 10 similarly achieves the same result, but by considering a variation of the experimental setup in which  $\tau_7$  keeps the same period in the new mode, but the WOET is 4.56 ms, resulting in the same utilization of  $\tau_7$  as in Fig. 9 after reconfiguration.



(a) Task Parking Detection 2 ( $\tau_3$ ).



(b) Task Parking Reservation Management ( $\tau_{13}$ ).

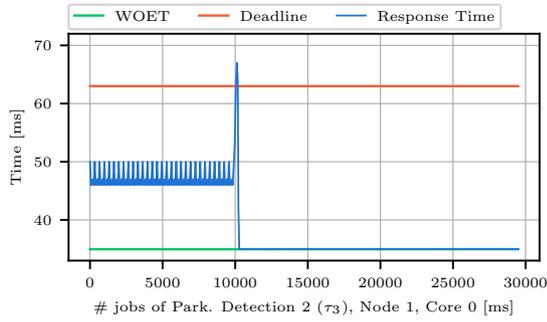


(c) Task System Update ( $\tau_{14}$ ).

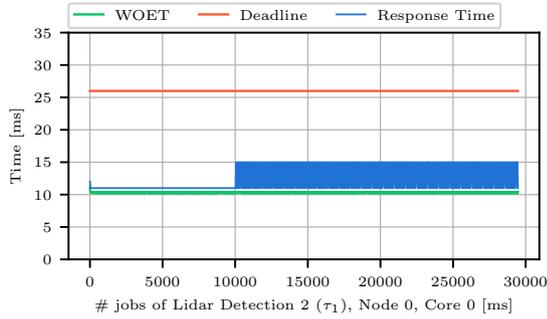
**Figure 11:** Runtime decision-making measurements with local reallocation and mode relaxation.

Fig. 11 considers instead the case in which the orchestrator reallocates the affected thread on the same node and can also change the mode of lower criticality threads. The solver finds a solution where the mode of  $\tau_{13}$  and  $\tau_{14}$  on  $p_2$  are degraded to mode 1 and 2, respectively. This allows  $\tau_7$  to be reallocated to  $p_2$ . Fig. 11 shows the affected threads.  $\tau_3$  (top) misses 4 deadlines and experiences no longer interference after  $\tau_7$  is reallocated to  $p_2$ .  $\tau_{13}$  and  $\tau_{14}$  (middle and bottom) experience more interference after moving  $\tau_7$  to the same core. Due to the degraded mode, both threads always meet their deadlines.

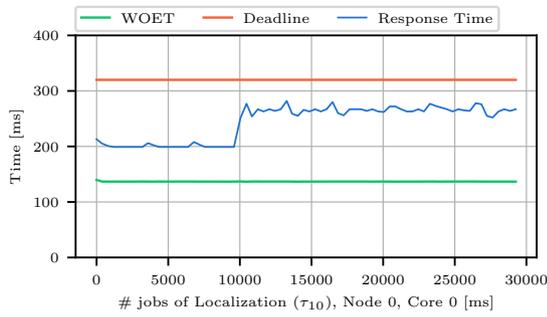
Fig. 12 considers a different orchestrator configuration in which the solver is allowed to orchestrate threads between nodes but can only change the mode of the thread that experiences a larger WOET. The solver relocates  $\tau_7$  to  $p_0$  of  $n_0$  and changes its mode to 1. Since  $\tau_7$  is now allocated to  $n_0$ , its copy-in threads on  $n_1$  can also be removed. Before  $\tau_7$



(a) Task Parking Detection 2 ( $\tau_3$ ).



(b) Task Lidar Detection 2 ( $\tau_1$ ).



(c) Task Localization ( $\tau_{10}$ ).

**Figure 12:** Runtime decision-making measurements with global reallocation and mode relaxation.

is relocated,  $\tau_3$  misses two deadlines but no longer receives interference afterward (top).

**Running times.** In all the runtime experiments, the time required by the solver is in the range [60, 667] ms. Overall, these runtimes are compatible with the timeframe of a soft real-time system that can accept some temporary deadline misses as long as schedulability is restored in a steady state.

## 10. Conclusions

This paper addressed the problem of the uncertainty in the execution times in modern interconnected heterogeneous platforms with a perspective based on well-consolidated real-time analysis for partitioned fixed-priority scheduling paired with a monitoring and analysis-based run-time decision-making approach providing the key property of

guaranteeing a determined degradation order configurable by the designer.

Future work will extend the approach to consider resource reservation and hardware accelerators, as well as incorporate more sophisticated (but harder to encode in a MILP) analysis techniques for end-to-end latency of task chains in the optimization problem [30]. Further directions involve optimizing performance restoration in case of the detection of temporary overruns with, e.g., introducing a timeout at the orchestrator level to restore the previous WOET estimate in case the new WOET was never detected again in the given timespan. Finally, we would like to investigate lightweight decision-making strategies for even more resource-constrained nodes, e.g., by using heuristic algorithms, and formally model the execution time of the decision-making task to allow executing it together with time-sensitive workloads while still allowing to provide predictable response times.

## References

- [1] Akoush, S., Sohan, R., Rice, A., Moore, A.W., Hopper, A., 2010. Predicting the performance of virtual machine migration, in: 2010 IEEE international symposium on modeling, analysis and simulation of computer and telecommunication systems, IEEE. pp. 37–46.
- [2] Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D., 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed, in: 2017 IEEE Real-Time Systems Symposium (RTSS), IEEE. pp. 104–115.
- [3] Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J., 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal* 8, 284–292.
- [4] Baek, H., Shin, K.G., Lee, J., 2020. Response-time analysis for multi-mode tasks in real-time multiprocessor systems. *IEEE Access* 8, 86111–86129.
- [5] Baruah, S.K., 2004. Partitioning real-time tasks among heterogeneous multiprocessors, in: International Conference on Parallel Processing, 2004. ICPP 2004., IEEE. pp. 467–474.
- [6] Baruah, S.K., Burns, A., Davis, R.I., 2011. Response-time analysis for mixed criticality systems, in: 2011 IEEE 32nd Real-Time Systems Symposium, IEEE. pp. 34–43.
- [7] Bate, I., Emberson, P., 2006. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems, in: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06), pp. 221–230.
- [8] Becker, M., Casini, D., 2024. The MATERIAL framework: Modeling and automatic code generation of edge real-time applications under the QNX RTOS. *Journal of Systems Architecture* 154, 103219.
- [9] Becker, M., Dasari, D., Casini, D., 2023. On the QNX IPC: Assessing predictability for local and distributed real-time systems, in: 2023 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).
- [10] Bettini, L., 2016. Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd.
- [11] Bini, E., Buttazzo, G., 2004. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers* 53, 1462–1473.
- [12] Bosch, 2024. Automated valet parking. URL: <https://www.bosch-mobility.com/en/solutions/parking/automated-valet-parking/>.
- [13] Burns, A., Baruah, S., 2013. Towards a more practical model for mixed criticality systems, in: Workshop on Mixed-Criticality Systems (colocated with RTSS).
- [14] Burns, A., Davis, R.I., 2017. A survey of research into mixed criticality systems. *ACM Comput. Surv.* 50.

- [15] Buttazzo, G.C., Lipari, G., Caccamo, M., Abeni, L., 2002. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers* 51, 289–302.
- [16] Caccamo, M., Buttazzo, G., Sha, L., 2002. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers* 51, 835–849.
- [17] Casini, D., Biondi, A., Buttazzo, G., 2020a. Task splitting and load balancing of dynamic real-time workloads for semi-partitioned EDF. *IEEE Transactions on Computers* 70, 2168–2181.
- [18] Casini, D., Biondi, A., Buttazzo, G., 2020b. Timing isolation and improved scheduling of deep neural networks for real-time systems. *Software: Practice and Experience* 50, 1760–1777.
- [19] Casini, D., Pazzaglia, P., Biondi, A., Di Natale, M., 2022. Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration. *Journal of Systems Architecture* 124, 102416.
- [20] Chen, H., Cheng, A.M.K., Kuo, Y.W., 2011. Assigning real-time tasks to heterogeneous processors by applying ant colony optimization. *Journal of Parallel and Distributed computing* 71, 132–142.
- [21] Cucinotta, T., Amory, A., Ara, G., Paladino, F., Natale, M.D., 2023. Multi-criteria optimization of real-time dags on heterogeneous platforms under P-EDF. *ACM Transactions on Embedded Computing Systems*.
- [22] Dasari, D., Becker, M., Casini, D., Blaß, T., 2022. End-to-end analysis of event chains under the QNX adaptive partitioning scheduler, in: *2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [23] Dasari, D., Hamann, A., Broede, H., Pressler, M., Ziegenbein, D., 2021. Brief industry paper: Dissecting the QNX adaptive partitioning scheduler, in: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 477–480.
- [24] Davare, A., Zhu, Q., Di Natale, M., Pinello, C., Kanajan, S., Sangiovanni-Vincentelli, A., 2007. Period optimization for hard real-time distributed automotive systems, in: *2007 44th ACM/IEEE Design Automation Conference*.
- [25] Davis, R.I., Burns, A., Bate, I., 2022. Compensating adaptive mixed criticality scheduling, in: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pp. 81–93.
- [26] Dell Technologies, 2024. Dell PowerEdge XE2420 Spec Sheet. URL: [https://i.dell.com/sites/csdocuments/Product\\_Docs/en/PowerEdge-XE2420-Spec-Sheet.pdf](https://i.dell.com/sites/csdocuments/Product_Docs/en/PowerEdge-XE2420-Spec-Sheet.pdf). accessed: 2024-11-25.
- [27] Durrieu, G., Fohler, G., Gala, G., Girbal, S., Pérez, D.G., Noulard, E., Pagetti, C., Pérez, S., 2016. Dreams about reconfiguration and adaptation in avionics, in: *ERTS 2016*.
- [28] Falk, H., Altmeyer, S., Hellinckx, P., et al., 2016. TACLeBench: A benchmark collection to support worst-case execution time research, in: *16th International Workshop on Worst-Case Execution Time Analysis*.
- [29] Gifford, R., Phan, L.T.X., 2022. Multi-mode on multi-core: Making the best of both worlds with omni, in: *2022 IEEE Real-Time Systems Symposium (RTSS)*, IEEE. pp. 118–131.
- [30] Girault, A., Prévot, C., Quinton, S., Henia, R., Sordon, N., 2018. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE transactions on computer-aided design of integrated circuits and systems* 37, 2578–2589.
- [31] González, O., Shrikumar, H., Stankovic, J.A., Ramamritham, K., 1997. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling, in: *Proceedings Real-Time Systems Symposium*, IEEE. pp. 79–89.
- [32] Griva, I., Nash, S.G., Sofer, A., 2008. *Linear and Nonlinear Optimization 2nd Edition*. SIAM.
- [33] Guo, Z., Yang, K., Vaidhun, S., Arefin, S., Das, S.K., Xiong, H., 2018. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate, in: *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 373–383.
- [34] Hamann, A., Dasari, D., Wurst, F., Saudo, I., Capodieci, N., Burgio, P., Bertogna, M., 2019. Waters industrial challenge 2019, in: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.
- [35] Hardy, D., Rouxel, B., Pauat, I., 2017. The Heptane static worst-case execution time estimation tool, in: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [36] Hassan, M., Pellizzoni, R., 2018. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 2323–2336.
- [37] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R., 2005. System level performance analysis - the SymTA/S approach. *IEEE Proceedings - Computers and Digital Techniques*.
- [38] Henzinger, T.A., Horowitz, B., Kirsch, C.M., 2001. Giotto: A time-triggered language for embedded programming, in: *International Workshop on Embedded Software*, Springer. pp. 166–184.
- [39] Höttger, R., Mackamul, H., Sailer, A., Steghöfer, J.P., Tessmer, J., 2017. App4mc: Application platform project for multi-and many-core systems. *it-Information Technology* 59, 243–251.
- [40] Kampmann, A., Alrifae, B., Kohout, M., Wüstenberg, A., Woopen, T., Nolte, M., Eckstein, L., Kowalewski, S., 2019. A dynamic service-oriented software architecture for highly automated vehicles, in: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE. pp. 2101–2108.
- [41] Kramer, S., Ziegenbein, D., Hamann, A., 2015. Real world automotive benchmarks for free, in: *6th Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.
- [42] Kritikakou, A., Rochange, C., Faugère, M., Pagetti, C., Roy, M., Girbal, S., Pérez, D.G., 2014. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems, in: *22nd International Conference on Real-Time Networks and Systems*, pp. 139–148.
- [43] Kritikakou, A., Skalistis, S., 2023. Mitigating mode-switch through run-time computation of response time. *ACM Transactions on Design Automation of Electronic Systems* 28, 1–26.
- [44] Lakshmanan, K., De Niz, D., Rajkumar, R., Moreno, G., 2010. Resource allocation in distributed mixed-criticality cyber-physical systems, in: *2010 IEEE 30th International Conference on Distributed Computing Systems*, IEEE. pp. 169–178.
- [45] Lehoczky, J., Sha, L., Ding, Y., 1989. The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: *[1989] Proceedings. Real-Time Systems Symposium*.
- [46] Litzkow, M., Solomon, M., 1999. Supporting checkpointing and process migration outside the UNIX kernel. *ACM Press/Addison-Wesley Publishing Co.* p. 154–162.
- [47] Milošević, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S., 2000. Process migration. *ACM Comput. Surv.* 32, 241–299.
- [48] Monaco, G., Gala, G., Fohler, G., 2023. Shared resource orchestration extensions for kubernetes to support real-time cloud containers, in: *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE. pp. 97–106.
- [49] Möstl, M., Nolte, M., Schlatow, J., Ernst, R., 2019. Controlling concurrent change-a multiview approach toward updatable vehicle automation systems, in: *Workshop on Autonomous Systems Design (ASD 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [50] Möstl, M., Schlatow, J., Ernst, R., Dutt, N., Nassar, A., Rahmani, A., Kurdahi, F.J., Wild, T., Sadighi, A., Herkersdorf, A., 2018. Platform-centric self-awareness as a key enabler for controlling changes in cps. *Proceedings of the IEEE* 106, 1543–1567.
- [51] Möstl, M., Schlatow, J., Ernst, R., Hoffmann, H., Merchant, A., Shraer, A., 2016. Self-aware systems for the internet-of-things, in: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 1–9.
- [52] Neema, S., Kalmar, Z., Shi, F., Vizhanyo, A., Karsai, G., 2005. A visually-specified code generator for simulink/stateflow, in: *2005*

- IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), IEEE. pp. 275–277.
- [53] Papadopoulos, A., Bini, E., Baruah, S., Burns, A., 2018. Adaptmc: A control-theoretic approach for achieving resilience in mixed-criticality systems, in: Proceeding ECRTS Conference, LIPICS. pp. 14–1.
- [54] Park, M., Park, H., 2012. An efficient test method for rate monotonic schedulability. *IEEE Transactions on Computers* 63, 1309–1315.
- [55] Pazzaglia, P., Biondi, A., Di Natale, M., 2019. Simple and general methods for fixed-priority schedulability in optimization problems, in: Proceedings of the International Conference on Design, Automation and Test in Europe (DATE 2019).
- [56] Peeck, J., Schlatow, J., Ernst, R., 2021. Online latency monitoring of time-sensitive event chains in safety-critical applications, in: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE. pp. 539–542.
- [57] Rashid, S.A., Nelissen, G., Tovar, E., 2020. Cache persistence-aware memory bus contention analysis for multicore systems, in: 2020 Design, Automation & Test in Europe Conference & Exhibition, pp. 442–447.
- [58] Real, J., Crespo, A., 2004. Mode change protocols for real-time systems: A survey and a new proposal. *Real-time systems* 26, 161–197.
- [59] Schlatow, J., Möstl, M., Ernst, R., 2019. Self-aware scheduling for mixed-criticality component-based systems, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE. pp. 267–278.
- [60] Schlatow, J., Schmidt, E., Ernst, R., 2021. Automating integration under emergent constraints for embedded systems. *SICS Software-Intensive Cyber-Physical Systems* 35, 185–199.
- [61] Shin, K.G., Meissner, C.L., 1999. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment, in: Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99, IEEE. pp. 29–36.
- [62] Sinha, S., West, R., Golchin, A., 2020. Pastime: Progress-aware scheduling for time-critical computing, in: 32nd Euromicro Conference on Real-Time Systems.
- [63] Stankovic, J.A., Lu, C., Son, S.H., Tao, G., 1999. The case for feedback control real-time scheduling, in: Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99, pp. 11–20.
- [64] Struhár, V., Craciunas, S.S., Ashjaei, M., Behnam, M., Papadopoulos, A.V., 2023. Hierarchical resource orchestration framework for real-time containers. *ACM Trans. Embed. Comput. Syst.* .
- [65] Tindell, K., Burns, A., Wellings, A.J., 1992. Mode changes in priority pre-emptively scheduled systems., in: RTSS, Citeseer. pp. 100–109.
- [66] Vestal, S., 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pp. 239–243.
- [67] Yang, K., Guo, Z., 2019. EDF-based mixed-criticality scheduling with graceful degradation by bounded lateness, in: 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–6.
- [68] Zeng, H., Di Natale, M., 2012. An efficient formulation of the real-time feasibility region for design optimization. *IEEE Transactions on Computers* 62, 644–661.
- [69] Zhu, Q., Yang, Y., Scholte, E., Natale, M.D., Sangiovanni-Vincentelli, A., 2009. Optimizing extensibility in hard real-time distributed systems, in: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 275–284.
- [70] Zhu, Q., Zeng, H., Zheng, W., Natale, M.D., Sangiovanni-Vincentelli, A., 2013. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Trans. Embed. Comput. Syst.* 11.
- [71] Zou, J., Dai, X., McDermid, J.A., 2023. Context-aware graceful degradation for mixed-criticality scheduling in autonomous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* .