The Concept of Unschedulability Core for Optimizing Real-Time Systems with Fixed-Priority Scheduling

Yecheng Zhao, and Haibo Zeng, *Member, IEEE*

Abstract—In the design optimization of real-time systems scheduled with fixed priority, schedulability analysis is used to define the feasibility region within which tasks meet their deadlines, so that optimization algorithms can find the best solution within the region. However, the complexity of schedulability analysis techniques often makes it difficult to leverage existing optimization frameworks and scale to large designs. In this paper, we propose the concept of unschedulability core, a compact representation of the schedulability conditions, and develop efficient algorithms for its calculation. We present a new optimization framework that leverages such a concept. We show that this concept is applicable to a range of optimization problems, for example, when the decision variables include the task priority assignment and the selection of mechanisms protecting shared buffers. Experimental results on two case studies demonstrate that the new optimization procedure maintains the optimality of the solutions, but is a few orders of magnitude faster than other exact algorithms (branch-and-bound, integer linear programming).

Index Terms—Real-Time Systems, Fixed-Priority Scheduling, Design Optimization, Audsley's Algorithm, Unschedulability Core.

1 Introduction

The design of real-time embedded systems is often subject to many requirements and objectives in addition to real-time schedulability constraints, including limited resources (e.g., memory), cost, quality of control, and energy consumption. For example, the automotive industry is hard pressed to deliver products with low cost, due to the large volume and the competitive international market [1]. Similarly, technology innovations for medical devices are mainly driven by reduced size, weight, and power (SWaP) [2]. In these application domains, it is important to perform *design optimization* in order to find the best design (i.e., optimized according to an objective function) while satisfying all the critical requirements.

Formally, a design optimization problem is defined by decision variables, constraints, and an objective function. The *decision variables* represent the set of design choices under the designers' control. The set of *constraints* forms the feasibility region, the domain of allowed values for the decision variables. The *objective function* characterizes the optimization goal. In general, the optimal design can be obtained by solving an optimization problem where the objective function is optimized within the feasibility region. For real-time systems, the *feasibility region* (also called *schedulability region* if concerning only real-time schedulability) must only contain the designs that satisfy the *schedulability constraints* where each task completes before its deadline.

In this paper, we consider the design optimization for real-time systems scheduled with fixed priority. The decision variables may include task priority assignment and the selection of mechanisms to ensure data consistency of shared variables. Besides real-time schedulability, these problems often contain constraints or an objective function related to other metrics (such as memory, power, thermal, etc.). Typically, this makes the problem complexity NP-hard, including the two case studies in this paper: the optimization of mixed-criticality Simulink models with Adaptive Mixed Criticality (AMC) scheduling (Section 7.1), and the memory minimization in the implementation of automotive AUTOSAR models (Section 7.2). Below we provide a summary of related work, focusing on the underlining approach but not intended to be complete.

• *Y. Zhao and H. Zeng are with the Department of ECE, Virginia Tech, Blacksburg, VA, 24060. E-mail: {zyecheng, hbzeng}@vt.edu*
Source code is available at https://github.com/zyecheng/UnschedCore. This work is partially supported by NSF Grants No. 1739318 and No. 1812963.

1.1 Related Work

There is a large body of work on priority assignment for real-time systems with fixed priority scheduling. In particular, Audsley's algorithm [3] is proven to be "optimal" for many task models and scheduling schemes, if the designer is only concerned to find a schedulable solution. See a recent survey by Davis et al. [4] on a complete list of applicable settings. However, if the design optimization problem contains constraints or an objective function related to other metrics (such as memory, power, thermal, etc.), Audsley's algorithm is no longer guaranteed to be optimal.

In general, the current approaches for optimizing priority assignment in complex design optimization problems (i.e., those without known polynomial-time optimal algorithms) can be classified into three categories. The *first* is based on meta heuristics such as simulated annealing (e.g., [5], [6]) and genetic algorithm (e.g., [7]). The *second* is to develop problem specific heuristics (e.g., [8], [9], [10]). These two categories do not have any guarantee on optimality.

The *third* category is to search for the exact optimum, often applying existing optimization frameworks such as branch-and-bound (BnB) (e.g., [11]), or integer linear pro-

gramming (ILP) (e.g., [12]). However, this approach typically suffers from scalability issues and may have difficulty to handle large industrial designs. For example, automotive engine control system contains over a hundred functional blocks [13], but the ILP based approach can only scale up to about 40 functional blocks (see Section 7). Furthermore, not all problems can easily be formulated in a particular framework due to the complexity of schedulability conditions. For example, the exact schedulability analysis for tasks with non-preemptive scheduling requires to check all the task instances in the busy period, but the number of instances is unknown a priori. Hence, it is difficult to formulate the exact schedulability constraints in ILP [14].

The above existing mindset is also followed by optimization of problems with other decision variables, such as the selection of mechanisms for protection of shared memory buffers [15], the mapping of functional blocks to tasks [16], and the use of rate transition buffers for semantics preservation in Simulink models [17]. Different from all existing work, our approach is to develop a domain-specific optimization framework that is optimal, scalable, yet still applicable to a large class of real-time systems.

1.2 Contributions and Paper Structure

In this paper, instead of directly reusing standard techniques (BnB, ILP, etc.), we present customized optimization techniques for real-time systems with fixed priority scheduling. Our framework can guarantee the optimality of the solution while drastically improving the scalability. Specifically, we make the following contributions:

- We propose the concept of **unschedulability core**, an abstraction of the schedulability conditions in real-time systems scheduled with fixed priority. It can be represented by a set of new and compact constraints to be learned efficiently during the execution of the optimization procedure (i.e., at runtime).
- We devise an optimization procedure that judiciously utilizes the unschedulability cores to drastically improve the scalability.
- We use two design optimization problems to illustrate the benefit of the proposed approach. The new unschedulability core guided optimization algorithm runs several orders of magnitude faster than other optimal algorithms (BnB, ILP) while maintaining the optimality of the solutions.

The rest of the paper is organized as follows. From Section 2 to Section 4, we first consider the optimization problems that assign priority orders to tasks. Specifically, Section 2 describes the task models and gives a formal definition of the problems that are suitable for the proposed approach. Section 3 defines the concept of unschedulability core that contains a set of partial priority orders among tasks, and studies its efficient calculation. Section 4 presents the optimization procedure that leverages the unschedulability cores for optimizing priority assignment, with proven properties on termination and optimality. In Section 5, we extend the framework to optimization problems with other decision variables. We generalize the concept of unschedulability core that allows to take problem-specific interpretations. We also discuss the applicability and efficiency for the proposed optimization framework. Section 6 gives two examples of application. Section 7 evaluates the effectiveness of the proposed approach with industrial case studies and synthetic systems. Finally, Section 8 concludes the paper.

2 Preliminary

We consider a real-time system scheduled by fixed priority. It consists of a set of tasks $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$. Each task τ_i is assumed to have a *unique* priority π_i (the higher the number, the higher the priority) to be assigned by the designer. The concept of unschedulability core applies to any systems scheduled with fixed priority. However, its application in design optimization is most effective when there is a simple algorithm to determine the existence of a schedulable priority assignment for a given task set. Hence, we consider a list of task models and scheduling schemes where Audsley's algorithm [3] is applicable (i.e., it can find a schedulable priority assignment if there exists one). The list, as summarized in [4], includes a large number of task models and scheduling schemes:

- The periodic task model, where independent tasks are scheduled on a single-core platform with preemptive scheduling. Each task is characterized by a tuple of parameters: T_i denotes the period; $D_i = T_i$ represents the implicit relative deadline; C_i denotes the worst-case execution time (WCET).
- Tasks with arbitrary deadlines, and/or static offsets.
- Probabilistic real-time systems where task WCETs are described by independent random variables [18].
- Systems scheduled with deferred preemption [19].
- Tasks modeled as arbitrary digraphs [20], where the vertices represent different kinds of jobs, and the edges represent the possible flows of control.
- Tasks accessing shared resources protected by semaphore locks to ensure mutual exclusion.

We note that Audsley's algorithm runs very efficiently: out of the possible $n!$ priority assignments, it only needs to explore $O(n^2)$ of them.

A priority assignment can be represented using a set of binary variables $\mathbf{P} = \{p_{i,j} | i \neq j, \tau_i, \tau_j \in \Gamma\}$ denoting the partial priority orders among tasks, where $p_{i,j}$ is defined as

$$p_{i,j} = \begin{cases} 1 & \pi_i > \pi_j, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

A priority assignment shall satisfy the antisymmetric and transitive properties: If τ_i has a higher priority than τ_j ($p_{i,j} = 1$), then τ_j has a lower priority than τ_i ($p_{j,i} = 0$); If τ_i has a higher priority than τ_j ($p_{i,j} = 1$) and τ_j has a higher priority than τ_k ($p_{j,k} = 1$), then τ_i must have a higher priority than τ_k ($p_{i,k} = 1$). These properties can be formally formulated as

$$\begin{array}{rll} \text{Antisymmetry:} & p_{i,j} + p_{j,i} = 1, & \forall i \neq j \\ \text{Transitivity:} & p_{i,j} + p_{j,k} \leq 1 + p_{i,k}, & \forall i \neq j \neq k \end{array} \tag{2}$$

We first focus on a design optimization problem where the decision variables $\mathbf{X}$ include the task priority assignment, i.e., $\mathbf{P} \subseteq \mathbf{X}$.

$$\begin{array}{ll} & \min\ C(\mathbf{X}) \\ s.t. & \text{system schedulability} \\ & \mathbf{F}(\mathbf{X}) \leq 0 \end{array} \tag{3}$$

TABLE 1: An Example Task System Γ_e

τ_i	T_i	D_i	C_i	τ_i	T_i	D_i	C_i
τ_1	10	10	2	τ_2	20	20	3
τ_3	40	40	16	τ_4	100	100	3
τ_5	200	200	17	τ_6	400	400	32

Here $C(\mathbf{X})$ is the objective function to be minimized, $\mathbf{F}(\mathbf{X}) \leq 0$ defines the set of additional constraints that the solutions in the feasibility region shall satisfy, including those in Equation (2).

3 The Concept of Unschedulability Core

Our technique is centered around the concept of unschedulability core. Intuitively, it is an irreducible set of partial priority orders that cause the system unschedulable. In this section, we first introduce its formal definition, and study its properties and usage in modeling the schedulability region. We then introduce an efficient algorithm for computing unschedualbility cores. We use an example system Γ_e configured as in Table 1 to illustrate, where all tasks are assumed to be *independent and preemptive*.

3.1 Definition of Unschedulability Core

Definition 1. A **partial priority order (PPO)**, denoted as $r_{i,j} \equiv (\pi_i > \pi_j) \equiv (p_{i,j} = 1)$, defines a priority order that τ_i has a higher priority than τ_j. A **PPO set** $\mathcal{R} = \{r_{i_1,j_1}, r_{i_2,j_2}, ..., r_{i_m,j_m}\}$ is a collection of one or more partial priority orders that are *consistent with the properties in Equation (2)*. The number of elements in $\mathcal{R}$ is defined as its **cardinality**, denoted as $|\mathcal{R}|$.

Definition 2. Let Γ be a task system and $\mathcal{R}$ be a PPO set on Γ. Γ is **$\mathcal{R}$-schedulable** if and only if there exists a feasible priority assignment $\mathcal{P}$ that respects all elements (i.e., all partial priority orders) in $\mathcal{R}$.

For convenience, we also say that "$\mathcal{R}$ is schedulable" when Γ is $\mathcal{R}$-schedulable, and similarly "$\mathcal{R}$ is unschedulable" when Γ is not $\mathcal{R}$-schedulable.

Example 1. Consider the system Γ_e in Table 1 and two PPO sets $\mathcal{R}_1 = \{r_{1,2}, r_{2,3}\}$, $\mathcal{R}_2 = \{r_{5,4}, r_{4,3}\}$. Γ_e is $\mathcal{R}_1$-schedulable, since the system is schedulable under rate-monotonic priority assignment which respects $\mathcal{R}_1$. However, Γ_e is not $\mathcal{R}_2$-schedulable: τ_1 must have a higher priority than τ_3 (due to $C_3 > D_1$), hence assigning τ_4 and τ_5 with higher priority than τ_3 will make τ_3 miss its deadline.

The following theorem intuitively states that if the system is schedulable for a PPO set, then the system is also schedulable for any of its subset.

Theorem 1. Let $\mathcal{R}$ and $\mathcal{R}'$ be two PPO sets on Γ such that $\mathcal{R}' \subseteq \mathcal{R}$. The following always holds

$$\Gamma \text{ is } \mathcal{R}\text{-schedulable} \implies \Gamma \text{ is } \mathcal{R}'\text{-schedulable} \quad (4)$$

The proof is straightforward as any priority assignment satisfying $\mathcal{R}$ must also satisfy $\mathcal{R}'$. Applying the law of contrapositive on Theorem 1, we have that for any $\mathcal{R}' \subseteq \mathcal{R}$,

$$\Gamma \text{ is not } \mathcal{R}'\text{-schedulable} \implies \Gamma \text{ is not } \mathcal{R}\text{-schedulable} \quad (5)$$

We now give the definition of unschedulability core. Intuitively, it is an unschedulable PPO set that is irreducible, such that removing any element from it allows a schedulable priority assignment.

Definition 3. Let Γ be a task system and $\mathcal{R}$ be a PPO set on Γ. $\mathcal{R}$ is an **unschedulability core** for Γ if and only if $\mathcal{R}$ satisfies the following two conditions:

- Γ is not $\mathcal{R}$-schedulable;
- $\forall \mathcal{R}' \subset \mathcal{R}$, Γ is $\mathcal{R}'$-schedulable.

Remark 1. By Theorem 1, the second condition in Definition 3 can be replaced by

- $\forall \mathcal{R}' \subset \mathcal{R}$ s.t. $|\mathcal{R}'| = |\mathcal{R}| - 1$, Γ is $\mathcal{R}'$-schedulable.

Example 2. Consider the PPO set $\mathcal{R}_3 = \{r_{5,4}, r_{4,3}, r_{3,6}\}$, which equivalently defines the priority order $\pi_5 > \pi_4 > \pi_3 > \pi_6$. Obviously, Γ_e is not $\mathcal{R}_3$-schedulable as it is not schedulable for the subset $\mathcal{R}_2 = \{r_{5,4}, r_{4,3}\}$ of $\mathcal{R}_3$ (see Example 1). However, $\mathcal{R}_3$ is not an unschedulability core since it has a proper subset $\mathcal{R}_2$ for which Γ_e is not schedulable. $\mathcal{R}_2$ is a valid unschedulability core, as for each of its proper subset, there exists a respecting feasible priority assignment: $\mathcal{P} = [\pi_1 > \pi_2 > \pi_3 > \pi_5 > \pi_4 > \pi_6]$ respects $\mathcal{R}_2^{(1)} = \{r_{5,4}\}$ and $\mathcal{R}_2^{(2)} = \emptyset$, and $\mathcal{P}' = [\pi_1 > \pi_2 > \pi_4 > \pi_3 > \pi_5 > \pi_6]$ respects $\mathcal{R}_2^{(3)} = \{r_{4,3}\}$.

Let $\mathcal{U}$ denote an unschedulability core. The constraint that the PPOs in $\mathcal{U}$ cannot be simultaneously satisfied is:

$$\sum_{r_{i,j} \in \mathcal{U}} p_{i,j} \leq |\mathcal{U}| - 1 \quad (6)$$

Remark 2. An unschedulability core $\mathcal{U}$ essentially gives a necessary condition for schedulability, that any feasible priority assignment shall differ from $\mathcal{U}$ for at least one partial priority order. Constraint (6) captures such requirement and is much more friendly to ILP solver. Its coefficients on the left hand side are all small integers (0 or 1), which in many cases makes the ILP solver more efficient [21].

We now prove that the set of *all* unschedulability cores is a necessary and sufficient condition that makes the system unschedulable.

Lemma 2. Let Γ be a *schedulable* task system and $\mathcal{R}$ be a PPO set on Γ. Γ is not $\mathcal{R}$-schedulable if and only if $\mathcal{R}$ contains at least one unschedulability core.

Proof. "If" Part: It is straightforward by the definition of unschedulability core and the result in Equation (5).
"Only If" Part: Proof by induction on the cardinality of $\mathcal{R}$.

Base case. Let $\mathcal{R}$ be any PPO set such that $|\mathcal{R}| = 1$ and Γ is not $\mathcal{R}$-schedulable. The only proper subset of $\mathcal{R}$ is $\mathcal{R}' = \emptyset$. Since Γ is schedulable, $\mathcal{R}$ itself is an unschedulability core.

Inductive step. Assume any PPO set of cardinality from 1 to $k-1$ such that Γ is not schedulable contains an unschedulability core. We prove that any $\mathcal{R}$ of cardinality k such that Γ is not $\mathcal{R}$-schedulable shall contain an unschedulability core. By Definition 3, there must exist $\mathcal{R}' \subset \mathcal{R}$ such that Γ is not $\mathcal{R}'$-schedulable (otherwise, $\mathcal{R}$ itself is an unschedulability core). Now we consider $\mathcal{R}'$, which has a cardinality smaller than k. By the assumption for the inductive step, $\mathcal{R}'$ contains an unschedulability core, so does $\mathcal{R}$. □

Algorithm 1 Computing One Unschedulability Core

```
1: function UNSCHEDCORE(Task set Γ, PPO set R)
2:     for each r ∈ R do
3:         if Γ is not R\{r}-schedulable then
4:             remove r from R
5:         end if
6:     end for
7:     return R
8: end function
```

Theorem 3. Let $\widetilde{\mathbb{U}}$ denote the complete set of unschedulability cores. The exact feasibility region can be represented by constraints (6) of all unschedulability cores in $\widetilde{\mathbb{U}}$, i.e.,

$$\sum_{\forall p_{i,j} \in \mathcal{U}} p_{i,j} \leq |\mathcal{U}| - 1, \quad \forall \mathcal{U} \in \widetilde{\mathbb{U}} \tag{7}$$

Proof. By Lemma 2, every infeasible priority assignment must contain at least one unschedulability core. Thus the space of feasible priority assignments can be obtained by guaranteeing the absence of any unschedulability cores, which is equivalent to constraint (7). □

Theorem 3 states that if all the unschedulability cores for the system are known, then we can formulate the exact schedulability region by adding constraint (6) for each unschedulability core. However, the number of unschedulability cores may be exponential to the number of tasks. Hence, it is inefficient to rely on the complete knowledge of the unschedulability cores. In the following, we develop procedures that judiciously and efficiently add a selective subset of unschedulability cores to gradually form the needed part of the schedulability region. Specifically, in the rest of this section, we present procedures (Algorithms 1–2) that, given an unschedulable priority assignment, efficiently calculate unschedulability cores. In the next section, we propose an unschedulability core guided optimization algorithm.

3.2 Computing Unschedulability Core

Algorithm 1 takes as inputs the task set Γ and a PPO set $\mathcal{R}$, where Γ is not $\mathcal{R}$-schedulable. It leverages Remark 1 and checks if those subsets of $\mathcal{R}$ with cardinality $|\mathcal{R}| - 1$ (i.e., one less element) can allow Γ schedulable. Hence, it iterates through and tries to remove each element r in $\mathcal{R}$. If the resulting PPO set still does not allow Γ to be schedulable, then r is removed. In the end, it will return one unschedulability core. Since the cardinality of $\mathcal{R}$ is $O(n^2)$, the number of iterations in Algorithm 1 is $O(n^2)$.

We now prove that the resulting PPO set $\mathcal{R}$ of Algorithm 1 is indeed an unschedulability core.

Theorem 4. Given a task set Γ and a PPO set $\mathcal{R}$ where Γ is not $\mathcal{R}$-schedulable, Algorithm 1 produces an unschedulability core $\mathcal{R}$ that satisfies Definition 3.

Proof. The first condition in Definition 3 is satisfied since the algorithm maintains that Γ is not $\mathcal{R}$-schedulable.

For the second condition, it is sufficient to show that the condition in Remark 1 is satisfied. Consider any r^* in the returned $\mathcal{R}$. While Algorithm 1 iterates on r^* at Line 2, the corresponding PPO set $\mathcal{R}^*$ must satisfy that $\mathcal{R}^* \backslash \{r^*\}$ is schedulable (otherwise r^* will be removed and cannot be in $\mathcal{R}$). Also, it must be $\mathcal{R} \subseteq \mathcal{R}^*$ since the later iterations will only delete elements from $\mathcal{R}^*$. Hence, by Theorem 1, Γ is $\mathcal{R} \backslash \{r^*\}$-schedulable. □

Algorithm 2 Computing Multiple Unschedulability Cores

```
1: function MULTIUNSCHEDCORE(Task set Γ, PPO set R,
   Number of Unschedulability cores k, Set of cores U)
2:     while |U| < k do
3:         ⟨status, R'⟩ = PERTURB(Γ, R, U)
4:         if status == false then
5:             return
6:         end if
7:         U = UNSCHEDCORE(Γ, R')
8:         U = U ∪ {U}
9:     end while
10: end function
11: function PERTURB(Task set Γ, PPO set R, Set of cores U)
12:     for each PPO combination {r_1, r_2, ..., r_|U|} of U do
13:         remove r_1, r_2, ..., r_|U| from R to get R'
14:         if Γ is not R'-schedulable then
15:             return ⟨true, R'⟩
16:         end if
17:     end for
18:     return ⟨false, ∅⟩
19: end function
```

Algorithm 1 depends on an efficient $\mathcal{R}$-schedulability test (Line 3 in the algorithm). In this paper, we assume that Audsley's algorithm is applicable to the task system (i.e., it can find a schedulable priority assignment if one exists). For such systems, a *revised Audsley's algorithm* can check if Γ is $\mathcal{R}$-schedulable. Similar to Audsley's algorithm, it iteratively tries to find a task that can be assigned with a particular priority level starting from the lowest priority. However, when choosing the candidate task, it shall guarantee that assigning the priority does not violate any partial priority order in $\mathcal{R}$. This is done by checking if the current task is a legal candidate: A task τ_i is a legal candidate if and only if (a) it has not been assigned with a priority; and (b) all the tasks that should have a lower priority than τ_i according to $\mathcal{R}$ have already been assigned with a priority.

Note that a PPO set $\mathcal{R}$ may contain more than one unschedulability cores. One way to compute multiple unschedulability cores from a single unschedulable $\mathcal{R}$ is to perturb the input PPO set $\mathcal{R}$ after every invocation of Algorithm 1, such that successive calls would not return repetitive results. The key observation is that an unschedulability core $\mathcal{U}$ can be computed from $\mathcal{R}$ only if $\mathcal{U} \subseteq \mathcal{R}$. Thus to prevent Algorithm 1 from returning $\mathcal{U}$ (that is computed in the previous invocations), it suffices to modify $\mathcal{R}$ such that $\mathcal{U} \nsubseteq \mathcal{R}$. A simple solution is to select a partial priority order $r \in \mathcal{U}$ and remove it from $\mathcal{R}$ but still keep the resulting $\mathcal{R}$ unschedulable. In this sense, given a $\mathcal{U} \subseteq \mathcal{R}$, there can be $|\mathcal{U}|$ different modifications of $\mathcal{R}$ such that $\mathcal{R} \not\supseteq \mathcal{U}$ (i.e., each by removing a different $r \in \mathcal{U}$ from $\mathcal{R}$). Accordingly, given a set of already computed unschedulability cores $\mathbb{U} = \{\mathcal{U}_1, ...\mathcal{U}_{|\mathbb{U}|}\}$, there can be $\Pi_{i=1}^{|\mathbb{U}|}|\mathcal{U}_i|$ different modifications to ensure that $\mathcal{R} \not\supseteq \mathcal{U}_i, \forall \mathcal{U}_i \in \mathbb{U}$. It may be necessary to examine each of them to find one modification that still maintains $\mathcal{R}$ to be unschedulable. The new unschedulability

core is guaranteed to be different from any $\mathcal{U}_i \in \mathbb{U}$.

Algorithm 2 details a procedure for computing multiple unschedulability cores from an unschedulable PPO set $\mathcal{R}$. It takes as inputs the system Γ, a PPO set $\mathcal{R}$, the desired number of unschedulability cores k to compute, and $\mathbb{U}$ for storing computed unschedulability cores.

Algorithm 2 leverages a subroutine `Perturb`, which explores all possible PPO combinations $\{r_1, r_2, ..., r_{|\mathbb{U}|}\}$ consisting of one PPO r_i from each unschedulability core $\mathcal{U}_i$ in $\mathbb{U}$ (Lines 12–17). For each PPO combination, it removes the elements in the PPO combination from $\mathcal{R}$ (Line 13). If any resulting PPO set is unschedulable, then the subroutine `Perturb` returns true and an $\mathcal{R}'$ that leads to a new unschedulability core (Lines 14–16). Otherwise, it returns false to indicate there is no more unschedulability core.

In case `Perturb` returns true and an $\mathcal{R}'$, Algorithm 2 uses Algorithm 1 to compute a new unschedulability core and adds it to the set $\mathbb{U}$ (Lines 7–8). It iterates until k unschedulability cores are found or there is no more unschedulability core (in which case `Perturb` returns false).

4 Unschedulability Core Guided Optimization Algorithm

In this section, we develop a domain-specific optimization algorithm that leverages the concept of unschedulability core. As discussed earlier, finding all the unschedulability cores can form the complete schedulability region, but this is impractical due to their exponential growth with the system size. However, we observe that the optimization objective may be sensitive to only a small set of unschedulability cores. Hence, we consider the lazy constraint paradigm that only selectively adds unschedulability cores into the problem formulation. The paradigm starts with a relaxed problem that leaves out all the schedulability constraints. That is, the schedulability constraints are temporarily put in a lazy constraint pool. A constraint from the pool is added back only if it is violated by the solution returned for the relaxed problem. In addition, instead of adding the violated schedulability constraints back, we leverage the concept of unschedulability core to provide a much more compact representation of these constraints. In the following, we first details the proposed algorithm and then discusses its benefits compared with existing approaches.

4.1 Optimization Algorithm

The proposed procedure is summarized in Algorithm 3. It takes as inputs the task system Γ and an integer number k which denotes the maximum number of unschedulability cores to compute for each unschedulable solution (see Remark 3 below). The algorithm works as follows.

Step 1 (Line 2). Instantiate the relaxed problem Π as

$$\begin{aligned} & \min C(\mathbf{X}) \\ s.t. \quad & \mathbf{F}(\mathbf{X}) \leq 0 \end{aligned} \tag{8}$$

Different from the original problem in (3), (8) excludes all the system schedulability constraints.

Step 2 (Lines 4–9). Solve problem Π in (8). If Π is infeasible, then the algorithm terminates. Otherwise, let $\mathbf{X}^*$ denote the obtained optimal solution of Π. Construct the corresponding PPO set $\mathcal{R}_{\mathbf{X}^*}$ as follows

$$\mathcal{R}_{\mathbf{X}^*} = \{r_{i,j} | p_{i,j} = 1 \text{ in } \mathbf{X}^*\} \tag{9}$$

Apply the revised Audsley's algorithm to test $\mathcal{R}_{\mathbf{x}^*}$-schedulability. If Γ is not $\mathcal{R}_{\mathbf{X}^*}$-schedulable, go to step 3. Otherwise go to step 4.

Step 3 (Lines 10–11). Apply Algorithm 2 to compute a set of (at most k) unschedulability cores $\mathbb{U}$. Update problem Π by adding the following constraints, then go to step 2.

$$\sum_{r_{i,j} \in \mathcal{U}} p_{i,j} \leq |\mathcal{U}| - 1, \ \ \forall \mathcal{U} \in \mathbb{U} \tag{10}$$

Step 4 (Line 13). Return the optimal priority assignment $\mathcal{P}$ that respects $\mathcal{R}_{\mathbf{X}^*}$ with the revised Audsley's algorithm.

We now study the properties of Algorithm 3. We first prove that it always terminates.

Theorem 5. Algorithm 3 is guaranteed to terminate.

Proof. Let n denote the number of tasks in the system. There can be at most $n \times (n-1)$ partial priority orders. Since an unschedulability core is a subset of all PPOs, the total number of unschedulability cores is bounded by $2^{n \times (n-1)}$.

We now prove by contradiction that each iteration in Algorithm 3 will compute a set of new unschedulability cores, hence the total number of iterations is bounded by $2^{n \times (n-1)}$. At any iteration, let $\mathbb{U}^*$ be the set of known unschedulability cores, and $\mathcal{U}$ be a newly computed unschedulability core returned from Line 10 of Algorithm 3 with $\mathcal{R}_{\mathbf{X}^*}$ as the second input and $\mathbb{U}^*$ as the fourth input. Since $\mathcal{U}$ is computed from $\mathcal{R}_{\mathbf{X}^*}$, it is $\mathcal{R}_{\mathbf{X}^*} \supseteq \mathcal{U}$. Now assume that $\mathcal{U} \in \mathbb{U}^*$, then problem Π contains the constraint (6) induced by $\mathcal{U}$, which $\mathbf{X}^*$ and $\mathcal{R}_{\mathbf{X}^*}$ must satisfy. That is, $\mathcal{R}_{\mathbf{X}^*}$ cannot satisfy all PPOs from $\mathcal{U}$. However, this contradicts with the fact that $\mathcal{R}_{\mathbf{X}^*} \supseteq \mathcal{U}$. □

We now prove the correctness of Algorithm 3.

Theorem 6. Upon termination, Algorithm 3 reports infeasibility if the original problem (3) is infeasible; otherwise it returns a feasible and globally optimal solution.

Algorithm 3 Unschedulability Core Guided Optimization Algorithm

```
1: function FindOptimal(Task set Γ, Integer k)
2:     Build initial problem Π as in (8) // Step 1
3:     while true do
4:         X* = Solve(Π)
5:         if Π is infeasible then
6:             return Infeasibility
7:         end if
8:         Compute R_{X*} as in (9)
9:         if Γ is not R_{X*}-schedulable then
10:            MultiUnschedCore(Γ, R_{X*}, k, 𝕌)
11:            Add Constraint (10) corresponding to 𝕌 to Π
12:        else
13:            return priority assignment P respecting R_{X*}
14:        end if
15:    end while
16: end function
```

Proof. Since each iteration of Algorithm 3 computes only a subset of all unschedulability cores, the algorithm adds only a subset of the constraints in (7) to Π. By Theorem 3, the feasibility region of Π is always maintained to be an over-approximation of that of the original problem. We consider the following two cases on how Algorithm 3 terminates.

Case 1: The algorithm terminates at Line 6. This means that problem Π is infeasible. Since the feasibility region of Π is always no smaller than that of the original problem, the original problem must be infeasible as well.

Case 2: The algorithm exists at Line 13. Line 13 is reached only when the system is $\mathcal{R}_{\mathbf{X}^*}$-schedulable, which by Definition 2 is a stricter condition on schedulability. Thus the returned $\mathcal{P}$ is guaranteed to be feasible. Since an optimal solution in the over-approximated region that is also feasible must be optimal in the exact feasibility region, the returned priority assignment is guaranteed to be optimal. □

Remark 3. The parameter k in Algorithm 3 does not affect the optimality of the algorithm, but influences its runtime. If k is too small, the algorithm may need many iterations to terminate, which incurs solving a large number of problem Π. If k is too large, Algorithm 2 may need to explore numerous PPO combinations. In Section 7, we will use dedicated experiments to study the best choice of k.

We now illustrate Algorithm 3 by applying it to the example Γ_e in Table 1, where the parameter k is set to 1.

Example 3. Consider the following objective function

$$C(\mathbf{X}) = -p_{3,1} - p_{4,1} - p_{4,2} - p_{4,3} - p_{5,4} \tag{11}$$

The algorithm constructs the relaxed problem Π as (8), where $\mathbf{F}(\mathbf{X}) \leq 0$ only contains the set of antisymmetry and transitivity constraints as defined in (2).

The algorithm enters the first iteration and solves Π by possibly using ILP solvers. The solution is (for simplicity, we omit those not affecting the objective function)

$$\mathbf{X}^* = [p_{3,1}, p_{4,1}, p_{4,2}, p_{4,3}, p_{5,4}] = [1, 1, 1, 1, 1]$$

and the PPO set is $\mathcal{R}_{\mathbf{X}^*} = \{r_{3,1}, r_{4,1}, r_{4,2}, r_{4,3}, r_{5,4}\}$. Clearly, Γ_e is not $\mathcal{R}_{\mathbf{X}^*}$-schedulable. Algorithm 3 computes one unschedulability core of $\mathcal{R}_{\mathbf{X}^*}$ as $\mathcal{U}_1 = \{r_{4,3}, r_{5,4}\}$. The corresponding constraint is added to Π which becomes

$$\begin{aligned} & \min C(\mathbf{X}) \\ s.t. \quad & \mathbf{F}(\mathbf{X}) \leq 0 \\ & p_{4,3} + p_{5,4} \leq 1 \end{aligned} \tag{12}$$

In the second iteration, solving (12) gives the solution $\mathbf{X}^* = [p_{3,1}, p_{4,1}, p_{4,2}, p_{4,3}, p_{5,4}] = [1, 1, 1, 1, 0]$. The corresponding PPO set is $\mathcal{R}_{\mathbf{X}^*} = \{r_{3,1}, r_{4,1}, r_{4,2}, r_{4,3}, r_{4,5}\}$. Since Γ is still not $\mathcal{R}_{\mathbf{X}^*}$-schedulable, the algorithm computes another unschedulability core as $\mathcal{U}_2 = \{r_{3,1}\}$. The problem Π is correspondingly updated as

$$\begin{aligned} & \min C(\mathbf{X}) \\ s.t. \quad & \mathbf{F}(\mathbf{X}) \leq 0 \\ & p_{4,3} + p_{5,4} \leq 1, \;\; p_{3,1} \leq 0 \end{aligned} \tag{13}$$

In the third iteration, (13) is solved to obtain the solution $\mathbf{X}^* = [p_{3,1}, p_{4,1}, p_{4,2}, p_{4,3}, p_{5,4}] = [0, 1, 1, 1, 0]$. The corresponding PPO set is $\mathcal{R}_{\mathbf{X}^*} = \{r_{1,3}, r_{4,1}, r_{4,2}, r_{4,3}, r_{4,5}\}$. At this point, Γ_e becomes $\mathcal{R}_{\mathbf{X}^*}$-schedulable. The algorithm then terminates and returns the following optimal solution

$$\mathcal{P} = [\pi_4 > \pi_1 > \pi_2 > \pi_3 > \pi_5 > \pi_6]$$

4.2 Advantages

We now discuss the possible limitations of standard optimization frameworks such as BnB and ILP, before highlighting the advantages of our approach. *First*, a straightforward formulation of the schedulability region in these standard frameworks may force to check the schedulability of a large number of solutions, since the schedulability condition is often sophisticated and makes it difficult to find similarities among different solutions. *Second*, the complexity of the schedulability analysis may even prevent us from leveraging existing optimization frameworks. Consider the problem in Section 7.1, i.e., to optimize the mixed-criticality Simulink models under Adaptive Mixed Criticality (AMC) scheduling [22]. The most accurate schedulability analysis for AMC, AMC-max [22], hinders a possible formulation in ILP: It requires to check, for each possible time instant c of criticality change, whether the corresponding response time is within the deadline. However, the range of c is unknown a priori as it depends on the task response time in LO mode.

Comparably, Algorithm 3 comes with three advantages. First, it avoids modeling the complete schedulability region. Instead, it explores, in an objective-guided manner, much simpler over-approximations that are sufficient to establish an optimal solution. Second, it hides the complicated schedulability conditions by converting them into simple constraints induced from unschedulability cores, where system schedulability is checked using a separate and dedicated procedure (Line 3, Algorithm 1). This also allows to easily accommodate any form of schedulability analysis that may be difficult to formulate in frameworks such as ILP. Third, the conversion to unschedulability core is essentially a generalization from one infeasible solution to many, which is a key in the algorithm efficiency.

5 Generalization

In this section, we generalize to problems that may involve decision variables other than priority assignment. We first provide an alternative interpretation of the original optimization problem and generalize the concept of unschedulability core. We then discuss the applicability and efficiency of the generalized framework.

5.1 Generalized Concept of Unschedulability Core

Consider the illustrative problem in Example 3. The objective function depends on the satisfaction of a set of partial priority orders $\{r_{3,1}, r_{4,1}, r_{4,2}, r_{4,3}, r_{5,4}\}$, each of which represents an additional constraint on scheduling. Unlike the hard constraint of the problem, e.g., system schedulability, the constraint imposed by $r_{i,j}$ only need to be optionally satisfied. In other words, $r_{i,j}$ is regarded as a *soft* scheduling constraint. Its satisfaction comes with a reward of an improved objective, but also a possible penalty on the schedulability since now τ_i has to be assigned with a higher priority than τ_j. The optimization problem in Example 3 is

to optimally satisfy a subset of those scheduling constraints $\{r_{3,1}, r_{4,1}, r_{4,2}, r_{4,3}, r_{5,4}\}$ subject to system schedulability.

Besides priority assignment, there are many design choices that may affect system schedulability and can be considered as a scheduling constraint. Examples include

- Functional block to task mapping, where all blocks mapped to the same task share the same priority;
- The use of semaphore locks to protect shared resource, where a task suffers a blocking time equal to the largest WCET of the critical section from lower priority tasks;
- As an alternative to semaphore locks, it is also sufficient to prove that the tasks sharing the same resources do not preempt each other. However, this imposes a tighter execution window as a task must finish before the next activation of higher priority tasks that share the same resource.

We now introduce the general form of optimization problems that can be handled by the proposed framework.

Definition 4. Let $\mathcal{L} = \{\zeta_1, ...\zeta_m\}$ be a set of scheduling constraints that only need to be optionally satisfied. The satisfaction of each constraint ζ_i is associated with a cost. A scheduling constraint optimization problem is to optimally satisfy the given scheduling that the total cost is minimized. Formally, the problem is expressed as follows

$$\begin{aligned} &\min\ C(\mathbf{b}) \\ s.t.\ &\text{system schedulability} \\ &b_k = 1 \implies \zeta_k \text{ is enforced},\ \forall \zeta_k \in \mathcal{L} \end{aligned} \tag{14}$$

where $\mathbf{b} = \{b_1, ...b_m\}$ is the set of binary variables that define b_k for each ζ_k as follows

$$b_k = \begin{cases} 1 & \zeta_k \text{ is enforced,} \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

The problem considered in the previous three sections can be understood as a special instance of (14) where ζ_k is a partial priority order. Still, for this general form of optimization problem, the challenge mainly lies in the difficulty of efficiently formulating the feasibility region. To address this challenge, we extend the idea of schedulability region abstraction using unschedulability core. We first establish similar concepts and properties as those in Section 3.

Definition 5. A *scheduling constraint set* $\mathcal{R}$ is a set of scheduling constraints in $\mathcal{L}$, i.e., $\mathcal{R} \subseteq \mathcal{L}$. $\mathcal{R}$ is said to be *schedulable* if and only if the following problem is feasible.

$$\begin{aligned} &\min\ 0 \\ s.t.\ &\text{system schedulability} \\ &\zeta_k \text{ is enforced},\ \forall \zeta_k \in \mathcal{R} \end{aligned} \tag{16}$$

Comparing to problem (14), problem (16) removes the objective and treats all scheduling constraints ζ_k in $\mathcal{R}$ as hard constraints. Informally, $\mathcal{R}$ is schedulable if and only if there exists a feasible solution such that all scheduling constraints in $\mathcal{R}$ are satisfied and all tasks are schedulable.

Theorem 7. Given two scheduling constraint sets $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\mathcal{R}_1 \supseteq \mathcal{R}_2$, the following properties hold.

$$\begin{aligned} &\mathcal{R}_1 \text{ is schedulable} \implies \mathcal{R}_2 \text{ is schedulable} \\ &\mathcal{R}_2 \text{ is not schedulable} \implies \mathcal{R}_1 \text{ is not schedulable} \end{aligned} \tag{17}$$

Proof. Each element in a scheduling constraint set $\mathcal{R}$ imposes an additional constraint on problem (16). Since $\mathcal{R}_1 \supseteq \mathcal{R}_2$, $\mathcal{R}_1$ is stricter than $\mathcal{R}_2$. Thus if $\mathcal{R}_1$ is schedulable, $\mathcal{R}_2$ must also be schedulable. □

Definition 6. A scheduling constraint set $\mathcal{U}$ is an *unschedulability core* if and only if the following two conditions hold.

- $\mathcal{U}$ is not schedulable.
- For all $\mathcal{R} \subset \mathcal{U}$, $\mathcal{R}$ is schedulable.

Intuitively, an unschedulability core refers to a minimal subset of $\mathcal{L}$ that can not be simultaneously satisfied. It implies the following constraints that must always be met.

$$\sum_{\forall \zeta_k \in \mathcal{U}} b_k \leq |\mathcal{U}| - 1 \tag{18}$$

We refer to (18) as the implied constraint by unschedulability core $\mathcal{U}$. Similar to the framework for optimizing partial priority orders, our overall idea is to use (18) as an alternative form for modeling the feasibility region of (14). The algorithms (Algorithms 1–3) for calculating unschedulability cores and the optimization procedure are applicable to the new concept of unschedulability core. This comes from the general property of the scheduling constraints as stated in Theorem 7: the system schedulability is monotonic with respect to the set of scheduling constraints.

5.2 Applicability and Algorithm Efficiency

In the following, we discuss the factors that affect the algorithm efficiency of the proposed framework and highlight where it is beneficial (i.e., much faster than existing approaches such as ILP). We note in each iteration Algorithm 3 mainly performs two operations: the computation of unschedulability cores (Line 10 in the algorithm) and solving the relaxed problem Π (Line 4).

By Algorithm 1, the computation of unschedulability core can be further decomposed into a series of schedulability test on a given scheduling constraint set $\mathcal{R}$, i.e., testing the feasibility of (16). Obviously, such a subroutine depends on the actual form of the scheduling constraint ζ_k and the task model. For example, the problem considered in Sections 2–4 defines ζ_k as a partial priority order. In this case, for many task models and scheduling schemes as summarized in [4], schedulability of $\mathcal{R}$ can be tested using a revised Audsley's algorithm. It only needs to check $O(n^2)$ priority assignments out of the $n!$ possible ones.

However, for some other forms of scheduling constraints or task models, the test for the schedulability of $\mathcal{R}$ may be much more difficult. For example, for systems with preemption threshold scheduling [11], Audsley's algorithm is inapplicable, and the exact test of schedulability of $\mathcal{R}$ may require to check an exponential number of priority assignments. In general, if checking the schedulability $\mathcal{R}$ needs to explore a large number of scheduling constraint sets, the proposed framework may not be advantageous compared to other approaches (such as ILP). In the next section, we use two examples to illustrate how certain scheduling constraints can be handled.

An important consideration in the development process is how the priority assignments are stable with respect to small changes, which may arise unforeseeably. In cases

where (revised) Audsley's algorithm is still optimal, for example, for robustness to additional interference [23], our framework can be applied.

To compute unschedulability cores, it also relies on the schedulability analysis, i.e., to analyze if the system is schedulable for a given valuation on the decision variables. Although our framework is applicable for any schedulability analysis technique, its efficiency depends on that of the schedulability analysis. In practice, the schedulability analysis is typically efficient enough (as demonstrated in Section 7.1 for AMC-scheduled systems). However, it can still be a major bottleneck. For example, the analysis of digraph real-time tasks [24] (and similarly systems modeled with finite state machines [25]) requires to enumerate all paths in the digraph of a higher priority task to identify its worst-case interference. In this scenario, our framework has the flexibility, and it is usually beneficial, to optimize the implementation of schedulability analysis. For example, within one invocation of Algorithm 2, the resulting unschedulability cores across consecutive iterations are mostly similar, suggesting that some previous results of schedulability analysis can likely be reused. In addition, during the computation of unschedulability cores, typically most invocations of schedulability analysis will return unschedulability. This suggests that the use of fast, but mostly accurate necessary only analysis may be helpful, as it can quickly detect obviously unschedulable solutions. We apply the proposed framework in Algorithm 3 to optimizing the implementation of synchronous finite state machines [26]. Our experimental results show that with a direct use of the analysis technique in [25], over 95% of the total runtime is spent on schedulability analysis. However, this bottleneck is avoided and the overall runtime is reduced by 3 orders of magnitude, by combining a schedulability memoization technique (which exploits the reuse of previous schedulability analysis results), and a relaxation-recovery strategy (which leverages a simple necessary only analysis for ruling out obviously infeasible solutions).

We now discuss the difficulty of solving the relaxed problem II. Since II does not contain the schedulability constraints, it opens the possibility to use appropriate mathematical programming framework while adopting the most accurate, but possibly sophisticated schedulability analysis. For example, if the objective and constraints in (14) are all linear (except the schedulability constraints), then we can leverage integer linear programming solvers such as CPLEX to solve II. Our framework allows to combine the power of these modern solvers (which adopt numerous highly sophisticated techniques for generic branch and cut strategy) and domain-specific algorithms (such as Audsley's algorithm for finding a schedulable priority assignment).

6 Examples of Application

In this section, we provide two examples of optimization problems that fit the proposed optimization framework, both proven to be NP-hard [17], [27]. The first is optimizing semantics-preserving implementation of Simulink models, optionally with memory constraint. The second is minimizing memory consumption for shared resource protection, in the context of the automotive AUTOSAR standard. In Section 7, we apply our framework to these two example systems and compare with standard techniques (BnB, ILP).

6.1 Optimizing Implementation of Simulink Models

A Simulink model is a Directed Acyclic Graph (DAG) where nodes represent functional blocks and links represent data communication between the blocks [17].

For simplicity and demonstration purpose only, we assume that each functional block is implemented in a dedicated task (hence *use the terms functional block and task interchangeably*). However, it should be noted that in practice, a design may contain hundreds or thousands of blocks and thus a more common strategy is to allocate multiple blocks to a single task where blocks inside the task are statically scheduled. Mapping of blocks to tasks have been studied in various works (e.g., [9]). The focus of this paper is instead on priority assignment. It is possible to simultaneously consider both function-to-task mapping and priority assignment, and we leave it as future work.

The semantics-preserving implementation of a Simulink model has to match its functional behavior. This typically requires the addition of a Rate Transition (RT) block between a reader and a writer with different but harmonic periods, which is a special type of wait-free communication buffers. However, the costs of RT blocks are additional memory overheads and in some cases, functional delays in result delivery. The latter degrades control performance.

Consider a fast reader τ_r and slow writer τ_w that writes to τ_r. Assigning higher priority to τ_r generally helps schedulability as it conforms with the rate monotonic policy. However, since the reader now executes before the writer, an RT block is needed to store the data from the previous instance of the writer, which also incurs a functional delay. On the other hand, if τ_r can be assigned with a lower priority while keeping the system schedulable, then no RT block is needed and no functional delay is introduced.

The software synthesis of Simulink model is to exploit *priority assignment as the design variable to minimize the weighted sum of functional delays introduced by the RT blocks* (hence improving control quality). We note that Audsley's algorithm is no longer optimal as system schedulability is not the only constraint. Formally, the problem can be formulated as follows.

$$\begin{aligned} &\min \sum_{\forall(w,r)} \beta_{w,r} \cdot p_{r,w} \\ &s.t. \text{ system schedulability} \\ &\text{constraints in (2)} \end{aligned} \tag{19}$$

where (w, r) represents a pair of communicating tasks, and the parameter $\beta_{w,r}$ is the penalty on control performance if τ_r is assigned with a higher priority than τ_w.

Optionally, the implementation of Simulink models may be subject to memory constraints. An RT block is essentially a wait-free buffer between the reader and the writer, and thus comes with memory cost. A unit delay RT block, necessary whenever the reader task τ_r has a higher priority than the writer τ_w, is twice the size of the protected shared variable. For a higher priority writer and a lower priority reader, the RT block has a memory cost of the same size as the shared variable. However, it can be avoided if we can

ensure the absence of preemption, i.e., the lower priority reader finishes before the next activation of the writer to ensure the absence of preemption by the writer. In Simulink, the reader and writer tasks always have harmonic periods (one period is an integer multiple of the other) with synchronized release offsets. Hence, it suffices to ensure that the worst-case response time of the reader R_r satisfies $R_r \leq o_{r,w} = \min\{T_r, T_w\}$, where $o_{r,w}$ is the smallest offset from an activation of τ_r to the next activation of τ_w. To summarize, an RT block can be avoided if the following scheduling constraint is satisfied

$$(p_{w,r} = 1) \wedge R_r \leq \min\{T_r, T_w\} \tag{20}$$

Thus, for each writer and reader pair (τ_w, τ_r), we introduce the additional scheduling constraint

$$\zeta_{w,r} \equiv \text{constraint by (20)} \tag{21}$$

The associated binary variable $b_{\zeta_{w,r}}$ is defined as 1 if $\zeta_{w,r}$ is enforced and 0 otherwise. To ensure the memory budget, we add the following constraint to the problem (19)

$$\sum_{\forall(\tau_w,\tau_r)} \left(m_{w,r}(p_{w,r} - b_{\zeta_{w,r}}) + 2m_{w,r} \cdot p_{r,w}\right) \leq M \tag{22}$$

where $m_{w,r}$ is the size of the memory buffer shared between τ_w and τ_r, and M is the available memory for RT blocks.

We now discuss the schedulability test for $\mathcal{R}$ with the newly defined scheduling constraint (21). In addition to a partial priority order, it also specifies an upper-bound on the worst-case response time of the reader tasks τ_r, or equivalently a virtual deadline. Thus, a given scheduling constraint set $\mathcal{R}$ essentially specifies a set of partial priority orders as well as virtual deadlines for certain reader tasks. The revised Audsley's algorithm discussed in Section 3 can still be applied to test the schedulability of $\mathcal{R}$: it tries to find a feasible priority assignment respecting $\mathcal{R}$ where the deadline of any task τ_r for $\zeta_{w,r} \in \mathcal{R}$ is set to $\min\{T_r, T_w\}$. Thus the problem can be efficiently solved by the general framework in Algorithm 3.

6.2 Minimizing Memory of AUTOSAR models

The second example is to minimize the memory usage of AUTOSAR components [28], where a set of runnables (the AUTOSAR term for functional blocks) communicates through shared buffers that shall be appropriately protected to ensure data integrity. We assume that each runnable is implemented in a dedicated task, and use the terms runnable and task interchangeably. We consider the problem for the optimal selection of (a) the priority assignment to tasks; (b) the selection of the appropriate mechanism for protecting shared buffers among a set of possible choices, including ensuring absence of preemption, lock-based method (priority ceiling semaphore lock), and wait-free method [28]. These mechanisms are associated with different scheduling constraints and memory costs.

1) *Ensuring absence of preemption.* It has no memory or timing cost, but requires that the two communicating tasks satisfy that the lower priority task always finish before the next activation of the higher priority task. Specifically, the minimum distance between activations of two communicating tasks τ_i and τ_j is given by the greatest common divisor of their period, i.e., $gcd(T_i, T_j)$. To ensure absence of preemption, it suffices to guarantee that

$$\begin{cases} R_i \leq gcd(T_i, T_j) \\ R_j \leq gcd(T_i, T_j) \end{cases} \tag{23}$$

2) *Wait-free method* imposes no extra timing constraints but incurs a memory cost equal to the size of the shared buffer.

3) *Lock-based method* introduces blocking delay to higher priority tasks but reduces the memory overhead to minimal (only one-bit for implementing semaphore locks). Let s denote the shared variable between τ_i and τ_j. The timing constraints are formally expressed as follows

$$\begin{cases} B_i \geq C_j^s, \text{ if } \tau_j \text{ has lower priority than } \tau_i \\ B_j \geq C_i^s, \text{ if } \tau_i \text{ has lower priority than } \tau_j \end{cases} \tag{24}$$

where B_i (B_j) represents the blocking time of τ_i (τ_j), and C_i^s (C_j^s) represents the worst-case execution time of the critical section for τ_i (τ_j) to access s.

The objective is to minimize the total memory usage while ensuring system schedulability. In the following, we show how the problem can be solved by the general framework in Algorithm 3.

Specifically, for each pair of communicating tasks (τ_i, τ_j), we define the following scheduling constraints for each of the three mechanisms to protect the shared variable

$$\begin{array}{ll} \text{absence of preemption:} & \zeta_{i,j}^a \equiv \text{constraint by (23)} \\ \text{wait-free method:} & \zeta_{i,j}^w \equiv \text{None} \\ \text{lock-based method:} & \zeta_{i,j}^l \equiv \text{constraint by (24)} \end{array} \tag{25}$$

Note that the wait-free method does not impose an additional scheduling constraint but comes with an additional memory cost, hence the scheduling constraint associated with $\zeta_{i,j}^w$ is empty. Also, we define the binary variables $b_{\zeta_{i,j}^a}$, $b_{\zeta_{i,j}^w}$ and $b_{\zeta_{i,j}^l}$ to denote the use of each of the mechanisms.

$$\begin{aligned} b_{\zeta_{i,j}^a} = 1 &\implies \zeta_{i,j}^a \text{ is enforced} \\ b_{\zeta_{i,j}^w} = 1 &\implies \zeta_{i,j}^w \text{ is enforced} \\ b_{\zeta_{i,j}^l} = 1 &\implies \zeta_{i,j}^l \text{ is enforced} \end{aligned} \tag{26}$$

It is sufficient to protect each communication pair with one of the mechanisms

$$b_{\zeta_{i,j}^a} + b_{\zeta_{i,j}^w} + b_{\zeta_{i,j}^l} \geq 1, \ \forall(\tau_i, \tau_j) \tag{27}$$

The optimization objective can be written as

$$C(\mathbf{X}) = \sum_{\forall(\tau_i,\tau_j)} \beta_{i,j}^w b_{\zeta_{i,j}^w} + \beta_{i,j}^l b_{\zeta_{i,j}^l} \tag{28}$$

where $\beta_{i,j}^w$ and $\beta_{i,j}^l$ are the memory cost for wait-free method and lock-based method, respectively. The optimization problem is to minimize the memory cost in (28), subject to the constraints in (26), (27), (2), and system schedulability.

We now examine whether there is an efficient schedulability test for $\mathcal{R}$ over the scheduling constraints. Constraint (23) is equivalent to setting a virtual deadline for τ_i and τ_j. Constraint (24) specifies how blocking time should be computed. Thus a given scheduling constraint set $\mathcal{R}$ essentially specifies a PPO set as well as a setting of virtual deadlines

and blocking times for associated tasks. Finding a schedulable priority assignment under the specified setting can still be performed using the revised Audsley's algorithm. This allows to keep the algorithm efficiency of the general framework in Algorithm 3.

7 Experimental Evaluation

In this section, we present results of our experimental evaluation for the proposed technique. We consider the two example problems discussed in the previous section.

7.1 Optimizing Implementation of Mixed-Criticality Simulink Models

To demonstrate that our approach can accommodate any schedulability analysis, we consider the problem of software synthesis for Simulink model as discussed in Section 6.1, but the model contains functional blocks with different criticality levels scheduled with the Adaptive Mixed Criticality (AMC) scheme [22]. This problem is NP-hard as the special case where all tasks are LO-critical is proven to be NP-hard [17]. The schedulability of AMC scheduled systems can be analyzed with two methods [22]: AMC-max and AMC-rtb. We compare the proposed technique and a direct ILP formulation. The straightforward ILP formulation of AMC-max is excluded due to its extreme high complexity (see Section 4). We also include brute-force BnB algorithms, to evaluate the benefit from modern ILP solvers (e.g., CPLEX). The list of compared methods includes:

- **UC-AMC-max**: Unschedulability core guided algorithm (Algorithm 3) with AMC-max as schedulability analysis;
- **UC-AMC-rtb**: Algorithm 3 with AMC-rtb analysis;
- **ILP-AMC-rtb**: ILP with AMC-rtb analysis, solved by CPLEX;
- **BnB-AMC-max**: BnB with AMC-max analysis;
- **BnB-AMC-rtb**: BnB with AMC-rtb analysis.

We use TGFF [29] to generate random systems. Each functional block has at most an in-degree of 3 and an out-degree of 2. We first randomly choose a number of sink functional blocks and assign it with HI-criticality. The criticality of the remaining blocks are determined by the following rules [30]:

- If a block is the predecessor of any HI-critical block, then it is assigned an HI-critical level as well;
- All blocks not assigned HI-critical by the above rule are assigned LO-critical level.

We first study the scalability with respect to the number of functional blocks which varies from 5 to 100. The system utilization in LO-criticality mode is randomly selected from [0.5, 0.95]. For each task in the system, its utilization is generated using the UUnifast-Discard algorithm [31]. Task period is randomly chosen from a predefined set of values {10,20,40,50,100,200,400,500,1000}. The criticality factor of HI-criticality task is uniformly set to 2.0 ($\frac{C_i(HI)}{C_i(LO)} = 2.0$). We generate 1000 systems and report their average for each point in the plots. We first set k to 5 in Algorithm 3 as the corresponding runtime is typically within 10% of the optimal setting. For all random systems (including those in Section 7.2), we set a timeout of 900s for each problem instance for all algorithms to avoid excessive waiting.

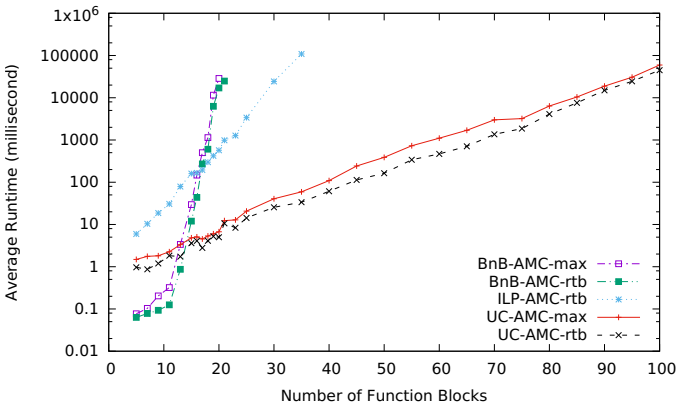


Fig. 1: Runtime vs. System Size for Implementation of Simulink Model

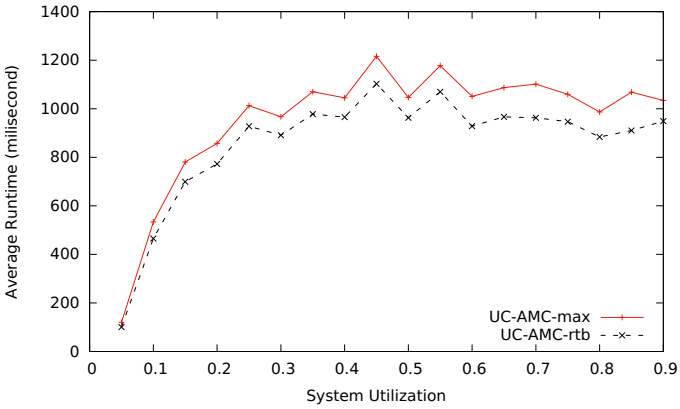


Fig. 2: Runtime vs. Utilization for Implementation of Simulink Model

Figure 1 illustrates the runtime of these methods. **AMC-max** based methods give slightly better optimal solutions (no more than 5%) due to the better accuracy of **AMC-max** than **AMC-rtb**, but they also run slower than their counterpart based on **AMC-rtb** (e.g., **UC-AMC-max** vs. **UC-AMC-rtb**). The superiority of branch-and-bound based algorithms in small-sized systems is mainly due to the overhead in ILP model construction in other methods, which consumes a significant portion of the runtime when the ILP problem is rather simple. The scalability for **UC-AMC-rtb** and **UC-AMC-max** is remarkably better than that of the other methods. For example, for systems with 35 tasks, the unschedulability core guided techniques are more than 1000 times faster compared to **ILP-AMC-rtb**. In addition, **UC-AMC-rtb** and **UC-AMC-max** are quite close in their runtimes, demonstrating that Algorithm 3 is not very sensitive to the complexity of the schedulability analysis. Finally, **ILP-AMC-rtb** scales much better than **BnB-AMC-rtb**. This demonstrates that modern ILP solvers, which are equipped with various sophisticated techniques, are generally more efficient than brute force BnB.

We also evaluate the scalability of **UC-AMC-max** and **UC-AMC-rtb** with respect to different system utilization

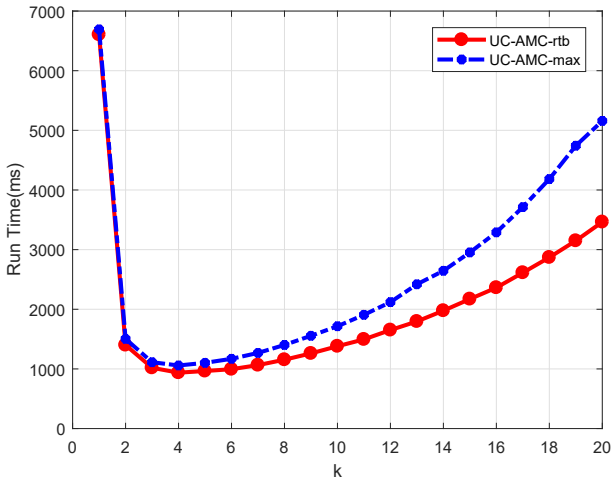


Fig. 3: Runtime vs. Parameter k for Implementation of Simulink Model

TABLE 2: Results on fuel injection case study

Method	Objective	Runtime	Status
UC-AMC-max	23	1.32s	Terminate
UC-AMC-rtb	23	0.19s	Terminate
ILP-AMC-rtb	23	17.3h	Terminate
BnB-AMC-max	23	$\geq 24h$	Timeout
BnB-AMC-rtb	23	$\geq 24h$	Timeout

ranging from 0.05 to 0.90. The number of functional blocks in a system is fixed to 70 while the other parameters remain the same. The result is shown in Figure 2. The optimization problem is relatively easy at very low utilization levels, as the system is easily schedulable for most of the priority assignments. As utilization continues to increase, the runtime grows but eventually remains at a similar value. The result illustrates that the scalability of unschedulability core guided algorithms is not sensitive to system utilization.

We next study the effect of parameter k, the number of unschedulability cores to compute for each infeasible solution, on the algorithm efficiency. The number of functional blocks and system utilization are fixed to 70 and 70% respectively. The other parameters and system generation scheme remain the same. Figure 3 shows the average runtime of the algorithms **UC-AMC-max** and **UC-AMC-rtb** w.r.t. different values of k. It can be seen that the algorithms run the fastest when $k = 5$. In cases with other system sizes and utilizations, $k = 5$ still remains to be a good choice.

Finally, we apply the proposed technique to an industrial fuel injection controller case study [17]. The system contains 90 functional blocks and 106 communication links. The total utilization in LO mode is 94.1%. We assign task criticality in the same way as the randomly generated synthetic systems, which results in 42 HI-critical tasks. The criticality factor is set to 2.0. We compare the same methods, and we set a time limit of 24 hours. The results are summarized in Table 2. As in the table, the proposed **UC-AMC-max** and **UC-AMC-rtb** solve the optimization problem in about a second, which is 4 orders of magnitude faster than the other approaches.

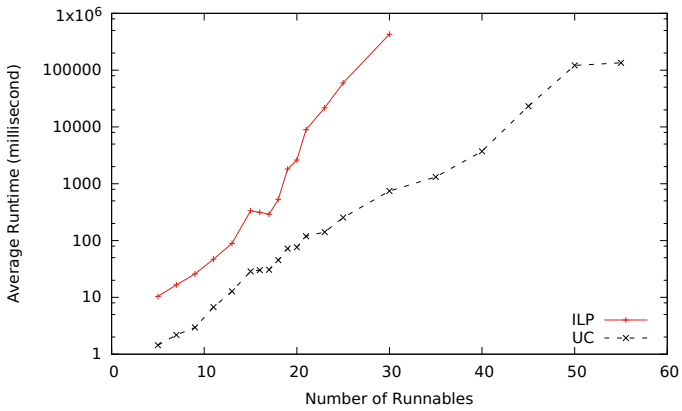


Fig. 4: Runtime vs. System Size for Memory Minimization of AUTOSAR

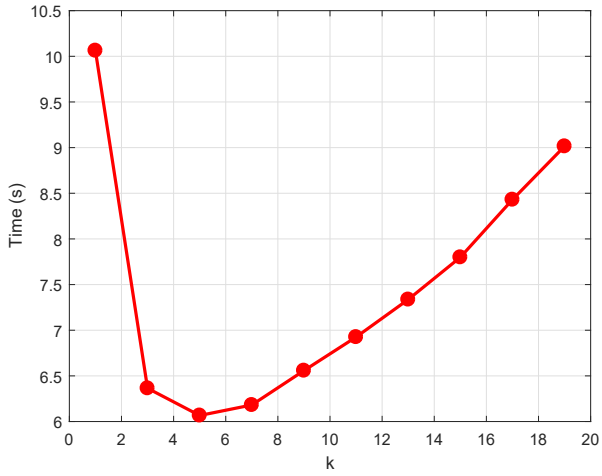


Fig. 5: Runtime vs. Parameter k for Memory Minimization of AUTOSAR

7.2 Minimizing Memory of AUTOSAR Components

In this experiment, we consider the problem discussed in Section 6.2, where the tasks are assumed to be periodic. We compare our technique (denoted as **UC**) with the request bound function based ILP formulation [14] (denoted as **ILP**) on randomly generated synthetic task systems. We omit BnB as it is demonstrated to be less scalable than ILP. Task utilization and period are generated in the same way as Section 7.1. Each task communicates with 0 to 5 other tasks. The size of the shared buffer is randomly selected between 1 to 512 bytes. The WCET of the critical section for each task τ_i on each shared buffer is randomly generated from $(0, 0.1 \cdot C_i]$.

Figure 4 plots the runtime versus system sizes for the synthetic systems. As in the figure, **UC** always takes significantly smaller amount of time than **ILP** while giving the same optimal results, and the difference becomes larger with larger systems. For example, for systems with 25 runnables, **UC** runs about 200 times faster than **ILP**. This demonstrates that the carefully crafted algorithm **UC** can achieve much better scalability than the other exact algorithms while maintaining optimality.

Finally, we study the effect of k on the algorithm runtime. Figure 5 shows the results for systems with 35 tasks. Similar to Figure 3, k being too large or too small may negatively affect the algorithm efficiency, and $k = 5$ is typically suitable for most problem settings.

8 Conclusions

In this work, we introduce the concept of unschedulability core, a compact representation of schedulability conditions for use in design optimization of real-time systems with fixed priority scheduling. We develop efficient algorithms for calculating unschedulability cores and present an unschedulability core guided optimization framework. Experiments show that our framework can provide optimal solutions while scaling much better than standard optimization approaches such as BnB and ILP.

References

[1] S. Chakraborty, "Keynote talk: Challenges in automotive cyber-physical systems design," in *International Conference on VLSI Design*, 2012.

[2] K. Bazaka and M. V. Jacob, "Implantable devices: issues and challenges," *Electronics*, vol. 2, no. 1, pp. 1–34, 2012.

[3] N. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39 – 44, 2001.

[4] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems," *J. Syst. Archit.*, vol. 65, no. C, pp. 64–82, Apr. 2016.

[5] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: An np-hard problem made easy," *Real-Time Syst.*, vol. 4, no. 2, pp. 145–165, May 1992.

[6] I. Bate and P. Emberson, "Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[7] A. Hamann, M. Jersak, K. Richter, and R. Ernst, "Design space exploration and system optimization with symta/s - symbolic timing analysis for systems," in *IEEE Real-Time Systems Symposium*, 2004.

[8] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *IEEE Real-Time Systems Symposium*, 2000.

[9] H. Zeng, M. Di Natale, and Q. Zhu, "Minimizing stack and communication memory usage in real-time embedded applications," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 5s, pp. 1–25, Jul. 2014.

[10] C. Wang, Z. Gu, and H. Zeng, "Global fixed priority scheduling with preemption threshold: Schedulability analysis and stack size minimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3242–3255, 2016.

[11] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *International Conference on Real-Time Computing Systems and Applications*, 1999.

[12] Q. Zhu, H. Zeng, W. Zheng, M. D. Natale, and A. L. Sangiovanni-Vincentelli, "Optimization of task allocation and priority assignment in hard real-time distributed systems," *ACM Trans. Embedded Comput. Syst.*, vol. 11, no. 4, pp. 85:1–85:30, 2012.

[13] M. Panic, S. Kehr, E. Quiones, B. Boddecker, J. Abella, and F. J. Cazorla, "Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2014.

[14] H. Zeng and M. Di Natale, "An efficient formulation of the real-time feasibility region for design optimization," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 644–661, April 2013.

[15] H. Zeng and M. D. Natale, "Efficient implementation of autosar components with minimal memory usage," in *IEEE International Symposium on Industrial Embedded Systems*, 2012.

[16] P. Deng, Q. Zhu, F. Cremona, M. Di Natale, and H. Zeng, "A model-based synthesis flow for automotive cps," in *ACM/IEEE International Conference on Cyber-Physical Systemsn*, April 2015.

[17] M. D. Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli, "Synthesis of multi-task implementations of simulink models with minimum delays," *IEEE Trans. Industrial Informatics*, vol. 6, no. 4, pp. 637–651, 2010.

[18] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Syst.*, vol. 51, no. 1, pp. 77–123, Jan. 2015.

[19] R. Davis and M. Bertogna, "Optimal fixed priority scheduling with deferred pre-emption," in *IEEE Real-Time Systems Symposium*, 2012.

[20] M. Stigge and W. Yi, "Combinatorial abstraction refinement for feasibility analysis of static priorities," *Real-Time Systems*, vol. 51, no. 6, pp. 639–674, 2015.

[21] A. N. Letchford and A. Lodi, "Strengthening chvátal-gomory cuts and gomory fractional cuts," *Oper. Res. Lett.*, vol. 30, no. 2, pp. 74–82, Apr. 2002.

[22] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *IEEE Real-Time Systems Symposium*, 2011.

[23] R. I. Davis and A. Burns, "Robust priority assignment for fixed priority real-time systems," in *28th IEEE International Real-Time Systems Symposium*, Dec 2007, pp. 3–14.

[24] M. Stigge and W. Yi, "Combinatorial abstraction refinement for feasibility analysis of static priorities," *Real-time systems*, vol. 51, no. 6, pp. 639–674, 2015.

[25] H. Zeng and M. D. Natale, "Schedulability analysis of periodic tasks implementing synchronous finite state machines," in *Euromicro Conference on Real-Time Systems*, 2012.

[26] Y. Zhao, C. Peng, H. Zeng, and Z. Gu, "Optimization of real-time software implementing multi-rate synchronous finite state machines," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 175:1–175:21, Sep. 2017.

[27] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, "An optimization approach for the synthesis of autosar architectures," in *IEEE 18th Conference on Emerging Technologies Factory Automation*, Sept 2013.

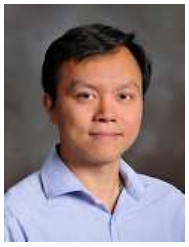
[28] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai, "Time and memory tradeoffs in the implementation of autosar components," in *Conference on Design, Automation and Test in Europe*, 2009.

[29] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *6th international workshop on Hardware/software codesign*, 1998.

[30] S. Baruah, "Implementing mixed-criticality synchronous reactive programs upon uniprocessor platforms," *Real-Time Syst.*, vol. 50, no. 3, pp. 317–341, 2014.

[31] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *IEEE Real-Time Systems Symposium*, 2009.

Yecheng Zhao Yecheng Zhao is currently pursuing the PhD degree in Computer Engineering at Virginia Tech. He received his B.E. in Electrical Engineering from Harbin Institute of Technology, Harbin, China. His main research interest is design optimization for real-time embedded systems.



Haibo Zeng Haibo Zeng is currently a faculty member at Virginia Tech. He received his Ph.D. in Electrical Engineering and Computer Sciences from University of California at Berkeley, and B.E. and M.E. in Electrical Engineering from Tsinghua University, Beijing, China. He was a senior researcher at General Motors R&D until October 2011, and an assistant professor at McGill University, Canada until August 2014. His research interests are embedded systems, cyber-physical systems, and real-time systems, with four best paper/best student paper awards in the above fields.