Optimal Implementation of Simulink Models on Multicore Architectures with Partitioned Fixed Priority Scheduling

Shamit Bansal[†*], Yecheng Zhao[†*], Haibo Zeng[†], and Kehua Yang[‡]
[†]Virginia Tech, USA. Email: {shamitb,zyecheng,hbzeng}@vt.edu
[‡]Hunan University, China. Email: khyang@hnu.edu.cn
[*]The first two authors contributed equally to this work.

***Abstract*—Model-based design using the Simulink modeling formalism and associated toolchain has gained popularity in the development of real-time embedded systems. However, the current research on software synthesis for Simulink models has a critical gap for providing a deterministic, semantics-preserving implementation on multicore architectures with partitioned fixed-priority scheduling. In this paper, we consider a semantics-preservation mechanism that combines (1) the RT blocks from Simulink, and (2) task offset assignment to separate the time windows to access shared buffers by communicating tasks. We study the software synthesis problem that optimizes control performance by judiciously assigning task offsets, task priorities, and task communication mechanisms. We develop a problem-specific exact algorithm that uses an abstraction layer to hide the complexity of timing analysis. Experimental results show that it may run a few orders of magnitude faster than a direct formulation in integer linear programming.**

I. Introduction

In the development of control-centric real-time embedded systems, the use of the Simulink formalism and associated toolchain is becoming widespread, largely because of the possibility to validate/verify the correctness of the Simulink model. To reduce implementation errors and shorten turnaround times, code generators such as Simulink Coder are provided to automatically generate software implementations on single-core architectures. Simulink Coder adds Rate Transition (RT) blocks between communicating functional blocks with different rates [25], to ensure semantics-preserving software implementations (i.e., those matching the logical-time semantics in the model).

With the single-core processors reaching their limit, modern embedded systems are now moving towards multicore architectures for higher efficiency and performance. In this paper, we consider the problem of optimizing the software implementation of Simulink models, under partitioned preemptive fixed priority scheduling on multicore platforms. Such a scheduling policy is adopted by industrial standards like AUTOSAR, by commercial real-time operating systems (e.g., VxWorks, LynxOS, and ThreadX), and in particular, by Simulink Coder [25].

The current solutions for semantics-preserving implementation for Simulink models, including those provided by the commercial code generators, do not scale to multicore architectures. For example, the Simulink toolchain from MathWorks relies on users to specify the data communication mechanisms, and the generated code may have non-deterministic behavior and cannot guarantee to be semantics-preserving [25]. Briefly speaking, RT blocks alone only work for communicating functional blocks on the same core, since they leverage priority orders between blocks to ensure deterministic execution orders. But for multicore with partitioned scheduling, they are obviously insufficient since blocks on different cores are now scheduled separately.

To fulfill this critical need, we provide a mechanism that leverages RT blocks and additionally allocates offsets to blocks, to enforce a deterministic execution order that matches the model semantics. Such a mechanism additionally benefits system timing predictability, as tasks on different cores do not simultaneously access the same global variables in the shared memory, alleviating the difficulties to analyze task worst case execution times on multicore [4], [2].

The RT blocks, however, come with a cost on additional memory requirements. Moreover, they may introduce additional functional delays in the control loop, causing control performance degradation and even system instability [10]. On the other hand, the functional delay can also relax the input/output dependency and increase system schedulability. Hence, we consider the problem of software synthesis for Simulink, that preserves the logical-time execution semantics and optimizes a weighted sum of the functional delays in the Simulink model to approximate their impact on control quality.

A. Contributions and Paper Organization.

In this paper, we make the following contributions.

- We leverage a mechanism for ensuring semantics-preserving software implementation of Simulink models on multicore with partitioned fixed-priority scheduling. The idea is to separate their time windows of accessing the shared memory buffer, by assigning appropriate activation offset to software tasks.
- We propose to optimize the software implementation of Simulink models by judicious task priority assignment, task offset assignment, and addition of RT blocks and their functional delays on communication links.
- We design a new, problem-specific exact algorithm that is substantially faster than integer linear programming (ILP) while preserving the optimality of the solution, demonstrated by randomly generated systems and an industrial case study.

2576-3172/18/$31.00 ©2018 IEEE
DOI 10.1109/RTSS.2018.00041

The rest of the paper is organized as follows. Section II summarizes the related work. Section III provides the preliminary knowledge on Simulink model semantics, and Section IV presents the mechanism for semantics preservation on multicore platforms. Section V defines the optimization problem. Section VI proposes the problem specific exact algorithm. Section VII shows the experimental results. Finally, Section VIII concludes the paper.

II. Related Work

Although our focus is on Simulink, we discuss the related work in the broader context of Synchronous Reactive (SR) model of computation, as it is the underlying modeling formalism for Simulink [25]. SR is supported in several other languages and tools such as Esterel [3], Lustre [15], and Prelude [13], [19].

On single-core platforms, the research has been fairly advanced, and we provide a selective review below. Esterel or Lustre models are typically implemented as a single executable that runs according to an event server model [22]. The longest chain of reactions to any event shall be completed within the system base period (the greatest common divisor of all periods in the system). For multi-rate systems, this imposes a very strong condition on real-time schedulability that is typically infeasible in cost-sensitive application domains such as automotive [10]. The commercial code generators for Simulink models (e.g., Simulink Coder from MathWorks) provide two options. The first is a single-task executing at the base period, which is essentially the same approach as [22]. The second is a fixed-priority multitask implementation, where one task is generated for each period in the model, and tasks are scheduled by rate monotonic policy. Caspi et al. [6] provide the conditions of semantics-preservation in a multitask implementation. Di Natale et al. [11] propose to optimize the multitask implementation of multi-rate Simulink models with respect to the control performance and the required memory, and develop a branch-and-bound algorithm. Later in [10], an ILP formulation is provided. The task implementation and schedulability analysis for SR models containing finite state machines (FSMs) are studied in [18] and [29] respectively. Zhao et al. [30] develop a set of optimization techniques to efficiently optimize the real-time software implementing systems with FSMs.

Comparably, the research on the implementation of SR models on multicore and distributed systems is rather limited. Prelude [13], [19] provides rules and operators for the selection of a mapping onto platforms with Earliest Deadline First (EDF) scheduling, including multicore architectures [23], [20]. The enforcement of the partial execution order required by the SR model semantics is obtained in Prelude by a deadline modification algorithm. The communication mechanisms on multicore platforms including those for semantics preservation are discussed in [28]. Pagetti et al. [20] provide design experiences for an avionics case study modeled in Simulink and implemented on a many-core platform. This case study is also used to develop a tool that generates code, where task scheduling is time-triggered and the functional delays are presumed to be given [14]. Puffitsch et al. present approaches to automatically map tasks to cores on a many-core architecture with EDF [23] or tick-based scheduling [24]. The commercial Simulink tool requires the user to specify if a delay block shall be added on a communication link and ensure the associated deadlines are met [25], but this is very difficult without automated tool support. [27] studies the problem of mapping multi-rate synchronous blocks onto multicore architectures with partitioned fixed-priority scheduling. However, it assumes the task execution order and task priority assignment are given. In addition, it only considers the limited case where communication is restricted among blocks with the same period. Our focus is different from [27] in that we assume block to core mapping is given, and we aim to optimize task execution order (and consequently delay block assignment) w.r.t. control performance. Overall, *our paper is the first* to automate and optimize delay block assignment in the synthesis of semantics-preserving software for Simulink models on multicore with fixed-priority scheduling.

On distributed architectures, the implementation of SR models has been discussed in several papers such as [7], [21], [5], [26]. Specifically, techniques for generating semantics-preserving implementations of SR models on Time-Triggered Architecture (TTA) are presented in [7]. Methods for desynchronization in distributed implementations are discussed in [5], [21]. A general mapping framework from SR models to unsynchronized architecture platforms is presented in [26], where the mapping uses intermediate layers with queues and then back-pressure communication channels.

III. Simulink Model Semantics

A Simulink model is represented as a *Directed Graph* $\Gamma = \{\mathcal{N}, \mathcal{E}\}$, where $\mathcal{N} = \{N_1, \ldots, N_{|\mathcal{N}|}\}$ is the set of nodes representing the functional blocks, and $\mathcal{L} = \{L_1, \ldots, L_{|\mathcal{L}|}\}$ is the set of edges representing the communication links between the blocks.

In Simulink, each implementable functional block N_i is triggered periodically, and is associated with a period T_i. That is, the k-th instance of N_i is triggered at time $r_i(k) = k \cdot T_i$. Blocks interface with other blocks using a set of input ports and a set of output ports. Input ports carry signals sampled with the period T_i. The output signals are produced with the same period on the output ports.

Each link $\langle N_i, N_j \rangle$ in $\mathcal{E}$ connects the output port of block N_i (the writer) to an input port of block N_j (the reader). Simulink assumes that for each writer-reader relation, the periods of the reader and writer are *harmonic*. If the output of N_j is directly dependent on its input from N_i, then there is a precedence constraint associated with the link. We refer to this precedence as *direct feedthrough* dependency and denote it as $N_i \rightarrow N_j$. Let $i_j(k)$ be the input to the k-th instance of N_j. The SR semantics specifies that $i_j(k)$ equals the output of the last instance of N_i, denoted by $o_i(m)$, that is triggered no later than the k-th instance of N_j:

$$i_j(k) = o_i(m), \text{ where } m = \max\{n | r_i(n) \leq r_j(k)\}. \quad (1)$$

The SR semantics also allows for delayed communication, where the delay is limited to one unit in Simulink. If the communication is delayed, N_j does not depend on the output of the most recently triggered instance of N_i; instead the previous value is read. That is,

$$i_j(k) = o_i(m-1), \text{ where } m = \max\{n | r_i(n) \leq r_j(k)\}. \quad (2)$$

We refer to this scenario as *unit delay* communication, and denote it as $N_i \xrightarrow{-1} N_j$.

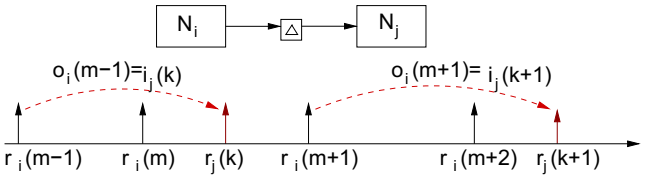

Fig. 1: Input/output relation with unit delay on the communication link.

Figure 1 shows the effect of adding a unit delay on the communication link. We can see that there is more time between the writer instance from which the data is produced and the reader instance by which it is consumed. This gives more flexibility in scheduling the reader/writer blocks. However, this delay requires additional storage in memory for buffering the variables during the time interval between the data production and consumption. Furthermore, the added delay increases end-to-end latency which might cause performance degradation especially for control algorithms.

In summary, in Simulink semantics the data exchanged by two communicating blocks are clearly defined by the model. With direct feedthrough dependencies, the reader reads the data produced by the most recently triggered instance of the writer. For communication with unit delay, data from the previous instance is used. In both cases, there should be no confusion and the writer instance of each data item consumed by a reader is explicitly defined by the model.

IV. Semantics-Preserving Implementation on Multicore

In this paper, we consider the problem where the communication mode (direct feedthrough or unit delay) for each link is part of the decision variables. When generating software code for Simulink models, the implementation shall behave identically to the model, in the sense that the input/output data flows of the model with the selected modes are preserved in the implementation. We focus on multicore architectures with partitioned fixed-priority preemptive scheduling. We assume that each block N_i is implemented by a dedicated real-time software task τ_i, and *use the terms block and task interchangeably*. Thus the number of tasks equals the number of nodes in the directed graph Γ. In the implementation, each task τ_i is statically allocated to a core E_i, and is assigned with a fixed priority p_i, where $p_i > p_j$ represents that τ_i has a higher priority than τ_j. C_i denotes the Worst Case Execution Time (WCET) of τ_i. Also, Simulink assumes that each task τ_i has an implicit deadline, meaning any instance of τ_i shall finish before the trigger time of the next instance.

We consider a semantics-preservation mechanism that combines the RT blocks from Simulink and task offset assignment [28]. RT blocks are a specialization of the more general category of wait-free buffers [8]. They require that the sender and receiver have harmonic periods (one period must be an integer multiple of the other). These blocks are placed between the writer and the reader, to forward appropriate data from the writer to the reader and to provide initial data values when necessary. Specifically, an RT block consists of a shared buffer and an update function that writes the data by the writer to the shared buffer. It executes within the context of the writer at the end of its execution. Correspondingly, the reader reads the data from the shared buffer at the beginning of its execution.

Offset assignment intends to separate the access to shared memory from the communicating blocks on different cores and enforce a global execution order. Specifically, for each block τ_i, we assign an *activation offset* O_i that is smaller than its period T_i. Whenever τ_i is triggered, it will wait until O_i time unit later to be ready for execution. Hence, the *activation time* (the time it is ready for execution) of the k-th instance of τ_i becomes $r_i(k) + O_i$ (but the deadline is still $r_i(k) + T_i$). The worst-case response time (WCRT) of task τ_i, denoted as R_i, is the maximum delay from its activation to its finish. It is computed as the least fixed point of the following equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{3}$$

where $hp(i)$ is the set of tasks that have higher priority than τ_i and are allocated to the same core.

We now discuss the requirements of semantics preservation and how the proposed mechanism works. A general rule is that the execution orders among blocks must be properly enforced, as defined below.

Definition 1. An *execution order* $f_{i,j}$ denotes the constraint that τ_i must execute before τ_j whenever they are triggered together. Equivalently, whenever two tasks τ_i and τ_j are triggered at the same time, τ_i must complete before τ_j starts.

A. Intra-core Communication

For intra-core communication, i.e., when the reader and writer are assigned to the same core, the fact that these blocks are on the same core and the priority order is well defined helps ensure proper operation of RT blocks and hence semantics preservation.

For a direct feedthrough link $\tau_i \rightarrow \tau_j$, $f_{i,j}$ shall be enforced to ensure τ_i executes before τ_j. This requires to assign τ_i with a higher priority than τ_j. Additionally, since blocks have activation offset, τ_i shall be activated before τ_j. The RT block behaves like a Zero-Order Hold block. Its output update function executes at the slower rate of the two, but within (and at the priority of) the writer.

The scheduling diagram is depicted in Figure 2. Specifically, the offset and priority assignments ensure that the writer τ_i executes first, followed by the RT block's output update function (denoted as striped boxes in the figure). Afterward, the reader τ_j reads data from the RT block. The data will be held by the RT block and read by all instances of the reader until it is updated again.

This rule is formally summarized as follows.

Rule 1. *Enforcing execution order $f_{i,j}$ for intra-core communication from τ_i to τ_j implies the following constraints*

$$\forall \langle \tau_i, \tau_j \rangle \text{ with } E_i = E_j : \; f_{i,j} \Rightarrow (O_i \leq O_j) \wedge (p_i > p_j) \tag{4}$$

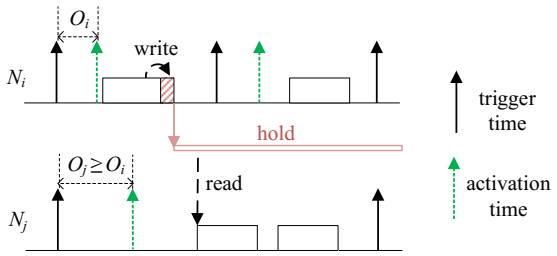

Fig. 2: Enforcing execution order $f_{i,j}$ for intra-core feedthrough communication $\langle \tau_i, \tau_j \rangle$.

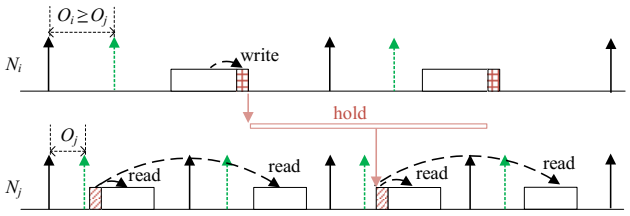

Fig. 3: Enforcing execution order $f_{j,i}$ for intra-core unit delay communication $\langle \tau_i, \tau_j \rangle$.

When a low rate writer τ_i communicates to a high rate reader τ_i, enforcing execution order $f_{i,j}$ typically worsens schedulability since it violates the rate-monotonic policy. One alternative is to add a unit delay to relax the direct feedthrough dependency.

For a unit delay communication $\tau_i \overset{-1}{\rightarrow} \tau_j$, the reader τ_j should be assigned with a higher priority, and the offset of the writer τ_i shall be no smaller than that of τ_j. The purpose of the assignment is to prevent race condition caused by τ_j preempting τ_i while τ_i is updating the shared buffer. In this case, the RT block behaves like a Unit Delay block plus a Hold block (Sample and Hold). It supplies an initial value for the data and holds the delayed data values for the necessary time period. As illustrated in Figure 3, the RT block state update function (the gridded box) executes in the context (and at the rate) of the lower priority writer. The RT block output update function (the striped box) runs in the context of the higher priority reader, but at the rate of the slower block.

The design rule is formally summarized as follows. In fact, it is symmetric to Rule 1.

Rule 2. *For intra-core communication from τ_i to τ_j, adding a **unit delay** between τ_i and τ_j requires to enforce execution order $f_{j,i}$ that implies the following constraints*

$$\forall \langle \tau_i, \tau_j \rangle \text{ with } E_i = E_j : \; f_{j,i} \Rightarrow (O_i \geq O_j) \wedge (p_i < p_j) \quad (5)$$

B. Inter-core Communication

When the reader and writer blocks are assigned to different cores with partitioned scheduling, preserving the Simulink semantics is more challenging since there is no notion of global priority. Hence, we shall rely on offset assignment to enforce a global execution order and avoid simultaneous access to shared memory from the communicating blocks on different cores.

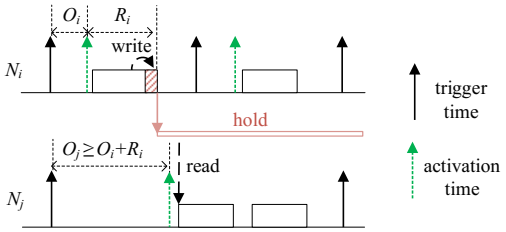

Fig. 4: Enforcing execution order $f_{i,j}$ for inter-core feedthrough communication $\langle \tau_i, \tau_j \rangle$.

The implementation of the RT blocks themselves remains the same as the single-core case. Feedthrough RT blocks consist of an output update function executing within the context of the writer but at the rate of the slower block. The *global shared variable* is the output variable of the RT block. Unit delay RT blocks consist of a state update function executing along with the writer, and an output update function executing in the context of the reader but at the rate of the slower block. The *global shared variable* is the state variable of the RT block.

For direct feedthrough communication $\tau_i \rightarrow \tau_j$, to ensure that τ_i executes before τ_j, it suffices to assign τ_j with an activation offset no smaller than the offset of τ_i plus the WCRT of τ_i, such that τ_j can only start after the finish of the instance of τ_i that feeds it. Formally, the design rule is stated in Rule 3. The corresponding scheduling diagram is illustrated in Figure 4. As in the figure, the global variable shared between cores, the output variable of the RT block, is never accessed simultaneously from different cores: the RT block updates it before the reader's activation time.

Rule 3. *Enforcing execution order $f_{i,j}$ for a writer τ_i and reader τ_j allocated on different cores requires the following constraint*

$$\forall \langle \tau_i, \tau_j \rangle \text{ with } E_i \neq E_j : \; f_{i,j} \Rightarrow R_i + O_i \leq O_j \quad (6)$$

For a unit delay communication $\tau_i \overset{-1}{\rightarrow} \tau_j$, we assign τ_i with an offset no smaller than the offset of τ_j plus the delay caused by performing the RT block output update (denoted as $R_{i,j}^{RT}$). Since this update is performed in the context of τ_j, $R_{i,j}^{RT}$ can be computed as the WCRT of τ_j assuming its WCET is that of the RT block output update function C^{RT}. That is,

$$R_{i,j}^{RT} = C^{RT} + \sum_{k \in hp(j)} \left\lceil \frac{R_{i,j}^{RT}}{T_k} \right\rceil \cdot C_k \quad (7)$$

As shown in Figure 5, the writer τ_i may not start until the RT block output update has finished execution to allow the RT block copies from its state variable (and consequently the data generated by the previous instance of τ_i). Implicitly, a partial execution order $f_{j,i}$ is enforced. Also, the global shared variable, the state variable of the RT block, is never accessed simultaneously by tasks on the two cores: the RT block state

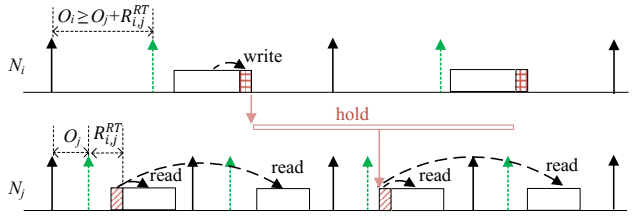

Fig. 5: Enforcing execution order $f_{j,i}$ for writer τ_i and reader τ_j on different cores.

update function (gridded box in the figure) is guaranteed to execute after the previous RT block output update (striped box) is finished and before the next one starts. The rule can be stated as follows.

Rule 4. *For inter-core communication from τ_i to τ_j, adding a **unit delay** between τ_i and τ_j requires to enforce execution order $f_{j,i}$ that implies the following constraints*

$$\forall \langle \tau_i, \tau_j \rangle \text{ with } E_i \neq E_j : \; f_{j,i} \Rightarrow R^{RT}_{i,j} + O_j \leq O_i \quad (8)$$

V. Problem Definition

The previous section shows that different priority assignments and execution orders require different numbers of unit delay RT blocks. The use of unit delay RT blocks comes at the cost of introducing additional functional delay, which may worsen the control performance. In this paper, we consider the problem of optimizing semantics-preserving implementation of Simulink models such that the weighted sum of added unit delay blocks is minimized, where the weight for each link is determined by the designer, based on the effect of added unit delay on the control performance. The decision variables are the task execution order (and consequently the addition of unit delay), priority assignment, and offset assignment. The constraints include system schedulability and those implied by the execution order enforcement. Task periods, WCETs, and allocation to cores are assumed to be given.

By Rules 1–4, a unit delay RT block is introduced whenever an execution order $f_{j,i}$ for a writer τ_i and reader τ_j is enforced. Thus the objective is equivalent to minimizing the weighted cost of enforcing $f_{j,i}$ for all writer-reader pairs $\langle \tau_i, \tau_j \rangle$. We introduce a set of binary variables $t_{i,j}$ defined as

$$t_{i,j} = \begin{cases} 1, & f_{i,j} \text{ is enforced} \\ 0, & \text{otherwise} \end{cases} \quad (9)$$

Each link $\langle \tau_i, \tau_j \rangle$ introduces two binary variables $t_{i,j}$ and $t_{j,i}$ corresponding to two possible execution orders. The optimization problem can then be formally expressed as

$$\begin{aligned} &\min_{\forall \mathbf{P}, \mathbf{O}, \mathbf{t}} \sum_{\forall \langle \tau_i, \tau_j \rangle} w_{i,j} \cdot t_{j,i} \\ s.t.\ &\text{Schedulability} \\ &t_{i,j} = 1 \implies \text{implied design contraint by } f_{i,j}, \ \forall t_{i,j} \\ &t_{i,j} + t_{j,i} = 1, \ \forall i \neq j \\ &t_{i,j} \geq t_{i,k} + t_{k,j} - 1, \ \forall i \neq j \neq k \end{aligned} \quad (10)$$

where $\mathbf{P} = [p_1, ...p_n]$ and $\mathbf{O} = [O_1, ...O_n]$ represent the vectors of priority and offset assignment respectively, $\mathbf{t} = [t_{i,j}, t_{j,i} | \langle \tau_i, \tau_j \rangle]$ is the set of execution order variables, and $w_{i,j}$ is the cost on adding a unit delay to the link $\langle \tau_i, \tau_j \rangle$. The last two sets of constraints correspond to anti-symmetry and transitivity of execution orders. The former means that if τ_i has a higher order than τ_j ($t_{i,j} = 1$), then τ_j must have a lower order than τ_i ($t_{j,i} = 0$). The later enforces that if τ_i has a higher order than τ_k ($t_{i,k} = 1$) and τ_k has a higher order than τ_j ($t_{k,j} = 1$), then τ_i must have a higher order than τ_j ($t_{i,j} = 1$).

In (10), the (rather simplified) objective function approximates the impact of unit delays on control performance. It assumes that unit delay blocks are independent from each other in causing control degradation. The cost of a unit delay block can be computed following the procedure in e.g., [16]. Specifically, it first simulates the control model configured with a set of (selected) scenarios of unit delay addition, to get the control error (the difference between the control output and the reference) corresponding to each scenario. It then uses a linear function to approximate the dependency of the control performance (in terms of control error) on the added delay.

VI. Customized Optimization Algorithm

A direct ILP formulation of the problem (as detailed in Appendix A) is inherently complex, mainly caused by the formulation of priority assignment and response time analysis, which introduces $O(n^2)$ number of integer variables. In addition, the extensive use of big-M method in the formulation also increases numerical difficulty. As a result, this approach, even solved with commercial ILP solvers such as CPLEX, does not scale well to large systems.

In this paper, we propose an alternative technique that is exact but runs much faster than ILP. Our main idea is to use a simple, abstract form of feasibility constraints to hide the details of response time analysis, schedulability constraints, and execution order implied constraints from Problem (10). These constraints are handled using a dedicated procedure that is much more efficient than formulating them in ILP. Central to our abstraction technique is the concept of Minimal Infeasible partial eXecution Orders (MIXO), which represents a minimal set of execution orders that is sufficient to cause infeasibility. We first give its definition and study its property.

A. The concept of MIXO

Definition 2. For task system Γ, we define a *partial execution order set* $F = \{f_{i_1,j_1}, .., f_{i_m,j_m}\}$ as a collection of execution orders. The number of elements in F is denoted as $|F|$.

Definition 3. A task system Γ is said to be *F-feasible* for a given partial execution order set F, or informally F is feasible, if and only if there exists a priority and offset assignment such that (i) all tasks are schedulable; and (ii) the implied constraints by each execution order $f_{i,j} \in F$ are satisfied. Formally, this is described by the following constraint satisfiability problem

$$\begin{aligned} &\min 0 \\ s.t.\ &\text{Schedulability} \\ &\text{Constraint implied by } f_{i,j} \ , \forall f_{i,j} \in F \end{aligned} \quad (11)$$

τ_i	T_i	C_i	E_i
0	100	20	0
1	100	40	0
2	20	10	1
3	200	96	1

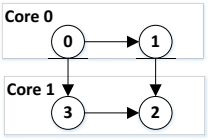

Fig. 6: An illustrative system Γ_e on a dual-core processor. All weights on the links are assumed to be 1.

Example 1. Considering the example Γ_e in Figure 6 and a partial execution order set $F = \{f_{0,1}, f_{3,2}, f_{1,2}, f_{0,3}\}$. Γ_e is F-feasible if and only if the following problem is feasible

$$\begin{aligned} &\min 0 \\ s.t.\ &\text{Formulation of } R_i \text{ as in Appendix A1, } \forall i \\ &R_i + O_i \leq T_i, \ \forall i \\ &f_{0,1} : O_1 \geq O_0 \wedge p_0 > p_1 \\ &f_{3,2} : O_2 \geq O_3 \wedge p_3 > p_2 \\ &f_{1,2} : O_2 \geq R_1 + O_1 \\ &f_{0,3} : O_3 \geq R_0 + O_0 \end{aligned} \tag{12}$$

F is obviously infeasible since given that $C_3 > T_2$, any priority assignment that schedules τ_3 at higher priority than τ_2 would cause τ_2 to miss its deadline.

We now reformulate problem (10) using the concept of F-feasibility. Specifically, we re-interpret the original problem (10) as a problem of finding the optimal feasible partial execution order set F, which specifies one execution order, $f_{i,j}$ or $f_{j,i}$, for each communication link $\langle \tau_i, \tau_j \rangle$.

$$\begin{aligned} &\min_{\forall F} \sum_{\forall \langle \tau_i, \tau_j \rangle} w_{i,j} \cdot t_{j,i} \\ s.t.\ &\Gamma \text{ is } F\text{-feasible} \\ &t_{i,j} + t_{j,i} = 1, \ \forall i \neq j \\ &t_{i,j} \geq t_{i,k} + t_{k,j} - 1, \ \forall i \neq j \neq k \end{aligned} \tag{13}$$

In the following, we introduce our abstraction technique for formulating the constraints of F-feasibility.

Theorem 1. Let F and F' be two partial execution order sets such that $F' \subseteq F$. It is

$$\Gamma \text{ is } F\text{-feasible} \Rightarrow \Gamma \text{ is } F'\text{-feasible} \tag{14}$$

Proof: Since F' will impose constraints that are a subset of those from F, if a feasible priority and offset assignment exists for F, it must be a feasible assignment for $F^{'}$ too. ■

Corollary 1. By contrapositive law, for two partial execution order sets $F' \subseteq F$, there is

$$\Gamma \text{ is not } F'\text{-feasible} \Rightarrow \Gamma \text{ is not } F\text{-feasible} \tag{15}$$

From the perspective of the reformulated problem (13), Corollary 1 suggests that if a partial execution order set F' is known to be infeasible, then the search for the optimal feasible F shall avoid any superset of F'. Thus an infeasible partial execution order set F' provides a clue for performing search space reduction. Intuitively, the smaller the F', the greater the number of infeasible partial execution order sets it can capture. In the following, we introduce a special type of partial execution order set that is minimal and infeasible.

Definition 4. A partial execution order set U is a *Minimal Infeasible partial eXecution Order set (MIXO)* if and only if

- Γ is not U-feasible
- $\forall F \subset U$, Γ is F-feasible

Example 2. Consider two partial execution order sets $F_1 = \{f_{0,1}, f_{3,2}\}$ and $F_2 = \{f_{3,2}\}$ for the example system in Figure 6. Though both are infeasible, F_1 is not a MIXO since $F_2 \subset F_1$ and F_2 is infeasible. F_2 is a MIXO however, as its only proper subset $F = \emptyset \subset F_2$ is feasible. Intuitively, F_1 is redundant in the presence of F_2 in the sense that the infeasibility of F_1 is implied by that of F_2.

A MIXO U implies the following constraint

$$\sum_{\forall f_{i,j} \in U} t_{i,j} \leq |U| - 1 \tag{16}$$

since we cannot simultaneously satisfy all the execution orders in U due to its infeasibility. We call (16) the implied feasibility constraint by U. Our main idea is to use the above constraint as an alternative for modeling F-feasibility. Comparing with the ILP formulation in Section A, (16) abstracts away the details of priority assignment, response time analysis, and the implied constraints by the execution orders in U.

In the following we first discuss algorithms for calculating MIXO, then introduce an iterative optimization procedure that selectively adds MIXO implied constraints.

B. MIXO Calculation

Given an infeasible partial execution order set F, Algorithm 1 computes a MIXO. Specifically, the algorithm iteratively visits each execution order $f_{i,j}$ in F and attempts to remove it. If removing $f_{i,j}$ allows F to be feasible, then $f_{i,j}$ is added back to F. Intuitively, this suggests that $f_{i,j}$ is part of the reason for causing F-infeasibility. Otherwise $f_{i,j}$ is removed. At the end of the algorithm, F is maintained to be infeasible and has a property that removing any element from it causes it to be feasible. Thus the resulting F satisfies the two conditions in Definition 4, and it is a MIXO.

Note that an infeasible set F may contain multiple MIXOs. To compute a different MIXO $\mathcal{U}'$ from F, one way is to first perturb F into a different infeasible F' such that $F' \not\supseteq \mathcal{U}$ (i.e., by removing an element $f_{i,j} \in \mathcal{U}$ from F), then apply Algorithm 1 on F'. Since $F' \not\supseteq \mathcal{U}$, it is guaranteed that the newly computed MIXO $\mathcal{U}'$ will be different from $\mathcal{U}$.

The key of Algorithm 1 is a procedure for testing F-feasibility (Line 4). The procedure needs to be efficient since it is invoked $|F|$ times for each run of Algorithm 1. In the following, we first introduce an exact analysis of F-feasibility and then a necessary-only but much faster analysis.

1) Exact Analysis: The exact analysis of F-feasibility requires to accurately solve the constraint satisfiability problem (11). A straightforward solution is to use an ILP formulation similar to the one introduced in Appendix A. However, this is

Algorithm 1 Algorithm for computing MIXO

1: **function** MIXOCOMPUTATION(Infeasible execution order set F, Task Set Γ)
2: **for** each $f_{i,j} \in F$ **do**
3: $F = F \backslash \{f_{i,j}\}$
4: **if** F becomes feasible **then**
5: $F = F \cup \{f_{i,j}\}$
6: **end if**
7: **end for**
8: **return** F
9: **end function**

very slow due to similar complexity issues to those discussed at the beginning of this section. In this paper, we leverage the MUDA (Maximal Unschedulable Deadline Assignment)-guided priority assignment optimization framework [31]. It can solve the following problem highly efficiently, where the decision space consists of priority assignment, and $G(\mathbf{X}) \leq 0$ represents additional linear constraints on task WCRTs

$$\begin{aligned} &\min 0 \\ s.t.\ &\text{Schedulability} \\ &G(\mathbf{X}) \leq 0 \end{aligned} \tag{17}$$

The main idea of the MUDA-guided framework is to avoid formulating task WCRT calculation and view (17) as a problem of finding a schedulable deadline assignment that satisfies $G(\mathbf{X}) \leq 0$. For instance, consider Example 1, MUDA-guided framework re-interprets the F-feasibility problem in (12) as the following deadline assignment problem

$$\begin{aligned} &\min 0 \\ s.t.\ &d_i + O_i \leq T_i,\ \forall i \\ &f_{0,1} : O_1 \geq O_0 \wedge p_0 > p_1 \\ &f_{3,2} : O_2 \geq O_3 \wedge p_3 > p_2 \\ &f_{1,2} : O_2 \geq d_1 + O_1 \\ &f_{0,3} : O_3 \geq d_0 + O_0 \\ &\text{deadline assignment } d_1, ...d_n \text{ is schedulable} \end{aligned} \tag{18}$$

where variable d_i represents the (virtual) deadline of τ_i.

The MUDA-guided framework consists of two main components: (i) an ILP that only tries to find a deadline assignment satisfying $G(\mathbf{X}) \leq 0$; and (ii) a revised Audsley's algorithm [1] that calculates task WCRTs and checks whether the deadline assignment returned from solving the ILP allows a schedulable priority assignment. If not, it generalizes the solution to a MUDA, converts to an abstract form of constraints and adds back to the ILP for refinement.

We now look closer on how F-feasibility analysis can be casted as a MUDA-guided optimization problem. Consider the example in (12). The constraints can be divided into two parts:

Offset constraints:	Partial priority order constraints:
$O_1 \geq O_0$	$p_0 > p_1$
$O_2 \geq O_3$	$p_3 > p_2$
$O_2 \geq R_1 + O_1$	R_i calculation, $\forall i$
$O_3 \geq R_0 + O_0$	
$R_i + O_i \leq T_i,\ \forall i$	

(19)

Algorithm 2 Algorithm for computing $\check{R}_i$

1: **function** RMIN(Task τ_i, Partial priority order constraints **PPO**)
2: **if** Unschedulable w.r.t **PPO then**
3: **return** $D_i + 1$
4: **end if**
5: Use binary search to find the smallest $d_i \in [C_i, D_i]$ that makes the system schedulable w.r.t. **PPO**
6: **return** d_i
7: **end function**

The offset constraints can be included in $G(\mathbf{X}) \leq 0$ in (17). The partial priority order constraints can be enforced in the second component of the framework, namely, the revised Audsley's algorithm for checking schedulability and computing MUDAs. Specifically, like Audsley's algorithm, it tries to find a task that can be assigned at a particular priority level starting from the lowest priority. However, when choosing the candidate task at the current priority level, it shall guarantee that no given partial priority order (PPO) constraint is violated.

We now present a necessary only analysis that runs much faster, hence can quickly find MIXOs.

2) Necessary Only Analysis: Given a partial execution order set F and consequently a set of partial priority order constraints, the necessary only analysis for F-feasibility uses a quick procedure (Algorithm 2) to derive the smallest WCRT for each task τ_i (denoted as $\check{R}_i$) as well as that of the RT block update function (denoted as $\check{R}_{i,j}^{RT}$), among all priority assignments that meet the partial priority order constraints. If the offset constraints (the left-hand side of Equation (19)) are satisfiable assuming $R_i = \check{R}_i$ and $R_{i,j}^{RT} = \check{R}_{i,j}^{RT}$, then F is definitely infeasible.

Algorithm 2 gives a procedure for computing $\check{R}_i$. It takes as input the set of partial priority order constraints implied by F. The observation is that computing $\check{R}_i$ is equivalent to finding the minimum deadline for τ_i while maintaining system schedulability (i.e., all tasks are schedulable), which can be done using a simple binary search procedure. $\check{R}_{i,j}^{RT}$ is computed similarly. The schedulability w.r.t. the PPO constraints at Line 2 and Line 5 can be tested using a similar revised Audsley's algorithm as in Section VI-B1.

The accuracy of the analysis can be further improved by finding a lower bound $\check{O}_i$ on each task offset O_i. Given all $\check{R}_i$ and $\check{R}_{i,j}^{RT}$, the offset constraints as those in (19) enforce a lower bound on the task offsets. Using (19) as an example, we can derive $\check{O}_3 = \check{R}_0$, and $\check{O}_2 = \max\{\check{R}_0, \check{R}_1\}$.

Since each task τ_i needs to be schedulable, $\check{O}_i$ suggests that any priority assignment needs to additionally satisfy $R_i \leq T_i - \check{O}_i$. Equivalently, this sets a stricter deadline $\check{D}_i = T_i - \check{O}_i$ for τ_i, which can be used to refine the schedulability analysis at Lines 2 and 5 of Algorithm 2. Specifically, the new stricter deadline setting may cause $\check{R}_i$ to further increase, which again increases $\check{O}_i$. We propose an iterative procedure for the above analysis until the fixed point is reached (i.e., none of $\check{O}_i$, $\check{R}_i$ or $\check{R}_{i,j}^{RT}$ of any task changes), as detailed in Algorithm 3.

Example 3. Consider a partial execution order set $F = \{f_{0,3}\}$. We now show how the necessary only analysis in Algorithm 3

Algorithm 3 Necessary only analysis for F-feasibility

1: **function** F−FEASIBILITY(Partial execution order set F)
2: Extract offset constraints **OFF** from F
3: Extract partial priority order constraint **PPO** from F
4: **while** $\check{O}_i$, $\check{R}_i$ or $\check{R}_{i,j}^{RT}$ of any τ_i changes **do**
5: Compute all $\check{R}_i$, $\check{R}_{i,j}^{RT}$ using Algorithm 2
6: **if** Unsatisfiable w.r.t **OFF then return** false
7: **end if**
8: Compute $\check{O}_i$ according to **OFF**
9: Update $\check{D}_i = T_i - \check{O}_i$ for all τ_i
10: **end while**
11: **return** true
12: **end function**

reasons its infeasibility. By Rule 3, F imposes the following constraint

$$O_3 \geq O_0 + R_0 \tag{20}$$

At Line 4, Algorithm 3 computes $\check{R}_0 = 20$ and $\check{R}_3 = 196$. Assuming $R_0 = \check{R}_0$ and $R_3 = \check{R}_3$, the following offset constraints are obviously unsatisfiable.

$$\begin{aligned} O_3 &\geq O_0 + R_0 \\ R_3 + O_3 &\leq T_3 \end{aligned} \tag{21}$$

Thus the algorithm returns false at Line 6.

Now consider another partial execution order set $F = \{f_{3,0}\}$. By Rule 4, it imposes the following constraint

$$O_0 \geq O_3 + R_{0,3}^{RT} \tag{22}$$

At Line 4, Algorithm 3 computes $\check{R}_{0,3}^{RT} = 10$ and $\check{R}_0 = 20$. All the offset constraints are satisfied at Line 5. Thus the algorithm proceeds to compute all $\check{O}_i$, to obtain $\check{O}_0 = 10$ and $\check{O}_3 = 0$ according to (22). At Line 9, the deadline of τ_0 is updated to $\check{D}_0 = T_0 - \check{O}_0 = 100 - 10 = 90$. In the second iteration of the while loop, Algorithm 3 recomputes all $\check{R}_i$, $\check{R}_{i,j}^{RT}$ and $\check{O}_i$, and finds them to be the same as the previous iteration, suggesting that a fixed point is reached. Thus the algorithm exits the while loop and returns true (i.e., $F = \{f_{3,0}\}$ is feasible) at Line 11.

Though Algorithm 3 is a necessary only F-feasibility analysis, it is quite useful for quickly identifying obviously infeasible Fs. Next, we present the overall customized algorithm that combines the use of necessary only and exact analyses to greatly improve the algorithm efficiency.

C. MIXO-guided Framework

One issue of designing a framework based on MIXO-implied constraint (16) for modeling the feasibility regions is that the total number of MIXOs grows exponentially w.r.t. the size of the system. For example, for a system with m communication links, there are $2m$ partial execution order variables, and the total number of MIXOs can be C_{2m}^{m} (a MIXO cannot contains another MIXO). Modeling the entire feasibility region would need to enumerate all MIXOs and their implied constraints, which is obviously impractical for large systems. However, this is rarely necessary. In our evaluation on randomly generated systems, we observe that in most cases, the optimal solution can be defined by a relatively small number of MIXOs. Thus, we propose an iterative refinement procedure that selectively explores and adds MIXO-implied constraints guided by the optimization objective. That is, we derive and enforce MIXO-implied constraints only when the optimization algorithm returns infeasible solutions (i.e., the returned partial execution order set is infeasible).

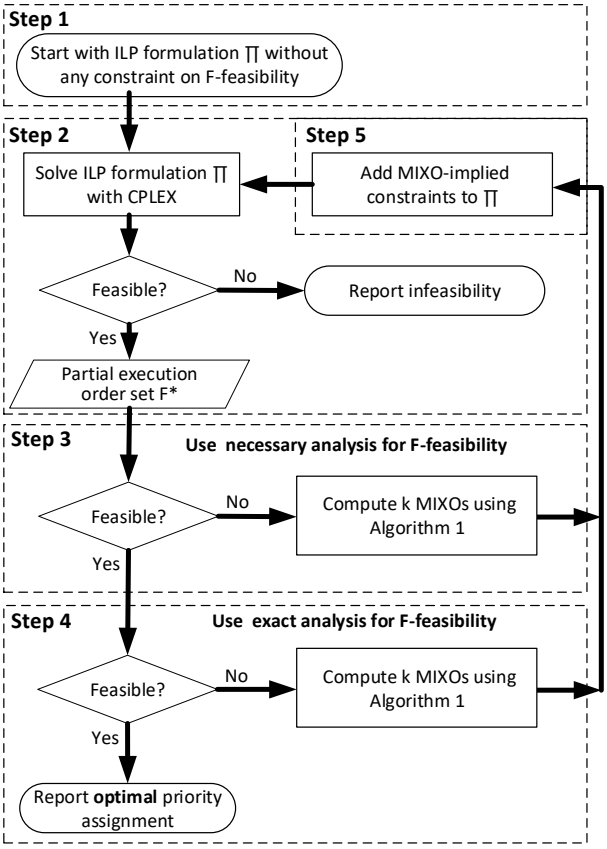

Fig. 7: MIXO-guided optimization algorithm.

We now present the algorithm in a stepwise manner. Also, Figure 7 summarizes the algorithm flow chart.

Initially, the algorithm takes as input a Simulink model specified in a Directed Graph Γ and with given information on periods, WCETs and core allocation for each functional block.

Step 1. The algorithm starts with an initial problem formulation Π as follows

$$\begin{aligned} &\min_{\forall F} \sum_{\forall \langle \tau_i, \tau_j \rangle} w_{i,j} \cdot t_{j,i} \\ &s.t.\ t_{i,j} + t_{j,i} = 1,\ \forall i \neq j \\ &\quad t_{i,j} \geq t_{i,k} + t_{k,j} - 1,\ \forall i \neq j \neq k \end{aligned} \tag{23}$$

Compared with the full problem (13), the constraint on F-feasibility is initially left out.

Step 2. Solve problem Π using integer linear programming solvers (e.g., CPLEX). If Π is infeasible, then the original problem is infeasible and the algorithm terminates (see Remark 1). Otherwise, from the valuation on $\{t_{i,j}\}$, construct the partial execution order set F^* as $F^* = \{f_{i,j} | t_{i,j} = 1\}$.

Step 3. Apply the necessary only feasibility analysis in Algorithm 3 to test the feasibility of F^*. If F^* is deemed feasible, go to **Step 4** (and apply the exact analysis). Otherwise, apply Algorithm 1 using the necessary analysis to compute k MIXOs where k is a predefined parameter. Go to **Step 5**.

Step 4. Apply the exact feasibility analysis introduced in Section VI-B1 on F^*. If F^* is feasible, then terminate and return the optimal F^*. The returned F^* specifies the set of execution orders that shall be enforced and thus the set of required unit delay RT blocks. The corresponding priority and offset assignments are derived from the feasibility check of F^*. They can then be used in the actual implementation of the Simulink model. If F^* is not feasible, apply Algorithm 1 using the exact analysis to compute k MIXOs. Go to **Step 5**.

Step 5. Add the MIXO-implied feasibility constraints of the k computed MIXOs to problem Π. Return to **Step 2**.

Remark 1. In **Step 2**, it is possible that problem Π becomes infeasible at some point. This happens for example, when the utilization of the system is so high that no priority assignment can schedule it. In this case, given any partial execution order set F, Algorithm 1 always returns an empty set as the calculated MIXO. The MIXO-implied constraint by an empty set, as defined in (16), would be $0 \leq -1$, which causes Π to be infeasible.

The algorithm is also guaranteed to *terminate*. This is because the number of MIXOs is finite, and during each iteration between Steps 2–5, the algorithm will find new MIXOs that are different from known ones.

Finally, if a solution is deemed feasible at Step 4, it must be *optimal* with respect to the original problem, as it is optimal to a relaxed problem Π: Π only includes the implied constraints of a subset of all MIXOs.

The worst case complexity of the MIXO-guided algorithm may be the same as that of the ILP. Its efficiency mainly comes from exploiting a small number of constraints in simple forms to find the optimal solution, which is often sufficient in practice.

In the following, we demonstrate the proposed framework using the example system in Figure 6.

Example 4. The algorithm starts with the initial problem Π as follows, since all weights are assumed to be 1

$$\min t_{1,0} + t_{3,0} + t_{2,1} + t_{2,3} \tag{24}$$

Π also includes the anti-symmetry and transitivity constraints of $t_{i,j}$ variables, and we omit them for ease of presentation.

Iteration 1: Solving Π returns the partial execution order set.

$$F^* = \{f_{0,1}, f_{0,3}, f_{1,2}, f_{3,2}\} \tag{25}$$

The system Γ_e is not F-feasible due to the existence of partial execution order $p_3 > p_2$ (as explained in Example 1). The algorithm then computes the following MIXO

$$U_1 = \{f_{3,2}\} \tag{26}$$

The following MIXO-implied constraint is added to Π.

$$t_{3,2} \leq 0 \tag{27}$$

Iteration 2: Solving the updated problem Π returns the following partial execution order set

$$F^* = \{f_{0,1}, f_{0,3}, f_{1,2}, f_{2,3}\} \tag{28}$$

F^* is not feasible and the algorithm computes the following MIXOs

$$U_2 = \{f_{1,2}\}, U_3 = \{f_{0,3}\} \tag{29}$$

The following MIXO-implied constraints are added to Π.

$$t_{1,2} \leq 0 \wedge t_{0,3} \leq 0 \tag{30}$$

Iteration 3: Solving Π returns the following partial execution order set

$$F^* = \{f_{0,1}, f_{3,0}, f_{2,1}, f_{2,3}\} \tag{31}$$

F is now feasible. The optimal solution uses unit delay blocks for communication link $\langle \tau_3, \tau_2 \rangle$, $\langle \tau_0, \tau_3 \rangle$ and $\langle \tau_1, \tau_2 \rangle$. The corresponding priority assignments for the two cores are $\tau_0 \succ \tau_1$ and $\tau_2 \succ \tau_3$.

VII. Experimental Results

In this section, we present the results of our experimental evaluation. We compare the proposed MIXO-guided optimization framework (denoted as `MIXO-guided`) and the direct ILP formulation (denoted as `ILP`), using randomly generated systems with different settings, as well as an industrial case study of automotive fuel injection system. Since both are exact algorithms, we compare them only in terms of their runtimes.

A. Experiments on Random Systems

We first evaluate the performance of `MIXO-guided` and `ILP` on random systems across a wide range of settings. We use the tool Task Graphs For Free (TGFF) [12] for generating random directed graphs as the Simulink model. The maximum number of writers to each task is limited to 3 and the maximum number of readers is 2. We consider a dual-core platform. The total number of tasks is varied from 10 to 70, which are then randomly and evenly distributed to the two cores. The system total utilization is randomly selected from the interval $[1.4, 1.8]$. The utilization of each task is then generated using the UUnifast-Discard algorithm [9]. Task periods are randomly chosen from a predefined set $\{1, 5, 10, 20, 40, 50, 100, 200, 400, 500, 1000\}$ms, which contains all the periods from the automotive benchmark [17]. To avoid excessive waiting time on difficult problem instances, we set a time limit of 30min for both techniques.

The average runtime by the two techniques is summarized in Figure 8. Each data point in the figure represents the average out of 1000 randomly generated systems. For systems with up to 40 tasks, `MIXO-guided` and `ILP` have comparable

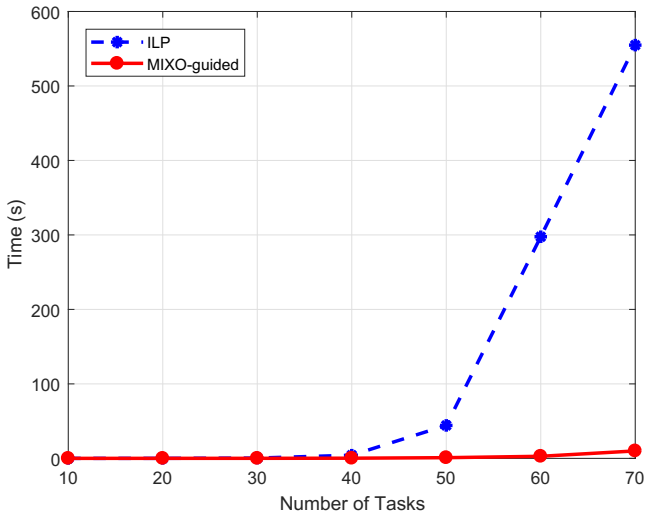

Fig. 8: Run Time vs Number of Tasks

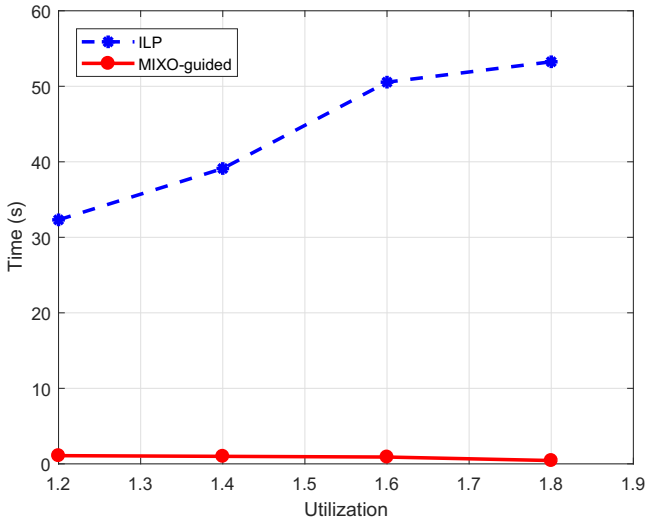

Fig. 9: Run Time vs Utilization

runtimes. However, for larger systems of 60 or more tasks, the runtime of `ILP` drastically increases. `MIXO-guided` becomes 10× to 100× faster than `ILP` for systems above 50 tasks.

We then fix the total number of tasks to be 50, and check how the runtimes of the algorithms vary with respect to a given system utilization. Hence, we fix the system utilization to be some value between 1.2 and 1.8, and collect the average runtime of both algorithms. Figure 9 illustrates the results. As in the figure, `MIXO-guided` always runs faster than `ILP`, and the gap in their algorithm efficiency becomes larger for higher utilization: at 180% system utilization (averagely 90% on each core), `MIXO-guided` is about 1 to 2 orders of magnitudes faster than `ILP`.

B. Automotive Fuel Injection System

Our second experiment evaluates on an industrial case study of a simplified fuel-injection system [10]. The system

# Tasks		Runtime		Objective	Util
E_0	E_1	ILP	MIXO-guided		
90	0	39197 sec	0.33 sec	21	94%
80	10	13967 sec	1.36 sec	21	190%
70	20	8866 sec	10.47 sec	21	190%
60	30	2325 sec	9.22 sec	21	190%
45	45	1219 sec	5.46 sec	21	190%

TABLE I: Results for the fuel-injection system

contains 90 blocks executing at 7 different periods: 4, 5, 8, 12, 50, 100, and 1000 ms. There are in total 106 communication links among the blocks, 37 of which are from high rate to low rate blocks, and 31 are from low rate to high rate. The communication graph and the task period and WCET can be found in [10]. The case study was originally configured to run on a single-core platform, with a total utilization of 94%. In this paper, we modify the case study for use on a dual-core processor. Specifically, we scale the WCET of each task to reach a total utilization of 190% (on average 95% for each core). We consider various task partition schemes on the two cores, as well as the original system on a single-core architecture.

The results are summarized in Table I, where the first two columns are the number of tasks allocated to each of the two cores. As in the table, while both are capable of finding the same optimal solution, the MIXO-guided optimization technique is 2 to 5 orders of magnitude faster than ILP. It is also interesting to note that although the size of the fuel injection case study is larger than the randomly generated systems in the previous experiment, the performance of MIXO-guided algorithm is sometimes better.

This is mainly due to the following characteristics of the case study: (i) The total utilization of the case study is very high, almost approaching the limit that allows schedulability; (ii) For many of the low rate to high rate communication links $\langle \tau_i, \tau_j \rangle$, the periods of the reader and writer are drastically different (e.g., a 1Hz block communicating to a 250Hz block). Since the WCETs of blocks are roughly proportional to their periods, enforcing an execution order for these links where the lower rate task executes first would easily cause system unschedulability. As a result, for many of the low rate to high rate communication links, there is only one possible partial execution order that may be feasible. For other high rate to low rate communications $\langle \tau_i, \tau_j \rangle$, enforcing $f_{i,j}$ is typically the optimal decision as it mostly helps schedulability without having to introduce unit delay blocks.

The proposed MIXO-guided optimization technique readily exploits such characteristics. Specifically, since it starts with an over-approximation of the feasibility region (the constraints on F-feasibility is initially omitted), it naturally attempts to enforce $f_{i,j}$ for all communication $\langle \tau_i, \tau_j \rangle$, which is typically optimal for high rate to low rate communication. For low rate to high rate communication $\langle \tau_i, \tau_j \rangle$, the fact that many of them have only one feasible execution order would lead the algorithm to compute lots of MIXOs that contain only one element (i.e., $U = \{f_{i,j}\}$). The implied constraints of such one-element MIXOs essentially fix the value on the variable $t_{i,j}$. This quickly leads the algorithm to reduce the search space and identify the optimal solution. Such problem-specific

optimization structure appears to be more difficult to exploit in standard ILP.

VIII. Conclusions

In this paper, we study the problem of software synthesis for Simulink models on multicore architectures with partitioned fixed-priority scheduling. We consider a mechanism for semantics preservation on such platforms, that judiciously assigns task offsets and leverages the Simulink RT blocks. This avoids accessing the global shared variables at the same time from the writer and reader on different cores, and enforces a proper execution order between them. We propose to optimize the cost associated to the unit delay RT blocks, and present two approaches. One is a direct ILP formulation, the other is a customized exact procedure. Our evaluation on random systems and on an industrial case study shows that the customized optimization procedure may run several orders of magnitude faster than ILP.

For future work, we plan to include task-to-core allocation into the design space to further improve the solution quality. In addition, the problem can be enhanced by taking into consideration the memory overhead introduced by RT blocks. This may require to impose a constraint on the memory cost from RT blocks, introduced by the limited memory resources on the embedded platform.

Acknowledgments

This paper is partially supported by NSF Grant No. 1739318 and NSFC Grant No. 61471165.

References

[1] N. C. Audsley. On priority asignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, May 2001.

[2] P. Axer et al. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, March 2014.

[3] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[4] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference*, 2011.

[5] P. Caspi and A. Benveniste. Time-robust discrete control over networked loosely time-triggered architectures. In *IEEE Conf. Decision & Control*, 2008.

[6] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–40, 2008.

[7] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to tta: A layered approach for distributed embedded applications. In *Conf. Language, Compiler, and Tool for Embedded Systems*, 2003.

[8] J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *IEEE Conference on Real-Time Computing Systems and Applications*, 1999.

[9] R. 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.

[10] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of multitask implementations of simulink models with minimum delays. *IEEE Trans. Industrial Informatics*, 6(4):637–651, 2010.

[11] M. Di Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Trans. Embedded Computing Systems*, 7(3):23, 2008.

[12] R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *International Workshop on Hardware/Software Codesign*, 1998.

[13] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *High Assurance Systems Engineering Symposium*, 2008.

[14] A. Graillat, M. Moy, P. Raymond, and B. Dupont De Dinechin. Parallel Code Generation of Synchronous Programs for a Many-core Architecture. In *Design, Automation and Test in Europe*, 2018.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[16] Gang Han, Marco Di Natale, Haibo Zeng, Xue Liu, and Wenhua Dou. Optimizing the implementation of real-time simulink models onto distributed automotive architectures. *Journal of Systems Architecture*, 59(10):1115–1127, 2013.

[17] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

[18] M. Di Natale and H. Zeng. Task implementation of synchronous finite state machines. In *Conference on Design, Automation Test in Europe*, 2012.

[19] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[20] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study: From simulink specification to multi/many-core execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[21] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Int. Conf. Application of Concurrency to System Design*, 2004.

[22] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[23] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2013.

[24] W. Puffitsch, E. Noulard, and C. Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

[25] The MathWorks. The mathworks simulink and stateflow user's manuals. web page: http://www.mathworks.com.

[26] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, Oct 2008.

[27] C. Tuncali, G. Fainekos, and Y. Lee. Automatic parallelization of multirate block diagrams of control systems on multicore platforms. *ACM Transactions on Embedded Computing Systems*, 16(1):15, 2016.

[28] H. Zeng and M. Di Natale. Mechanisms for guaranteeing data consistency and flow preservation in autosar software on multi-core platforms. In *IEEE Symposium on Industrial Embedded Systems*, 2011.

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

[30] 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.*, 16(5s):1–21, September 2017.

[31] Y. Zhao and H. Zeng. The virtual deadline based optimization algorithm for priority assignment in fixed-priority scheduling. In *IEEE Real-Time Systems Symposium*, 2017.

Appendix

A. ILP Formulation

In this appendix, we detail an integer linear programming (ILP) formulation of the problem in (10). It is derived by formulating the first two constraints in (10) as (33) and (35) –(39) detailed below.

1) Schedulability Constraints: We first introduce a binary variable $P_{i,j}$ for each pair of blocks τ_i and τ_j *allocated on the same core*, to define their priority order

$$P_{i,j} = \begin{cases} 1, & p_i > p_j \\ 0, & p_j > p_i \end{cases} \tag{32}$$

The priority order shall satisfy the properties of anti-symmetry and transitivity.

$$\begin{aligned} &P_{i,j} + P_{j,i} = 1,\ \forall \tau_i \neq \tau_j, E_i = E_j \\ &P_{i,j} \geq P_{i,k} + P_{k,j} - 1,\ \forall \tau_i \neq \tau_j \neq \tau_k, E_i = E_j = E_k \end{aligned} \tag{33}$$

The response time analysis (3) can then be written as

$$R_i = C_i + \sum_{\forall \tau_j : E_j = E_i} P_{j,i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{34}$$

The product of variable $P_{j,i}$ and the ceiling operator can be linearized by introducing an *integer* variable $\Pi_{j,i}$ subject to the following constraint

$$\Pi_{j,i} \geq \frac{R_i}{T_j} - (1 - P_{j,i})M \tag{35}$$

where $\Pi_{j,i}$ is an integer variable and M is a large enough constant (e.g., T_i/T_j). Intuitively, when τ_i has a higher priority than τ_j (i.e., $P_{j,i} = 0$), the above constraint becomes trivially true. Otherwise, $\Pi_{j,i}$ is enforced to be at least $\frac{R_i}{T_j}$. The response time analysis can then be formulated as the following linear constraint

$$R_i = C_i + \sum_{\forall \tau_j : E_j = E_i} \Pi_{j,i} \cdot C_j \tag{36}$$

The *schedulability* of τ_i requires that it finishes before its deadline

$$O_i + R_i \leq T_i \tag{37}$$

2) Constraints Implied by Execution Orders: We now show how the constraints enforced by the execution orders can be formulated. Four cases are considered, which correspond to Rules 1–4.

Rule 1 and Rule 2 are symmetric to each other. Thus we only demonstrate the formulation for Rule 1 as follows

$$P_{i,j} \geq t_{i,j} \wedge O_j \geq O_i - (1 - t_{i,j})M \tag{38}$$

Intuitively, when $t_{i,j} = 1$, both $P_{i,j} = 1$ and $O_j \geq O_i$ are enforced. In the other case where $t_{i,j} = 0$, both constraints are trivially true.

In a similar manner, Rules 3–4 can be formulated as follows

$$\begin{aligned} &O_j \geq R_i + O_i - (1 - t_{i,j})M \\ &O_i \geq R^{RT}_{i,j} + O_j - (1 - t_{j,i})M \end{aligned} \tag{39}$$

$R^{RT}_{i,j}$ is computed according to (7). It can be formulated in ILP using constraints similar to (35) and (36).

3) Example:

Example 5. We now demonstrate the ILP formulation using the example system depicted in Figure 6. The system consists of two cores each of which contains two blocks. We first look at the response time formulation. Consider τ_0 as example. Its response time R_0 is formulated as follows.

$$\begin{aligned} &R_0 = C_0 + \Pi_{1,0} C_1 \\ &\Pi_{1,0} \geq \frac{R_0}{T_1} - (1 - P_{1,0})M \end{aligned} \tag{40}$$

where R_0 is a non-negative real variable representing the response time of τ_0. $P_{1,0}$ is a binary variable representing the partial priority order between τ_1 and τ_0. $\Pi_{1,0}$ is a non-negative integer variable for linearizing the product of partial priority order variable $P_{1,0}$ and the ceiling term as in (34).

The delay for the output update function between τ_0 and τ_3, namely $R^{RT}_{0,3}$ can similarly be formulated as follows.

$$R^{RT}_{0,3} = C^{RT} + \Pi_{1,0} C_1 \tag{41}$$

Next we look at the execution order implied constraints. Consider intra-communication $\langle \tau_0, \tau_1 \rangle$, $\langle \tau_3, \tau_2 \rangle$, which correspond to four execution orders: $t_{0,1}$, $t_{1,0}$, $t_{3,2}$ and $t_{2,3}$. The corresponding implied constraints are

$$\begin{aligned} &P_{0,1} \geq t_{0,1} \wedge O_1 \geq O_0 - (1 - t_{0,1})M \\ &P_{1,0} \geq t_{1,0} \wedge O_0 \geq O_1 - (1 - t_{1,0})M \\ &P_{3,2} \geq t_{3,2} \wedge O_2 \geq O_3 - (1 - t_{0,1})M \\ &P_{2,3} \geq t_{2,3} \wedge O_3 \geq O_2 - (1 - t_{2,3})M \end{aligned} \tag{42}$$

Similarly, for inter-communication $\langle \tau_0, \tau_3 \rangle$ and $\langle \tau_1, \tau_2 \rangle$. The corresponding implied constraints are

$$\begin{aligned} &O_3 \geq R_0 + O_0 - (1 - t_{0,3})M \\ &O_0 \geq R^{RT}_{0,3} + O_3 - (1 - t_{3,0})M \\ &O_2 \geq R_1 + O_1 - (1 - t_{1,2})M \\ &O_1 \geq R^{RT}_{1,2} + O_2 - (1 - t_{2,1})M \end{aligned} \tag{43}$$

Finally, execution order shall satisfy anti-symmetry and transitivity constraints, which amounts to the following

$$\begin{aligned} &t_{0,1} + t_{1,0} = 1 \\ &t_{3,2} + t_{2,3} = 1 \\ &t_{0,3} + t_{3,0} = 1 \\ &t_{1,2} + t_{2,1} = 1 \end{aligned} \tag{44}$$

Transitivity constraints are trivial to consider for the example here. Consider enforcing execution order $t_{0,1}$ and $t_{1,2}$, which seems to imply that $t_{0,2}$ is also be enforced. However, since the example does not have communication $\langle \tau_0, \tau_2 \rangle$, it is not necessary to consider.

The objective of the optimization is to minimize the use of unit delay RT block, which can be formulated as follows.

$$\min t_{1,0} + t_{2,3} + t_{3,0} + t_{2,1} \tag{45}$$

The final ILP formulation consists of constraints in the form of (40) for all tasks, constraints in the form of (41) for $R^{RT}_{0,3}$ and $R^{RT}_{1,2}$, (42), (43), (44), and the objective (45) .