Scheduling Real-time HiL Co-simulation of Cyber-Physical Systems on Multi-core Architectures

Salah Eddine Saidi
IFP Energies nouvelles
Rueil-Malmaison, France
salah-eddine.saidi@ifpen.fr

Nicolas Pernet
IFP Energies nouvelles
Rueil-Malmaison, France
nicolas.pernet@ifpen.fr

Yves Sorel
Inria
Paris, France
yves.sorel@inria.fr

***Abstract*—When designing complex cyber-physical systems, engineers have to integrate numerical models from different modeling environments in order to simulate the whole system and estimate its global performances. Co-simulation refers to such joint simulation of heterogeneous models. If some parts of the system are physically available, it is possible to connect these parts to the co-simulation in a Hardware-in-the-Loop (HiL) approach. In this case, the simulation has to be performed in real-time where models execution consists in periodically reacting to the real (physically available) components and providing periodic output updates. This paper deals with the parallelization and scheduling of real-time Hardware-in-the-Loop co-simulation of numerical models on multi-core architectures. A method for defining real-time constraints that have to be met is proposed. Also, an ILP formulation as well as a heuristic are proposed to solve the problem of scheduling the co-simulation on a multi-core architecture while satisfying the previously defined real-time constraints. The proposed approach is evaluated for different sizes of co-simulations and multi-core processors.**

***Index Terms*—co-simulation, HiL, scheduling, real-time, multi-core**

I. Introduction

Systems that combine computational elements and physical processes are referred to as Cyber-Physical Systems (CPS) [15]. The different elements are more or less tightly coupled and interact with each other using different communication media. The diversity of the involved disciplines makes the process of building CPS challenging, costly and time consuming. Enabling the prediction of the system's behavior before its deployment has the potential to reduce the risks, the cost and the needed effort. Simulation is an efficient way to achieve these requirements as it allows imitating the functioning of the system on a computer and assessing its design.

In co-simulation, the different subsystems are modeled in a segregated manner and then connected together to simulate the whole system on a computer. Integrating heterogeneous models usually results in computationally expensive co-simulation. Increasing CPU frequency by means of silicon integration has reached its possible limits and semiconductor manufacturers switched to building multi-core processors, allowing parallel processing on a single computer.

In Hardware-in-the-Loop (HiL) co-simulation, real components, that are physically available, are connected to simulated models. The goal here is to emulate the behavior of a part of the system using the simulated models in order to run the real component under realistic conditions. In HiL, two concepts of time have to be correctly meshed: the simulated time and the real time. Achieving a correct meshing of the simulated time and the real time defines a set of timing constraints imposed on the simulated models. Performing HiL co-simulations on multi-core processors can enhance the opportunities of satisfying these timing constraints which may be infeasible on single-core processors.

In the present paper, we focus on the execution of HiL co-simulation under real-time constraints on multi-core processors. We are interested in co-simulations of CPS that are compliant with the Functional Mock-up Interface (FMI) standard [10] that is widely adopted in industry. The Refined CO-SIMulation (RCOSIM) [2] is a co-simulation parallelization approach which takes advantage of the information given by FMI about input-output relationships inside a model that is exported as a Functional Mock-up Unit (FMU). In our recent work [20], we extended the use of RCOSIM to multi-rate co-simulation. In this paper, we extend the use of RCOSIM to HiL co-simulation under real-time constraints. First, we propose rules to propagate real-time constraints to FMI compliant models in a HiL co-simulation. Furthermore, we propose non preemptive multi-core scheduling algorithms to satisfy the timing constraints. These algorithms consist in an exact algorithm based on an Integer Linear Programming (ILP) to find the optimal solution and a heuristic giving an approximate solution.

The rest of the paper is organized as follows. The next section is dedicated to related work. In Section III, we present basic concepts and preliminaries related to FMI HiL co-simulation under real-time constraints and give a description of the problem. In Section IV, we define the real-time constraints that are imposed in a HiL co-simulation. Then, we propose rules and algorithms to propagate these constraints in the dependence graph in Section V. Next, we present in Section VI an ILP formulation and a list heuristic to solve the problem of non preemptive multi-core scheduling of HiL co-simulation under real-time constraints. We evaluate our solution in Section VII and conclude the paper in Section VIII.

II. Related Work

Some work has been carried out in academia as well as in industry in order to tackle the problem of real-time

HiL co-simulation. A first aspect of our contribution is to assign HiL real-time constraints to the different operations of the simulated part. In industry, this problem is tackled at the model or submodel level. Engineers have to define the different periods of the different models or submodels. Such approach has two drawbacks. First, it often leads the user to naively translate a model-rate (stepsize) requirement into a real-time period requirement. This results in more stringent period constraints than necessary. Second, to find an execution order, it is often necessary to break some data dependencies by adding a delay or memory function. Indeed, at the model or submodel level, cycles may be present but disappear at lower levels. Figures 1 and 2 give an example of such co-simulation. In [9], a proposed set of propagation rules allows fixing the problem of assigning an adequate period and deadline to each model. Nevertheless, Faure et al. address the problem at the model-level even if they distinguish between models where no output directly depends on the inputs and models where at least one output does. In this approach, the cycle problem is not solved. On the opposite, our approach, by exploiting FMI features, is able to use internal input-output relationships of each model to propagate constraints on a set of models without having to break any data dependency.

After propagating the real-time constraints, our approach deals with a real-time multi-core DAG scheduling problem that may look close to the multi-rate synchronous language scheduling problem which is addressed in [17]. Nevertheless, such approach implies strict periodicity which is not a constraint in the co-simulations that we target. In [5] and [21], online approaches are proposed for real-time DAG scheduling, but on mono-processor platforms. On the other hand, multi-core real-time scheduling of independent tasks has received a lot of attention in the literature [8]. In our work, operations may have deadlines greater than their respective periods and data dependencies have to be considered as constraints. In [18] and [19], multiprocessor real-time scheduling with dependencies is addressed. However, the proposed approaches are preemptive. We adopt an offline non preemptive approach since preemption seems not beneficial given that most industrial applications exhibit fine granularity with short computation times [13]. Also, online scheduling is likely to bring no significant advantages due to the short computation times. We are mainly lead by our experience in industrial co-simulation. Most of the FMI operations that we have to schedule have short computation times (few μs) and an industrial use case may easily lead to a number of operations between 100 and 10000.

III. FMI HiL Co-simulation

In this Section, we present basic concepts related to FMI co-simulation with a focus on HiL co-simulation. First, we give an overview of the FMI standard. Then, we describe briefly our method for transforming a model graph of an FMI co-simulation into a dependence graph. The resulting graph has a finer granularity than the model graph. Finally, we give some preliminaries regarding HiL co-simulation.

A. FMI Standard

The Functional Mock-up Interface (FMI) is a tool-independent and open standard designed within the MODELISAR project[1] and is currently developed and maintained by the Modelica Association[2] which promotes the Modelica language. The FMI standard was developed in order to facilitate the co-simulation of dynamical systems, such as CPS. It provides specifications in order to enable the exchange and the co-simulation of heterogeneous dynamical models that may be developed by different tools and originating from different parties. A modeling tool that supports FMI can export a model as a Functional Mock-up Unit (FMU) which can be used in co-simulation environments. An FMU is a package that encapsulates an XML file containing, among other data, the definitions of the model's variables, and a library defining the equations of the model as C functions. FMI defines interfaces for the involved models to allow their co-simulation.

B. Dependence Graph of an FMI Co-simulation

The method for automatic parallelization of FMU co-simulation that we propose in this paper is based on representing the co-simulation by a dependence graph. We present below how this graph is constructed and a set of attributes that characterize it. The graph construction and characterization method is done following the RCOSIM approach as presented in [2].

The entry point for the construction of a dependence graph of an FMU co-simulation is a user-specified set of interconnected FMUs as depicted in Figure 1. In this Figure, each model has a number of inputs, outputs and, even if not displayed, one state. The execution of each FMU is seen as computing a set of inputs, a set of outputs, and the state of the FMU. A computation of an input, output, or the state is performed by FMU C function calls. Thanks to FMI, it is additionally possible to access information about the internal structure of a model encapsulated in an FMU. In particular, as shown in Figure 2, FMI allows the identification of Direct Feedthrough (e.g. Y_{B1}) and Non Direct Feedthrough (e.g. Y_{A1}) outputs of an FMU and other information depending on the version of the standard.

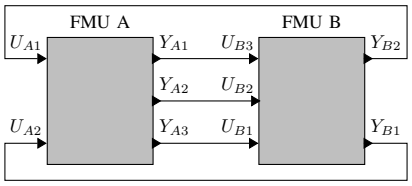

Fig. 1: Inter-FMU dependence specified by the user

The information provided by FMI on input-output dependence allows transforming the FMU graph into a graph with an increased granularity. For each FMU, the inputs, outputs, and the state are transformed into *operations*. An input, output, or a

[1] itea3.org/project/modelisar.html
[2] www.modelica.org/association/

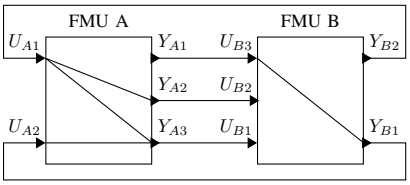

Fig. 2: Intra-FMU dependence provided by FMI

state operation is defined as the set of FMU function calls that are used to compute the corresponding input, output, or state respectively. The co-simulation is described by a dependence graph $G(V, A)$, called the *operation graph*, where each vertex $o_i \in V : 0 \leq i < n$ represents one operation, each arc $(o_i, o_j) \in A : 0 \leq i, j < n$ represents a *dependence* between operations o_i and o_j which means that o_i must be executed before o_j and, possibly, produces data to be consumed by o_j. $n = |V|$ is the size of the operation graph. The operation graph is built by exploring the relations between the FMUs and between the operations of the same FMU. A vertex is created for each operation and arcs are then added between vertices if a dependence exists between the corresponding operations.

The co-simulation of a system is run over a user-specified time interval. More specifically, each model is simulated over a certain number of time steps. At each step, the corresponding FMU inputs, outputs, and state are updated. An execution of the obtained operation graph corresponds to one simulation step. Therefore, running the co-simulation consists in repeatedly executing the graph until the desired number of steps is reached. A new execution of the graph cannot be started unless the previous one was totally finished. The operation graph corresponding to the FMUs of Figure 2 is shown in Figure 3 where input and output operations are represented by circle vertices and state operations are represented by square vertices. Note that state operations do not have successors. Although they do not appear in Figures 1 and 2, we recall that every FMU has one state operation.

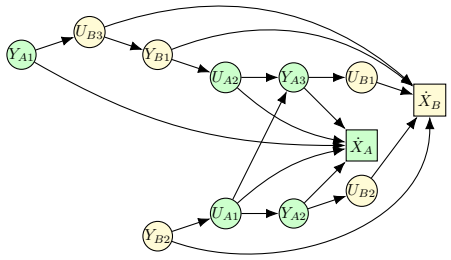

Fig. 3: Operation graph obtained from the FMUs of Figure 2

At each repetition of the execution of the operation graph, the equations of each FMU are integrated according to an integration step assigned by the user and which, in some cases, is driven by the dynamics of the modeled system and the control objective. In addition, each FMU exchanges data with the other FMUs according to a communication step also assigned by the user and which can be equal or larger than its integration step. In a *multi-rate* co-simulation, multiple FMUs are assigned different communication steps [20].

We use the notation $f_m(o_i)$ to refer to the FMU to which the operation o_i belongs. Once the operation graph is constructed, each operation o_i is characterized by a set of parameters. The first parameter consists in the Worst Case Execution Time (WCET) $C(o_i)$ obtained through a profiling phase. Each operation o_i is characterized by a communication step $H(o_i)$ which is equal to the communication step of its FMU. In addition, the following parameters are defined for each operation o_i:

- The set of successors (resp. predecessors) $succ(o_i)$ (resp. $pred(o_i)$).
- The earliest start time from the graph beginning $S(o_i) = \max_{o_j \in pred(o_i)}(E(o_j))$ $(0\ if\ pred(o_i) = \emptyset)$.
- The earliest end time from the graph beginning $E(o_i) = S(o_i) + C(o_i)$.
- The latest end time from the graph end $\overline{E}(o_i) = \max_{o_j \in succ(o_i)}(\overline{S}(o_j))$ $(0\ if\ succ(o_i) = \emptyset)$.
- The latest start time from the graph end $\overline{S}(o_i) = \overline{E}(o_i) + C(o_i)$.
- The flexibility $F(o_i) = R - S(o_i) - C(o_i) - \overline{E}(o_i)$.

$CP = \max_{o_i \in V_I}(E(o_i))$ denotes the critical path of the graph.

These notations are used afterward in the proposed scheduling algorithms.

In the case of multi-rate co-simulation, a multi-rate transformation algorithm is applied on the the operation graph. Its principle consists in repeating each operation o_i in the operation graph a certain number of times where each repetition is called an occurrence, assigning to it a repetition factor (number of occurrences) denoted $r(o_i)$, and adding arcs between these occurrences. More details can be found in our previous work [20].

C. Preliminaries on HiL Co-simulation

A HiL setup is composed of a simulated component and a physically available component that is interfaced with the simulated component via inputs and outputs. It should be noted that multiple parts may be physically available and involved in HiL, e.g. multiple controllers interacting with multiple parts of simulated physical processes. We regard all these physically available parts as a single real component. Figure 4 shows a basic example of a HiL co-simulation. The simulated component is represented by an operation graph. It consists of two FMUs A and B. The real component has one input (resp. output) connected to an output (resp. input) operation of the simulated component. We refer to output and input operations that are directly connected with the real component as *gate operations*.

The main goal of HiL co-simulation is to estimate the performance of the real component by providing a realistic environment through the simulated component. The simulated component has to interact with the real component at the same rate as its real counterpart. As such, the inputs and outputs of the real component, which we assume to be periodically

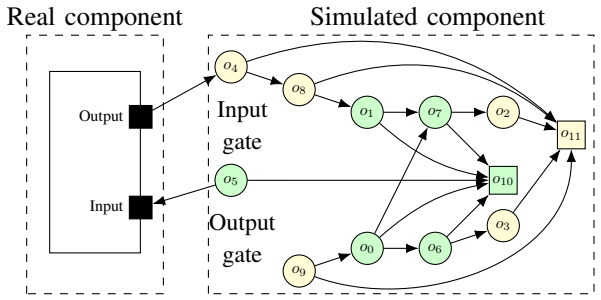

Fig. 4: Example of co-simulation under real-time constraints.

sampled, define real-time constraints which involve that the simulated time has to match the real time. These constraints are imposed on the outputs and inputs of the simulated component that are connected with the inputs (resp. outputs) of the real component.

The ultimate goal of this work is to perform real-time multi-core scheduling of operation graphs representing HiL co-simulations. In order to achieve this, we deal with a problem that is essentially composed of two parts. First, we need to characterize all the operations of the operation graph with real-time parameters. Starting from the constraints that are imposed on the gate operations, we seek to propagate these constraints to the rest of the operations. Second, we aim at proposing a real-time scheduling algorithm that allows the execution of the operations, characterized as described above, on a multi-core processor in such a way that the propagated real-time constraints are satisfied.

IV. Definition of Real-time Constraints

The propagation of real-time constraints that we propose avoids over-constraining the co-simulation compared to classical real-time co-simulation [1] where a unique real-time constraint inherited from the real component is imposed on all the tasks of the simulated component. On the other hand, in our approach, we only impose real-time constraints on gate operations of the simulated component and we define constraints for the rest of the operations accordingly, i.e. in such a way that the constraints imposed on the gate operations can be satisfied.

We assume that the sampling period of a given input (resp. output) of the real component is a multiple of the communication step size of the operation of the simulated component that is connected with it.

Let the inputs and the outputs of the real component be sampled with sampling periods T_x and T_y respectively where x and y denote the indices of the corresponding input and output respectively. In other words, an input (resp. output) of the real component is periodically activated every T_x (resp. T_y) units of time. The sampling periods of the different inputs and outputs of the real component can be identical or different.

The periodic activation of an output of the real component leads to the production of data that are consumed by an input gate operation o_i of the simulated component. Therefore, at the z^{th} sample $z \times T_y$, the input gate operation o_i consumes new data corresponding to simulated time $z \times T_y$. This defines a periodic release for this input gate operation, i.e. time instants at which the data to be consumed by o_i is periodically updated by the real component. A release constraint imposed on a gate operation is defined by its period $R(o_i) = T_y$.

Similarly, the periodic activation of an input of the real component requires data produced by an output gate operation o_j of the simulated component to be available. Therefore, before the w^{th} sample $w \times T_x$, the output gate operation o_j has to produce data corresponding to simulated time $w \times T_x$. This defines a periodic deadline for this output gate operation, i.e. it has to periodically produce the data to be consumed by the real component before specific periodic instants. A deadline constraint imposed on an output gate operation is defined by its period $D(o_j) = T_x$.

V. Propagation of Real-time Constraints

The aforementioned definitions specify real-time constraints for gate operations. These operations being dependent on other operations and vice versa, it becomes necessary to define the impact of the real-time constraints on the rest of the operations.

A. Propagation of Release Constraints

Let's consider an operation graph representing a multi-rate co-simulation. Let a release constraint of period T_y be applied on an input gate operation $o_i \in V$. Given that the sampling period of this release constraint T_y is greater or equal to the communication step size of $o_i, H(o_i)$, only a subset of the occurrences of o_i, $o_i^p, 0 \leq p < r(o_i)$ that belong to the operation graph are subject to this release constraint. We recall that $r(o_i)$ is the repetition factor of operation o_i which is equal to the number of its occurrences. The z^{th} occurrence of the release constraint $z \times T_y = t_k$ is applied on the occurrence o_i^{p,t_k}, i.e. the occurrence p of operation o_i executed at time step t_k, where $0 \leq p < r(o_i)$ and $t_k \in \mathbb{R}^+$. This subset of occurrences can be determined as follows. The occurrence $z \times T_y$ of the release constraint is applied on occurrence $o_i^p : p = z \times \frac{T_y}{H(o_i)}$ where $H(o_i)$ is the communication step of o_i. Therefore o_i^p is assigned a release $R(o_i^p) = z \times T_y$. Once o_i^p is released and executed, the operations that depend on o_i^p can be released. Therefore, the release constraint $R(o_i^p)$ is propagated towards all the successors of o_i^p. This propagation is given by expression $\forall o_{i'} \in succ(o_i) : R(o_{i'}) = R(o_i)$.

Let's consider, for example, the HiL co-simulation shown in Figure 4. Let $H_A = 2$ and $H_B = 4$ be the communication step sizes of FMUs A and B respectively and let the sampling period of the output of the real component be $T_{out} = 4$. Figure 5 shows the propagation of the release constraint in the operation graph. Note that the multi-rate transformation algorithm is applied before the propagation. The operations that are assigned a release are colored. The release assigned to each operation is shown below or above the corresponding operation. The dashed blue arrow indicates the direction of the propagation which is from a predecessor to a successor. The propagation of the release constraint is performed iteratively.

Starting from the input gate operation o_4^p, for each operation that is assigned a release constraint, this constraint is propagated towards all its successors as described above. The values of release that are placed below or above some operations, correspond to the first occurrence of the release constraint $z \times T_{out} = 0 \times 4 = 0$. In order to find the occurrence of the input gate operation o_4 that is subject to this occurrence of the release constraint, we use the formula given above: $p = z \times \frac{T_{out}}{H(o_4)} = 0 \times \frac{4}{4} = 0$. Therefore, in Figure 5, $p = 0$.

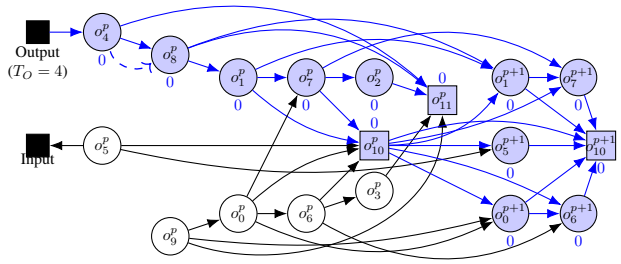

Fig. 5: Example of release propagation.

B. Propagation of Deadline Constraints

Now let a deadline constraint of period T_x be applied on an output gate operation o_j of the operation graph. Given that the sampling period of this deadline constraint T_x is greater or equal to the communication step size of $o_j, H(o_j)$, only a subset of the occurrences of o_j, $o_j^q : 0 \leq q < r(o_j)$ that belong to the operation graph are subject to this deadline constraint. The w^{th} occurrence of the deadline constraint $w \times T_x = t_k$ is applied on the occurrence o_j^{q,t_k}, i.e. the occurrence q of operation o_j executed at time step t_k, where $0 \leq q < r(o_j)$ and $t_k \in \mathbb{R}^+$. This subset of occurrences can be determined as follows. The occurrence $w \times T_x = t_k$ of the deadline constraint is applied on occurrence $o_j^q : \ q = w \times \frac{T_x}{H(o_j)}$ where $H(o_j)$ is the communication step of o_j. Therefore, o_j^q that is assigned a deadline $D(o_j^q) = w \times \frac{T_x}{H(o_j)}$. This constraint is, then, propagated towards all the predecessors of o_j^q. The propagation of a deadline constraint is given by expression $\forall o_{j'} \in pred(o_j) : D(o_{j'}) = D(o_j)$.

Figure 6 illustrates the propagation of the deadline constraint in the operation graph. The sampling period of the input of the real component is $T_{in} = 4$. We recall that the execution of the co-simulation consists in running the operation graph repeatedly. Therefore, each run corresponds to a pattern that involves specific occurrences of the operations. While in the pattern of the operation graph that is shown previously, operation o_5^p does not have a predecessor, the state operation o_{10}^{p-1} belonging to the preceding pattern is a predecessor of o_5^p. The operations that are assigned a deadline are colored. The deadline assigned to each operation is shown above the corresponding operation. The dashed red arrow indicates the direction of the propagation which is from a successor to a predecessor. The propagation of the deadline constraint is performed iteratively. Starting from the output gate operation o_5^q, for each operation that is assigned a deadline constraint, this constraint is propagated towards all its predecessors. The values of deadline that are shown correspond to the second occurrence of the release constraint $w \times T_{in} = 1 \times 4 = 4$. In order to find the occurrence of the output gate operation o_5 that is subject to this occurrence of the release constraint, we use the formula given previously: $q = w \times \frac{T_{in}}{H(o_5)} = 1 \times \frac{4}{2} = 2$. Therefore, in Figure 6, $q = 2$.

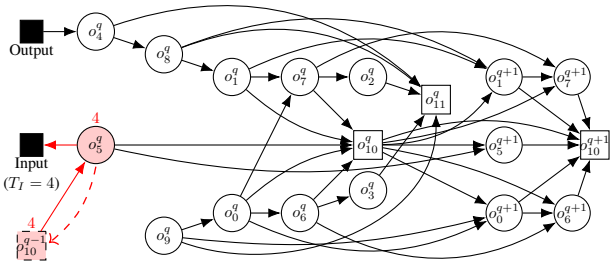

Fig. 6: Example of deadline propagation.

C. Propagation of real-time constraints towards unreached operations

It can be seen that the propagation of the real-time constraints is based on graph traversal. The Breadth First Search (BFS) [16] or the Depth First Search (DFS) graph traversal algorithms [6] can be used to perform this propagation. On the one hand, the release constraints are propagated following the topological ordering of the graph since the constraint is always propagated from a predecessor to a successor. We refer to such propagation as *forward propagation*. On the other hand, the deadline constraint is propagated in reverse topological ordering of the graph because the constraint is always propagated from a successor to a predecessor. We refer to such propagation as *backward propagation*. This means that the release (resp. deadline) constraint cannot propagate towards the operations that come before (resp. after) the input (resp. output) gate operation that is subject to this constraint.

Below, we propose a method to assign release and deadline constraints to operations that are not reached in the forward and backward propagation phases respectively due to the nonexistence of a path starting at a gate operation (resp. these operations) and leading to these operations (resp. a gate operation). This method is based on *looping* the propagation. In other words, a release (resp. deadline) constraint is propagated in a reverse order of the forward (resp. backward) propagation phase. Such looping is possible because the operation graph is repeated periodically in runtime.

We define the *hyperperiod* as the least common multiple of the real-time sampling periods of all inputs and outputs of the real component and the communication step sizes of all operations of the simulated component: $HP = lcm(H(o_1), H(o_2), \ldots, H(o_n), T_1, T_2, \ldots, T_m)$ where n is the number of operations in the operation graph and m is the number of inputs and outputs of the real component.

The hyperperiod notion resembles the notions of hyperperiod found in the real-time literature. More specifically, it specifies a time interval for describing a periodic pattern of the real-time constraints assigned to operations. It only differs in that it combines real-time periods and communication step sizes. In the context of co-simulation under real-time constraints, we apply the multi-rate transformation algorithm over the hyperperiod. Therefore, the repetition factor of each operation $o_i \in V$ is $r(o_i) = \frac{HP}{H(o_i)}$. In the example shown in Figure 7, we have only one real time constraint of period 4 and, therefore, the usual hyperperiod and the one proposed above are equal (4). Now, assume that FMU A has a communication step size of 3, the usual hyperperiod remains 4 whereas the proposed hyperperiod becomes 12.

Given the periodic repetition of the operation graph, a release constraint can be written as follows:

$$\forall o_i^p, o_i^{p'} : p' = p - \frac{HP}{H(o_i)},\ R(o_i^{p'}) = R(o_i^p) - HP \qquad (1)$$

Equation 1 expresses how a release constraint imposed on a given operation o_i, is repeated periodically over the occurrences of o_i. In other words, if we know the value of the release constraint imposed on occurrence o_i^p, we can determine the value p' such as $R(o_i^{p'}) = R(o_i^p) - HP$.

Similarly, a deadline constraint can be written as follows:

$$\forall o_j^q, o_j^{q'} : q' = q + \frac{HP}{H(o_j)},\ D(o_j^{q'}) = D(o_j^q) + HP \qquad (2)$$

Let's consider that a release constraint is propagated in the operation graph $G(V, A)$ starting from the gate operation o_i. Also, consider that the multi-rate transformation algorithm has been applied on the operation graph over the hyperperiod HP. Then, $o_{i'}^p$, the last occurrence of the state operation $o_{i'} : f_m(o_{i'}^p) = f_m(o_i)$ that is assigned a release constraint during the forward propagation phase corresponds to the time step that is equal to the hyperperiod, i.e. it can be written $o_{i'}^{p,HP}$. Note that such operation has no successor in the initial operation graph as well as the one obtained using the multi-rate transformation. In order to propagate the release constraint to the operations that come before o_i in the graph, we loop back to the occurrence $o_{i'}^{p'}$ of the state operation $o_{i'}$ by performing a negative shift of the release constraint whose length is equal to the hyperperiod. The occurrence $o_{i'}^{p'}$ is a predecessor of the first occurrence of the gate operation o_i that appear in the periodic pattern of the operation graph. This is done for every FMU, for which the state operation is assigned a release. An example is shown in Figure 7.

Afterward, starting from the operations that are newly assigned release constraints, a new forward propagation phase is applied.

Now consider that a deadline constraint is propagated in the graph $G(V, A)$ starting from the gate operation o_j. Let $o_{j'}^{q'}$ be the occurrence of the state operation $o_{j'} : f_m(o_{j'}^p) = f_m(o_j)$ that is a predecessor of the first occurrence of the gate

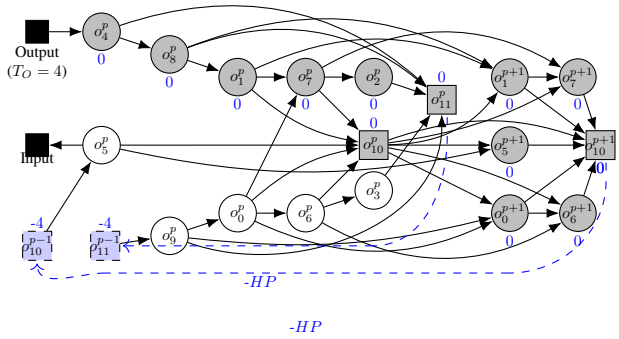

Fig. 7: Release back loop propagation phase.

operation o_j that appear in the periodic pattern of the operation graph. Also, let $o_{j'}^q$ be the last occurrence of the state operation $o_{j'}$ that appear in the periodic pattern of the operation graph. Like for the release constraint, the operations that come after o_j in the operation graph can be assigned deadline constraints by looping forward the deadline constraint that is assigned to $o_{j'}^{q'}$ towards $o_{j'}^q$. In other words, a positive shift equal to the length of the hyperperiod is applied by looping forward to the last occurrence of the state operation $o_{j'}^q$. This is performed for every FMU whose state operation was assigned a deadline date. Figure 8 shows an example of such looping.

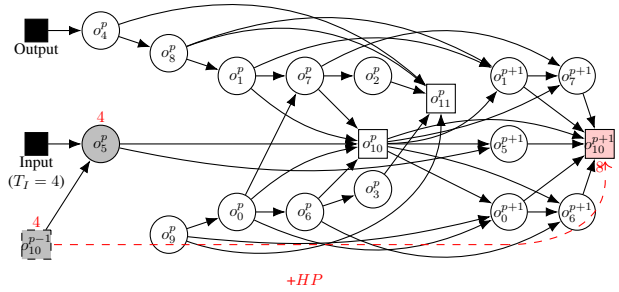

Fig. 8: Deadline forward loop propagation phase.

A backward propagation phase, starting from the operations that are newly assigned deadline constraints, is then applied.

D. Propagation of Multiple Real-time Constraints

So far, we considered that the operation graph is subject to real-time constraints that are equal. In industrial applications, this is not always the case. Now, let's consider that multiple input and output gate operations are subject to different release and deadline constraints respectively. This means the sampling periods of the inputs and the outputs of the real component are different. As before, we consider that every sampling period is a multiple of the communication step size of the gate operation that is applied to.

In order to propagate multiple real-time constraints, we follow the propagation process described previously. The release constraints are propagated by iteratively running forward propagation and back loop propagation phases whereas deadline

constraints are propagated by iteratively running backward propagation and forward loop propagation phases. The main difference here is that during propagation, several constraints may be applied to the same operation.

Gate operations are always subject to only one constraint because they are directly assigned the constraints imposed by the real component. Therefore, assigning constraints to gate operations remains unchanged. On the other hand, the rest of the operations of the operation graph may have several predecessors and successors. In fact, if an operation $o_i \in V$ has no more than one predecessor, i.e. $|pred(o_i)| \leq 1$, the propagation of a release constraint towards this operation remains unchanged. Similarly, if an operation $o_j \in V$ has no more than one successor, i.e. $|succ(o_j)| \leq 1$, the propagation of a deadline constraint towards this operation remains unchanged. Hence, we are interested here in propagating release (resp. deadline) constraints towards operations which have more than one predecessor (resp. successor), i.e. $|pred(o_i)| > 1$ (resp. $|succ(o_j)| > 1$).

We denote by $\mathcal{R}(o_i)$ the set of release constraints propagated towards the operation o_i and by $\mathcal{D}(o_j)$ the set of deadline constraints propagated towards the operation o_j. The set $\mathcal{R}(o_i)$ is built as the union of all the release constraints propagated towards o_i from its predecessors, i.e. $\mathcal{R}(o_i) = \{R(o_{i'}) : o_{i'} \in pred(o_i)\}$. Each operation o_i must be assigned only one release constraint which is chosen from the set $\mathcal{R}(o_i)$. A release constraint specifies the earliest date the associated operation can start its execution. As such, the most constraining release constraint has to be chosen from the set $\mathcal{R}(o_i)$: $\forall o_i \in V, R(o_i) = \max_{R(o_{i'}) \in \mathcal{R}(o_i)}(R(o_{i'}))$.

In the same way, the set $\mathcal{D}(o_j)$ is built as the union of all the deadline constraints propagated towards o_j from its successors, i.e. $D(o_{j'}) : o_{j'} \in succ(o_j)$. Each operation must be assigned only one deadline constraint which is chosen from the set $\mathcal{D}(o_j)$. A deadline constraint specifies the latest date by which the associated operation must finish its execution. Therefore, the most constraining deadline constraint is selected from the set $\mathcal{D}(o_j)$: $\forall o_j \in V, D(o_j) = \min_{D(o_{j'}) \in \mathcal{D}(o_j)}(D(o_{j'}))$

VI. Multi-core Scheduling of Real-time Co-simulation

In this section, we are interested in multi-core scheduling of FMU co-simulation under real-time constraints. We consider that the real-time constraints that are imposed by the real component have been propagated through the operation graph as described in Section V. Therefore, the aim here consists in scheduling the operations of the operations graph on a multi-core processor, such that these constraints are satisfied. Rather than an online scheduling algorithm, we use an offline scheduling approache which we consider to be more suitable given the fine granularity of the operations, and since information about the execution times of the operations and the dependence between them is available before runtime. In addition, we adopt a non preemptive scheduling approach. In fact, most of the operations have short execution times which limits the benefits of allowing preemption. Hereafter, we detail two important points necessary for performing real-time multi-core scheduling. Afterward, we propose an ILP formulation and a heuristic for offline scheduling of operation graphs under real-time constraints.

A. Accounting for Dependence in Real-time Scheduling

The model of computation for (co-)simulation is close to the synchronous paradigm [3], [4]. In this paradigm, a program evolves according to a sequence of ticks of logical time at which computations are considered to produce their results instantaneously. In co-simulation, the repetitions of the operation graph correspond to the ticks of the logical time. It means that the processor on which the graph will be executed is not taken into consideration. The propagation of the release and deadline constraints presented in Section V follows this model of computation. However, when real-time constraints are involved, they have to be taken into account since the processor and communication channels are known. In fact, each operation takes a certain execution time to run and, therefore, cannot produce the result instantaneously. In order to proceed to scheduling the operation graph, it is necessary to account for the dependence between operations.

We adopt an approach similar to the one proposed in [5] to modify the release and deadline dates assigned to each operation in order to account for execution times.

Let o_i and o_j be two operations such that $o_j \in pred(o_i)$. For a given schedule of the operation graph to be valid, the relations $S(o_i) \geq R(o_i)$ and $S(o_i) \geq E(o_j)$ must be satisfied. Therefore, a new release date for o_i can be computed as follows: $R(o_i) = \max(R(o_i), \max_{o_j \in pred(o_i)}(E(o_j)))$.

Consider now two operations o_i and o_j such that $o_j \in succ(o_i)$. For the operation graph to be schedulable, the relations $E(o_i) \leq D(o_i)$ and $E(o_i) \leq D(o_j) - C(o_j)$ must be satisfied. In fact, $D(o_j) - C(o_j)$ represents the latest time to start the execution of the successor o_j such that its deadline can be met. Therefore, a new deadline date of o_i can be computed as follows: $D(o_i) = \min(D(o_i), \min_{o_j \in succ(o_i)}(D(o_j) - C(o_j)))$.

B. Scheduling Interval

In offline scheduling, the schedule is computed over an interval of time. This schedule is then executed repetitively. For co-simulation under real-time constraints, a natural approach is to apply techniques that are used for classical real-time systems (such co-simulation is considered a real-time system after all). For this, we need first to represent the operation graph with a model that involves the parameters that are usually used for classical real-time systems. In particular, we need to define a relative deadline and a period for each operation. Note that so far, we have only spoken about sampling periods of data exchange between the real and the simulated component. Although related to the sampling periods, the periods that we seek to define here for each operation are different and correspond to task periods that are found in classical real-time systems. We propose to handle this requirement as follows. We consider that every operation that appears in the hyperperiod

pattern of the operation graph is a distinct operation. In other words, occurrences of one operation are not regarded as repetitions of a single operation. Therefore, we consider that the operation graph is *mono-period*, i.e. all the operations have the same period. The value of this period is equal to the hyperperiod (see Section V-C). The relative deadline of each operation can then be defined as the duration between its release and deadline.

In the real-time literature, we find contributions regarding the schedule interval targeting different kinds of real-time tasks, schedulers, and processors [12]. To the best of our knowledge, the existing results are either not applicable (e.g. constrained deadlines) to our problem or propose bounds that are intractable, i.e. as the size of the operation graph grows and depending on the parameters of the operations, they result in very large schedule intervals and we cannot guarantee to compute the schedule within an acceptable time. The length of the schedule interval for co-simulation under real-time constraints cannot be chosen in a straightforward manner to be equal to the hyperperiod. This is because the operation graph features *arbitrary deadlines*, i.e. relative deadlines that are greater than the periods. Indeed, some operations may have relative deadlines that exceed the hyperperiod which we consider to be the period of every operation. For instance, in Figure 8, operation o_{10}^{p+1} has a relative deadline of 8 while the hyperperiod is equal to 4. Arbitrary deadlines may lead to *hyperperiod spill* [7]. The latter refers to operations that are not scheduled in their hyperperiod and spill over the next one.

We propose to handle this situation as follows. We start with a schedule interval whose length is equal to the hyperperiod and iteratively increase it. If no hypeperiod occurs we consider that the operation graph is schedulable. If an operation misses its deadline, the operation graph is not schedulable and we stop the process. Otherwise, this process is stopped after being repeated for a certain number of times set by the user. In this case, we consider that the scheduling algorithm failed to find a schedule. Note that determining a non pessimistic schedulability interval length remains an open problem.

As shown in Section V, the propagation of the real-time constraints in an operation graph may lead to assigning negative release dates to some operations. This means that such operations can be considered to belong to the previous iteration of the operation graph.

C. ILP Formulation

In order to find the optimal solution of the real-time multi-core scheduling problem, we need to use an exact algorithm. We have chosen to use the ILP approach. Our ILP formulation for multi-core scheduling of co-simulation under real-time constraints is given below.

We define the decision binary variables x_{ik} which indicates whether the operation o_i is allocated to core p_k or not. Expression 3 states the constraint that each operation has to be allocated to one and only one core. The end time of each operation o_i is computed using the expression 4. The start date of every operation must be at the earliest equal to its release date. Expression 5 captures this constraint. The deadline date of every operation is the latest time before which the operation has to finish its execution. Expression 6 specifies this constraint.

For operations that are allocated to the same core and that are completely independent, i.e. no path exists between them, we have to ensure that they are executed in non overlapping time intervals. Expressions 7 and 8 capture this constraint where b_{ij} is a binary variable that is set to one if o_i is executed before o_j and M is a large positive constant. Therefore, expression 7 ensures the case where o_i is executed before o_j and expression 8 ensures the case where where o_j is executed before o_i

For dependent operations that are allocated to different cores, synchronization mechanism is necessary to ensure the order of their execution. While the computed schedule defines an order of execution for dependent operations, variations of execution times may occur in runtime leading to changing the start and end times of the operations. Therefore, we use synchronization to ensure the order of execution. The cost of synchronization is taken into account as follows. A synchronization cost is introduced in the computation of the start time of an operation o_j, if it has a predecessor o_i that is allocated to a different core and if its start time is the earliest among the successors of o_i that are allocated to the same core as the operation o_j. Since these successors are executed on the same core, and thus sequentially after o_j, it is not necessary to add synchronizations between them and o_i. $sync_{ijk}$ is a binary variable which indicates whether synchronization is needed between o_i and o_j if o_j is allocated to p_k. Therefore, $sync_{ijk} = 1$ iff $\alpha(o_j) = p_k$ and $\alpha(o_i) \neq p_k$ and $S(o_j) = \max_{o_{j'} \in succ(o_i):\ \alpha(o_{j'})=p_k}(S(o_{j'}))$ where α is the allocation function. Expressions 9 and 10 capture this constraint. V_{ik} is a binary variable that is set to one only if $\alpha(o_i) \neq p_k$. It is used to define for which cores a synchronization is needed between o_i and its successors and therfore needed to compute the value of variable $sync_{ijk}$. In other words, if a successor is allocated to the same core as o_i, no synchronization is needed. Expressions 11 and 12 capture this constraint. Variable Q_{ik} denotes the earliest start time among the start times of all the successors of o_i that are allocated to processor p_k. It is computed using expressions 13 and 14.

The start time of each operation o_j is computed using expression 15. The synchronization cost is introduced taking into account the synchronizations with all the predecessors of o_j that are allocated to different cores.

$$\forall\ o_i \in V, \sum_{p_k \in P} x_{ik} = 1 \tag{3}$$

$$\forall o_i \in V, E(o_i) = S(o_i) + C(o_i) \tag{4}$$

$$\forall\ o_i \in V, S(o_i) \geq R(o_i) \tag{5}$$

$$\forall\ o_i \in V, E(o_i) \leq D(o_i) \tag{6}$$

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, \\ E(o_i) \leq S(o_j) + M \times (3 - x_{ik} - x_{jk} - b_{ij}) \quad (7)$$

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, \\ E(o_j) \leq S(o_i) + M \times (2 - x_{ik} - x_{jk} + b_{ij}) \quad (8)$$

$$\forall o_i \in V, \sum_{\forall p \in P, \forall o_j \in pred(o_i)} sync_{ijk} = Q_{ik} \quad (9)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), sync_{ijk} \leq x_{jk} : \forall o_i \in V \quad (10)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), V_{ik} \geq x_{jk} - x_{ik} : \forall o_i \in V \quad (11)$$

$$\forall o_i \in V, V_{ik} \leq \sum_{\forall o_j \in succ(o_i)} \left(x_{jk} - x_{ik}\right) \quad (12)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ik} \leq S(o_j) + M \times (1 - x_{jk}) \quad (13)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), \\ Q_{ik} \geq S(o_j) - M \times (1 - sync_{ijk}) \quad (14)$$

$$\forall o_j \in V, \forall o_i \in pred(o_j), \\ S(o_j) \geq \\ \left[E(o_i) + \sum_{\forall p_k \in P, \forall o_{i'} \in pred(o_j)} sync_{i'jk} \times synCost\right] \quad (15)$$

D. Multi-core Scheduling Heuristic

Multi-core scheduling problems are known to be NP-hard resulting in exponential resolution times when exact algorithms are used. Heuristics have been extensively used in order to solve multi-core scheduling problems [14]. In most situations they lead to results of good quality in practicle resolution times. Greedy list heuristics are used in the context of offline multi-core scheduling since they are very fast. In the following, we propose a greedy list heuristic inspired by [11] for scheduling operation graphs representing FMU co-simulations under real-time constraints. The proposed heuristic is priority based. The scheduling priority expresses the criticality of a given operation which is measured by how close it is to miss its deadline if scheduled on a specific core: $\rho_{i,k} = D(o_i) - E(o_i)$ where $\rho_{i,k}$ and $E_j(o_i)$ are the scheduling priority and the end date of operation o_i respectively, computed when the latter is scheduled on core p_k.

The proposed heuristic builds the multi-core schedule iteratively. At each iteration, a list of candidate operations is constructed. An operation is added to the list of candidate operation if all its predecessors have been scheduled. The heuristic computes the priority for each candidate operation on every core and selects the core for the which the priority is maximized. After that, a list of operation-best core pairs is obtained. The heuristic selects from this list the operation whose priority is the smallest among all the operations in the list. Synchronization operations are added between the scheduled operation and all its predecessors that were allocated to different cores. The heuristic repeats this procedure and finally stops when all the operations have been scheduled. This heuristic is called repeatedly as long as the scheduling interval is increased according to the proposed method given in Section VI-B. Algorithm 1 lists the proposed real-time multi-core scheduling heuristic.

Algorithm 1: Multi-core scheduling heuristic

Input : Operation graph $G(V, A)$, set of cores P;
Output: Schedule of operations $o_i \in V$ on cores $p_k \in P$;
Set O the set of operations without predecessors;
Set $sync$ the cost of one synchronization operation;
Set $L_k : p_k \in P$ the length of schedule of core p_k;
foreach $p_k \in P$ **do**
  $L_k \leftarrow 0$;
while $O \neq \emptyset$ **do**
  foreach $o_i \in O$ **do**
    $\rho \leftarrow \infty$; `// Initialize the priority of` o_i
    $S(o_i) \leftarrow \max(R(o_i), \max_{o_j \in pred(o_i)}(E(o_j)))$;
    foreach $p_k \in P$ **do**
      $syncCost \leftarrow 0$;
      $S(o_i) \leftarrow \max(S(o_i), L_k)$; `// Start time of` o_i `if executed on` p_k
      foreach $o_j \in pred(o_i)$ **do**
        if *o_j is scheduled on a core $p_{k'} \neq p_k$* **then**
          $syncCost \leftarrow syncCost + sync$;
      $S(o_i) \leftarrow S(o_i) + syncCost$;
      $E(o_i) \leftarrow S(o_i) + C(o_i)$;
      $\rho' \leftarrow D(o_i) - E(o_i)$; `// priority of` o_i `if executed on` p_k
      if $\rho' > \rho$ **then**
        Set $\rho \leftarrow \rho'$;
        Set $BestCore(o_i) \leftarrow p_k$;
  Find $o_{i'}$ with the smallest priority ρ in O;
  Schedule $o_{i'}$ on its core $BestCore(o_{i'})$;
  $p_{best} \leftarrow BestCore(o_{i'})$;
  $L_{best} \leftarrow E(o_{i'})$;
  Remove $o_{i'}$ from the set O;
  Add to the set O all successors of $o_{i'}$ for which all predecessors are already scheduled;

The scheduling heuristic contains three nested loops. The outermost loop is executed until all the operations are scheduled. At each iteration, one operation is scheduled. Therefore, the outermost loop is executed n times where n is the number of operation in the operation graph. In the inner loops, the heuristic attempts to schedule all the ready operations on all the available cores. As such, the inner loops execute in

$\mathcal{O}(nm)$, where m is the number of cores. From the foregoing, the complexity of the heuristic is evaluated to $\mathcal{O}(mn^2)$.

VII. Evaluation

In this Section, we evaluate our proposed approach. We implemented a random generator of operation graphs to run tests. The random generation is not presented in this paper. We present the evaluation of the performances of the scheduling ILP and heuristic for different sizes of operation graphs and multi-core processors. We consider the proposed ILP as a reference and compare it with our heuristic performances.

A. Comparison of Resolution Times

We compared the resolution times of the real-time scheduling ILP and heuristic. We tested the real-time scheduling heuristic on operation graphs with a maximum number of operations of 1000. Tests were performed by fixing the number of cores and varying the number of operations. The obtained results are shown in Figure 9a, Figure 9b, and Figure 9c for 2, 4, and 8 cores respectively. The real-time scheduling ILP is able to solve smaller graphs than the heuristic ILP within acceptable times, and the resolution time increases exponentially as the graph size increases. On the other hand, the real-time scheduling heuristic produces results in short times keeping the resolution times within practical bounds even for large operation graphs. It is very important to produce the schedule within short times even if the latter is computed offline. In fact, in the context of co-simulation, the time needed to compute the schedule is suffered by the user. Therefore, a fast heuristic is needed to allow launching the co-simulation within acceptable times even for large industrial applications.

B. Schedulability

We run tests in order to measure the quality of our proposed heuristic. Because the execution of the ILP takes long times, we limited these tests to operation graphs of small sizes. We generated five sets of operation graphs containing each 10 graphs of sizes between five and 50. We ensured that all the generated operation graphs are schedulable by applying the ILP. Then, we apply our heuristic and save the number of schedulable operation graphs for which the heuristic is able to find a solution. The application of the heuristic resulted in an interesting quality, especially when considering its very fast resolution time compared to the ILP algorithm. The heuristic succeeded in scheduling $54\%, 80\%$, and 94% of the operation graphs for $2, 4$, and 8 cores respectively.

VIII. Conclusion

In this paper, we dealt with the problem of real-time scheduling of operation graphs representing FMI HiL co-simulations on multi-core processors. First, we extended the operation graph of a co-simulation under real-time constraints by assigning real-time parameters to the operations. We proposed a method for propagating real-time constraints imposed by inputs and outputs of the real component on gate operations of the simulated component. Second, we proposed an ILP

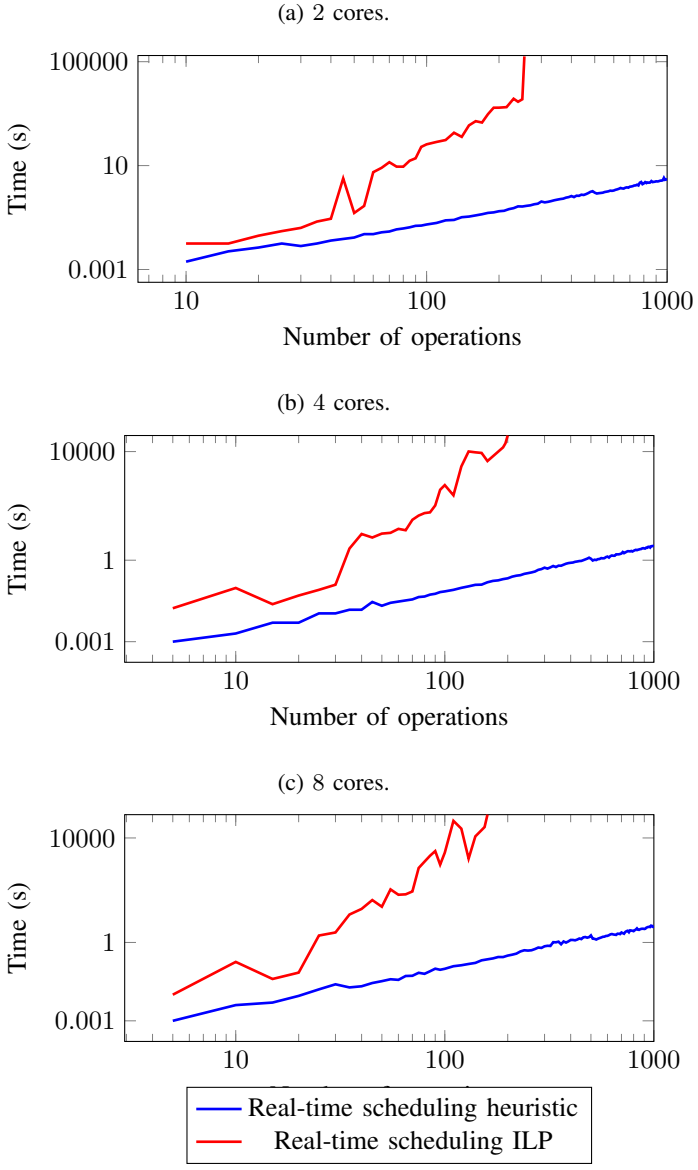

Fig. 9: Comparison of the real-time scheduling resolution time.

and an offline heuristic to perform the real-time multi-core scheduling. We evaluated these algorithms for different sizes of co-simulations and operation graphs. We present below some possible research directions for future work.

In our approach, we used a schedulability analysis based on simulation. In future work, we aim at deriving an analytic schedulability condition or finding a non pessimistic schedulability interval. Our approach relies on propagating the real-time constraints to all the operations. An interesting alternative to our method consists in defining latency constraints on gate operations only. Then, scheduling algorithms suitable for latency constraints can be applied. Finally, it is worth investigating the use of preemptive scheduling algorithms, for instance by only allowing the preemption of operations which have long execution times.

REFERENCES

[1] J. Bélanger, P. Venne, and J. N. Paquin. The what, where and why of real-time simulation. *Planet RT*, 1(0):1, 2010.

[2] A. Ben Khaled, M. Ben Gaid, N. Pernet, and D. Simon. Fast multi-core co-simulation of cyber-physical systems: application to internal combustion engines. *Simulation Modelling Practice and Theory*, 47:79–91, 2014.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 1990.

[7] B. P. Dave, G. Lakshminarayana, and N. K. Jha. Cosyn: hardware-software co-synthesis of heterogeneous distributed embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):92–104, 1999.

[8] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4), 10 2011.

[9] C. Faure and N. Pernet. Propagation rules of real-time constraints on physical systems simulators in a hardware-in-the loop context. In *Proceedings of the 20th International Conference on Real-Time Networks and Systems*, pages 31–40. ACM, 2012.

[10] FMI development group. Functional mock-up interface for model exchange and co-simulation, 07 2014.

[11] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[12] E. Grolleau, J. Goossens, and L. Cucu-Grosjean. On the periodic behavior of real-time schedulers on identical multiprocessor platforms. *arXiv preprint arXiv:1305.3849*, 2013.

[13] P. Jayachandran and T. F. Abdelzaher. The case for non-preemptive scheduling in distributed real-time systems. Technical report, 2007.

[14] Y. K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[15] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: a cyber-physical systems approach*. MIT Press, 2 edition, 2017.

[16] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on Switching Theory, 1959*, pages 285–292, 1959.

[17] 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.

[18] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 328–339. IEEE, 1999.

[19] M. Qamhieh, S. Midonnet, and L. George. Graph-to-segment transformation technique minimizing the number of processors for real-time multiprocessor systems. In *Workshop on Power, Energy, and Temperature Aware Real-time Systems (PETARS)*, 2012.

[20] S. E. Saidi, N. Pernet, and Y. Sorel. Automatic parallelization of multi-rate fmi-based co-simulation on multi-core. In *Proceedings of the Symposium on Theory of Modeling & Simulation*, page 5. Society for Computer Simulation International, 2017.

[21] M. Spuri and J. A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, 1994.