Acceleration of FMU Co-Simulation On Multi-core Architectures

Salah Eddine Saidi[1] Nicolas Pernet[1] Yves Sorel[2] Abir Ben Khaled[1]

[1]IFP Energies nouvelles, Rueil-Malmaison, France,
{salah-eddine.saidi,nicolas.pernet,abir.ben-khaled}@ifpen.fr
[2]INRIA, Paris, France, yves.sorel@inria.fr

Abstract

The design of cyber-physical systems is a complex process and relies on the simulation of the system behavior before its deployment. Co-simulation allows system designers to simulate a whole system composed of a number of interconnected subsystems. Traditionally, these models are modeled by experts of different fields using different tools, and then integrated into one environment to perform simulation at the system-level. This results in complex and heavy co-simulations and requires adequate solutions and tools in order to reduce the execution time. Unfortunately, most modeling tools perform only mono-core simulations and do not take advantage of the omnipresent multi-core processors. This paper addresses the problem of efficient parallelization of co-simulations. It presents a multi-core scheduling heuristic for parallelizing FMI-compliant models on multi-core processors. The limitations of this heuristic are highlighted and two solutions for dealing with them are presented. The obtained speed-up using each of these solutions is illustrated and discussed for further improvements.

Keywords: FMI, co-simulation, multi-core, scheduling, heuristic

1 Introduction

Cyber-physical systems incorporate a combination of computational elements which collaborate in order to control physical processes. The complex nature of such systems requires cost, time and effort-effective design methodologies; therefore predicting their behavior and functioning scenarios before testing the real system is becoming more and more an indisputable step. Co-simulation aids in achieving these requirements as it allows the assessment of the design of the system by imitating its behavior. It consists mainly in simulating, on a computer, the global behavior of a multi-physics system composed of a number of interconnected subsystems. System designers can then identify potential design flaws and correct them before deploying the system.

Co-simulation faces however a number of challenges. Actually, the simulated system is described by several interacting models which are often developed by experts of different fields using different tools and following different design approaches. The diversity of modeling tools and involved teams makes the coupling of the models a complex task. In fact, co-simulation necessitates efficient synchronized communications between the models where each model must be able to detect and respond to events of other models. Thanks to the FMI (Functional Mock-up Interface) standard (Blochwitz et al., 2011), it is now possible to easily couple diverse models originating from different developers and tools. Nevertheless, executing FMI-compliant models raises some issues, which unless well handled, may reduce the co-simulation performance and limit the benefits of FMI.

One major issue is the question of how to reduce the co-simulation execution time. Integrating heterogeneous models into one environment usually results in a complex and heavy to execute co-simulation which increases the demand of processing power.

As is well-known, increasing CPU frequency by means of silicon integration has reached its possible limits and semiconductor manufacturers switched in last years to building multi-core processors, i.e. integrating multiple processors into one chip allowing parallel processing on a single computer. Multi-core processors can reduce the execution time of a computational task by dividing it into several subtasks and assigning a subset of subtasks to each core to be processed in parallel. Most simulators, however, have mono-core simulation kernels and do not take advantage of the computation power brought by multi-core architectures. Therefore, enabling parallel execution of heavy co-simulations on multi-core processors is keenly sought by the developers and the users of simulation tools. However, fulfilling this objective is not trivial and appropriate parallelization schemes need to be applied on co-simulation models in order to accelerate their execution on multi-core processors. It is worth noting that in this paper the term co-simulation is generic and is used to refer to the simulation of FMUs generated from FMI for Co-Simulation as well as FMI for Model Exchange.

FMI gives information about inputs and outputs relationships inside a model that is exported as an FMU

DOI 10.3384/ecp16124106

(Functional Mock-up Unit). 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. Input, output and state variables are updated by what we name "operations" which may call different functions provided by the FMU.

Given these features, various execution possibilities can be realized and the parallelization of co-simulation models on a multi-core processor can be seen as the following problem: Find an allocation of the different operations to the different cores and define an execution order, i.e. schedule the operations that are allocated to each core. When solving this problem, the utilization of the available cores has to be optimized in order to achieve the best acceleration. Using parallel computing terminology, the problem consists in finding a schedule for all the operations of the co-simulation on a multi-core processor. This paper deals with the problem of scheduling operations of heavy complex co-simulation models on multi-core processors in order to accelerate the simulation execution. It follows the approach presented in (Ben Khaled et al., 2014) by addressing two limitations of the previous work. First, an efficient multi-core scheduling can not be obtained without taking into account a good estimation of each operation's execution time. Second, the non-thread-safe implementation of FMUs prevent full exploitation of the potential parallelism of co-simulation graphs. Techniques for dealing with these limitations are here compared.

The rest of the paper is organized as follows. Next section presents related work on multi-core execution of simulations. Then our parallelization approach, firstly presented in (Ben Khaled et al., 2014), is described in section 3, including a discussion about its present limitations. The fourth section presents our contribution, including the use of a toolchain for profiling co-simulation graph parallelism and explores the theoretical gain in execution speed-up over different architectures. Theoretical results are discussed and compared to real co-simulation executions in xMOD[1]. xMOD is a co-simulation and a virtual experimentation platform, which allows mixing stand-alone and tool coupling co-simulations and the optimization of complex models execution. It provides a user-friendly interface in order to extend the simulation use to non-experts and ensure the continuity from Model-in-the-Loop to Hardware-in-the-Loop simulations. The last section concludes the paper and gives an outlook into our ongoing and future work.

2 Related Work

In order to achieve simulation acceleration using multi-core execution, different approaches are possible and were already explored. From a user point of view, it is possible to modify the model design in order to prepare its multi-core execution, for example by using marked functions or Modelica extensions as in (Elmqvist et al., 2015; Gebremedhin et al., 2012). From a modeling tool provider point of view, if providing OpenMP ready libraries is possible, the key feature for simulation acceleration is to provide techniques which offer speed-up whatever the model is. Proposing parallel solvers or automatic parallel executions of model equations as in (Elmqvist et al., 2014; Sjölund et al., 2010) is also an efficient way. In this paper, we address the problem from a co-simulation tool provider point of view. In such a tool, the user connects different FMUs, embedding solvers or not. In this case, it is not possible to change the models, the solvers, or the modeling tools. Such FMU assembly defines a graph of operations and the main opportunity to improve the co-simulation execution is consequently to accomplish an automatic parallelization of this graph. As shown in (Ben Khaled et al., 2012), splitting a model into several FMUs, by isolating discontinuities, may reduce the simulation time, even in the case of a mono-core execution. (Ben Khaled et al., 2014) presented the RCOSIM approach. It consists in using each FMU information on input/output causality to build a graph, with an increased granularity and then exploiting the potential parallelism by using a heuristic to build an off-line multi-core schedule. This method has been tested on a real industrial model and significant speed-up was obtained. This approach was implemented in the co-simulation tool xMOD and is available in its 2015 release.

3 Parallelization approach

3.1 Principle

The parallelization concept of xMOD is based on a task DAG (Directed Acyclic Graph) scheduling approach. Thanks to FMI, it is possible to access information about the internal structure of a model encapsulated in an FMU. In particular, FMI allows the identification of Direct Feedthrough and Non Direct Feedhrough outputs of a model. Since connections between different models of the co-simulation are also known, all data dependencies between the operations are known. Figure 1 shows an example of two models and their inter and intra-model dependencies.

The co-simulation can be described by a DAG where each vertex represents one operation and each edge describes a precedence constraint between two operations. The approach proceeds in two steps: First, the co-simulation DAG is constructed and then, the operations are allocated to the available cores in such a way to minimize the makespan of the graph. The makespan corresponds the execution time of the whole DAG.

The transformation of each model into an operation

[1] http://www.xmodsoftware.com/

graph allows the parallelization of the model instead of considering it as an atomic block. Consequently the potential parallelism of the entire co-simulation is increased and can be better adapted to the hardware parallelism (number of cores in the case of a multi-core processor). The potential parallelism of a graph corresponds to vertices that are not dependent which characterize the partial order of the graph.

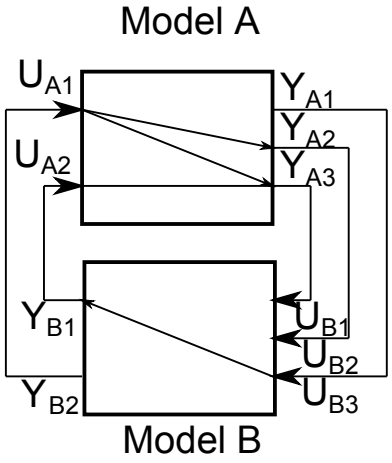

Figure 1. Inter and intra-model dependencies of two models.

3.2 Multi-core scheduling heuristic

The co-simulation DAG is built by exploring the relations between the models and between the operations of the same model. The operations are either $update_{output}$, $update_{input}$ or $update_{state}$. An $update_{output}$ operation corresponds to an FMI *Get* function that allows getting the value of an FMU output and an $update_{input}$ operation corresponds to an FMI *Set* function which allows setting the value of an input. An $update_{state}$ operation corresponds to calling FMI functions needed to perform an integration step (*SetTime*, *GetDerivatives*, and *SetContinuousStates*, etc., in the case of Model Exchange or *DoStep* in the case of Co-Simulation) (FMI development group, 2014). A vertex is created for each operation and edges are then added between vertices if a data dependency exists between the corresponding operations. This information can be extracted from the model's FMU. When using FMI 1.0 which does not give information about the dependencies between the state variables computation and the input and output variables computations, it is necessary that edges connect all $update_{input}$ operations and the $update_{state}$ operation of the same model, since all inputs at instant k need to be updated before updating the state to X_{k+1}. Furthermore, edges are placed between all $update_{output}$ operations and the $update_{state}$ operation of the same model, because the computation at instant k of an output Y_k must be performed with the same value of the state as for all the outputs belonging to the same model. Running the co-simulation consists in executing the graph repeatedly. At each co-simulation step the whole graph is executed and a new execution of the graph cannot be started unless the previous one was totally finished. Figure 2 illustrates the graph constructed from the two models of Figure 1.

In order to achieve fast execution of the co-simulation on a multi-core processor, an efficient allocation and scheduling of the DAG vertices has to be performed. xMOD uses an off-line scheduling heuristic similar to the one proposed in (Grandpierre et al., 1999). (Ben Khaled et al., 2014) presented the use of this heuristic and the speed-up obtained by applying it on an industrial combustion engine model. The heuristic considers the execution time of each operation and aims at computing a schedule that minimizes the makespan of the graph.

Using the execution time C_i of each operation OP_i, the heuristic computes first the earliest start and end dates from the graph start denoted S_i and E_i, then the critical path $CP := \max E_i$ (Algorithm 1). After that, the latest start and end dates from the graph end denoted S_i^* and E_i^* and then the flexibility $F_i = CP - E_i - E_i^*$ are computed (Algorithm 2).

```
Initialization;
Set Ω the set of all the operations;
Set O the set of operations without predecessors;
foreach OP_i ∈ O do
    S_i := 0; E_i := S_i + C_i;
end
Set O' the set of operations whose all immediate predecessors
  were treated;
while O' ≠ ∅ do
    foreach OP_i ∈ O' do
        S_i := max(E_h : OP_h → OP_i);
        (OP_h are the immediate predecessors of OP_i);
        E_i := S_i + C_i; Remove OP_i from the set O';
        Add to the set O' all successors of OP_i for which all
          predecessors were already scheduled;
    end
end
CP := 0;
foreach OP_i ∈ Ω do
    if CP < E_i then
        CP := E_i;
    end
end
```

Algorithm 1: Computation of S_i, E_i and CP

At each step, the heuristic computes for a given operation the schedule pressure on a specific core. The schedule pressure is the difference between the makespan increase, by allocating this operation to this core, and the operation's flexibility. The heuristic updates the set of candidate operations to be scheduled at each step. An operation is added to the set of candidate operations if it has no predecessor or if all of its predecessors have already been scheduled. The set of candidate operations holds the partial order associated to the graph. Then, for each candidate operation, the schedule pressure is computed on each core in order to find its best core, the one that minimizes the pressure. After this step, a list of candidate operation-best core pairs is obtained. Finally, the

DOI 10.3384/ecp16124106

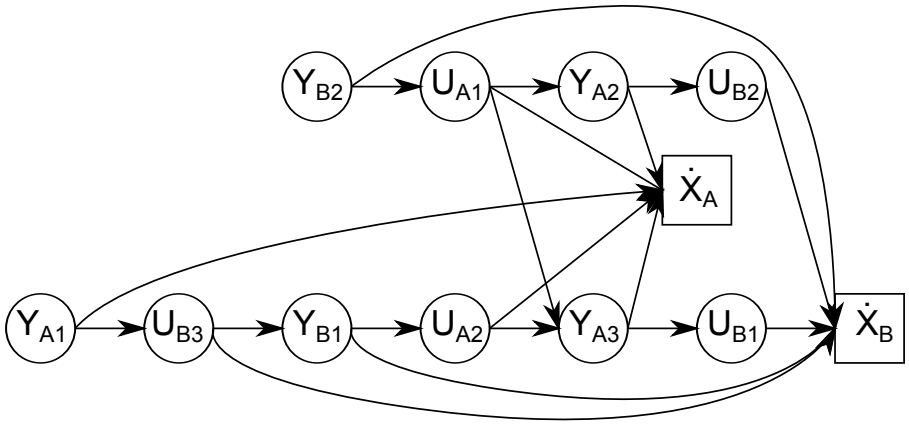

Figure 2. Dependency graph of the models of Figure 1.

```
Initialization;
Set Ω the set of all the operations;
Set O the set of operations without successors;
foreach OP_i ∈ O' do
    E*_i := 0; S*_i := E*_i + C_i;
end
Set O' the set of operations whose all immediate successors were
  treated;
while O' ≠ ∅ do
    foreach OP_i ∈ O do
        E*_i := max(S*_h : OP_i → OP_h);
        (OP_h are the immediate successors of OP_i);
        S*_i := E*_i + C_i; Remove OP_i from the set O';
        Add to the set O' all predecessors of OP_i for which all
          successors were already scheduled;
    end
end
foreach OP_i ∈ Ω do
    F_i := CP − E_i − E*_i;
end
```

Algorithm 2: Computation of S_i^*, E_i^* and F_i

```
Initialization;
Set Ω the set of all the operations;
Set Γ the set of all the available cores;
Set O the set of operations without predecessors;
while O ≠ ∅ do
    foreach OP_i ∈ O do
        Set cost_i to ∞; (cost of OP_i is set to the maximum
          value);
        foreach Core_j ∈ Γ do
            S'_i := max(S_i, T_Core_j); (new start date of OP_i when
              executed on Core_j);
            cost_i,j := S'_i + C_i + E*_i − CP; (cost of OP_i when
              executed on Core_j);
            if cost_i,j < cost_i then
                Set cost_i := cost_i,j;
                Set BestCore_i := Core_j;
            end
        end
    end
    Find OP_i with maximal cost_i in O;
    Schedule OP_i on its core BestCore_i;
    Set k := BestCore_i;
    T_Core_k := T_Core_k + C_i; (Advance the time of Core_k);
    Remove OP_i from the set O;
    Add to the set O all successors of OP_i for which all
      predecessors are already scheduled;
end
```

Algorithm 3: Multi-core scheduling heuristic

operation with the largest pressure on its best core is selected and scheduled. The heuristic repeats this procedure until all operations are scheduled (Algorithm 3).

This heuristic has originally been used to implement critical hard real-time applications where the execution times are usually estimated as the WCET (Worst Case Execution Time). On the contrary, co-simulation is not safety critical and the main goal here is to achieve fast execution, so average computation times can be used. So far, execution times in xMOD are estimated based on the observation of practical examples as follows: $update_{state}$ operations are by far more costly so they are assigned significantly higher execution times then $update_{output}$ operations, whereas $update_{input}$ operations are just data copy whose cost is negligible.

3.3 Limitations of the approach

Although the presented scheduling heuristic resulted in interesting co-simulation speed-ups, it has some limitations that have to be considered in the multi-core scheduling problem in order to obtain better performances. First, so far, the multiprocessor scheduling heuristic uses empiric operations execution times. By using realistic execution times for each operation, the multi-core execution of the simulation should be improved. In this paper, we present some results, based on a profiling technique.

Second, FMI standard does not presently require that FMU functions have to be thread-safe, i.e. they cannot be executed simultaneously as they may share some resource (variables for example) that might be corrupted if two operations try to use it at the same time. This implies that at any instant during the execution of the co-simulation, one and only one operation of the same FMU can be executed. Consequently, if the scheduling

heuristic allocates two or more operations belonging to the same FMU to different cores, a mechanism that ensures these operations are executed in strictly different time intervals must be set up.

4 Proposed solutions

This section presents a theoretical study of the achievable speed-up on a use-case, using the SynDEx[2] software (Sorel, 2004, 2005). Then, these theoretical results are compared with xMOD co-simulation runs, with two different implementations for guaranteeing a mutual exclusion between different operations of the same FMU.

4.1 Toolchain

A toolchain is proposed to assist the developer in parallelizing co-simulations. Using this toolchain, it is possible to assess new solutions before implementing them in xMOD thanks to the SynDEx software. SynDEx is a system level CAD software based on the Algorithm-Architecture Adequation (AAA) methodology (Sorel, 1996). It was developed to optimize the implementation of real-time distributed applications onto multicomponent architectures. The workflow is illustrated in Figure 3. When different FMUs are imported into xMOD and connected together, a file which describes inter-model connections is generated. This file and the XML files of the different FMUs of the co-simulation are passed to a converter which parses the files and produces equivalent files (.sdx) compliant to the SynDEx format. The co-simulation code is profiled in order to obtain the execution times of the different operations which are introduced in SynDEx. SynDEx offers the possibility to use the multi-core scheduling heuristic outlined in this paper , as well as other kinds of heuristics, and therefore makes it possible to study the achievable co-simulation speed-up before implementing the heuristic in xMOD.

4.2 Use-case description

In this work, experiments have been carried out on a Spark Ignition (SI) RENAULT F4RT engine co-simulation using 5 FMUs. It is a four-cylinder in line Port Fuel Injector (PFI) engine in which the engine displacement is 2000 cm^3. The air path is composed of a turbocharger with a mono-scroll turbine controlled by a waste-gate, an intake throttle and a downstream-compressor heat exchanger (Figure 4). The engine model was developed using ModEngine library (Benjelloun-Touimi et al., 2011). ModEngine is a Modelica library that allows for the modeling of a complete engine with diesel and gasoline combustion models. The engine model was imported into xMOD using the FMI export features of the Dymola[3] tool. This use-case has over 100 operations which are scheduled by the multi-core scheduling heuristic.

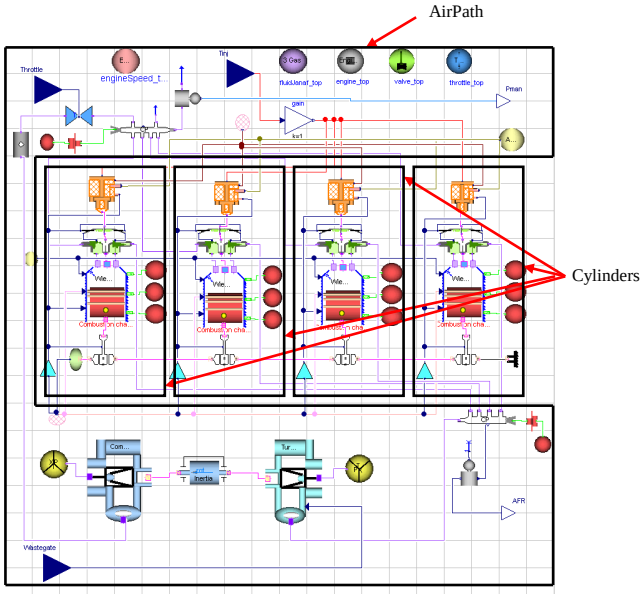

Figure 4. Spark Ignition (SI) RENAULT F4RT engine model.

4.3 Results and Discussions

Using the toolchain, a .sdx file of the use-case was generated in order to evaluate, in SynDEx, the theoretical speed-up obtained by parallelizing the model on different numbers of cores, using the multi-core scheduling heuristic of section 3.2. For each schedule the speed-up is computed by dividing the mono-core schedule makespan by the schedule makespan. Figure 5 gives the different theoretical speed-ups. The best speed-up is close to 3,6 and is reached with 6 cores. Finding the minimal number of cores which offers the maximum speed-up is interesting if a large number of simulation runs (possibly with different parameters) have to be performed: If a large number of cores is available, multiple runs could be launched in parallel with the adequate number of cores dedicated to each run. This research of the minimal number of necessary cores to reach the maximum speed-up could be scripted and automatically performed before the simulation.

In order to tackle the constraint of non thread-safe FMU functions, two mutual exclusion strategies have been implemented in xMOD and the performance obtained using each of them has been evaluated. The first one does not modify the multi-core scheduling heuristic result and uses a dedicated mutex (system object that guarantees mutual exclusion) for each FMU: Every time an FMU function call is made at runtime, the associated mutex have to be acquired before the execution of

[2]http://www.syndex.org/

[3]http://www.3ds.com/products-services/catia/products/dymola

DOI 10.3384/ecp16124106

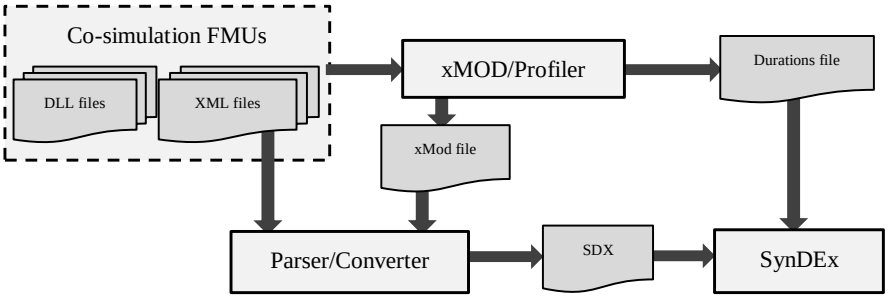

Figure 3. Proposed toolchain to assist in the development and assessment of scheduling heuristics.

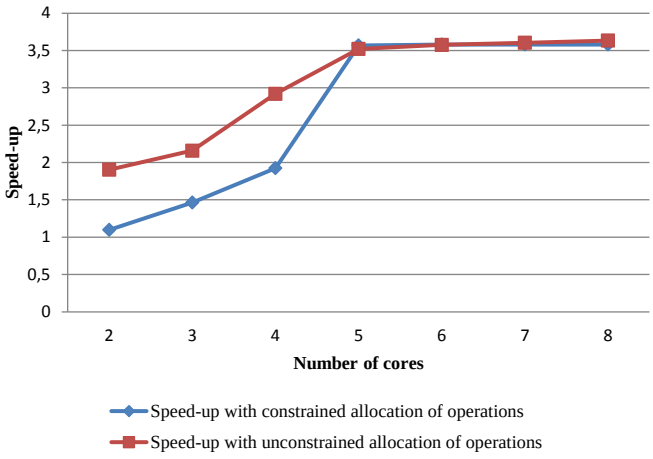

Figure 5. Theoretical speed-up.

the function code can be started. The second solution is explained in (Ben Khaled et al., 2014) and consists in modifying the multi-core scheduling heuristic to always allocate the operations of a same FMU to the same core (constrained allocation). If constrained allocation is used, the search space of the scheduling heuristic is reduced, i.e. at each step, for a given candidate operation, if there is another operation of the same FMU that has already been allocated to a specific core, the candidate operation is allocated to this same core without the need to test it on the other cores. Thanks to SynDEx, it is easily possible to theoretically estimate the impact of using the constrained allocation in the multi-core scheduling heuristic. Results are given in Figure 5. It shows that the expected speed-up in the case of constrained allocation is less than the one using unconstrained allocation, when the number of cores is less than 5, but similar when 5 cores or more are available. When using less than 5 cores, the large number of $update_{output}$ operations can be efficiently allocated only if the unconstrained allocation is used: The speed-up difference between the constrained and unconstrained allocation cases is due to this restriction on the allocation. Five is the minimal number of cores for enabling the execution of each $update_{state}$ operation on a different core. Due to the predominant execution times of the $update_{state}$ operations, their impact on the speed-up overrides the possibility of optimizing the allocation of the other operations. This explains why the speed-up difference between the unconstrained and the constrained allocation cases becomes very small with 5 cores or more.

In order to compare the two mutual exclusion strategies, we implemented them in xMOD. Execution times measurements were performed by getting the system time stamp at the beginning of the simulation and after 30 seconds of the simulated time. As previously, we compare the speed-up by dividing the mono-core simulation execution time by the simulation execution time on a fixed number of cores. Figure 6 sums up the results, where unconstrained allocation corresponds to the use of mutex objects. It shows the impact of mutex overhead on the speed-up. Whatever the number of the available cores, the speed-up remains close to 1,3. On the contrary, the implementation in xMOD of the constrained allocation gives similar results in terms of speed-up improvement when increasing the number of cores until 5. Nevertheless, the maximum measured speed-up (2,4) remains smaller than the theoretical one (3,5). In fact, the theoretical speed-up computation considers the makespan ratio without estimating any synchronization cost between cores. The real implementation in xMOD contains synchronization objects between operations to ensure the consistency of data dependencies which certainly have an important impact on the speed-up.

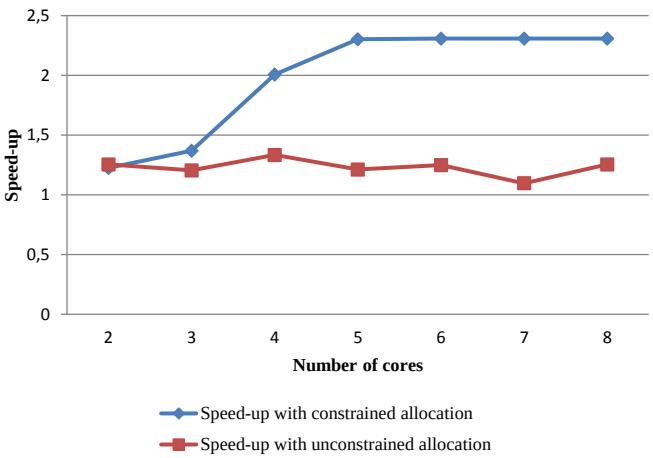

Figure 6. Measured speed-up.

5 Conclusion and Future Work

The work presented in this paper dealt with the problem of co-simulations acceleration by means of parallelization on multi-core processors. We proposed to extend our previous work by taking into account operations execution times in the multi-core scheduling heuristic. This allows the optimization of the number of the dedicated cores to the simulation, by performing architecture exploration with SynDEx. Our experiments in xMOD on an industrial use-case, gave important speed-up results (2,4). Nevertheless, it also shows the impact of the mutual exclusion constraint on the co-simulation acceleration. Providing thread-safe FMU implementation could offer important simulation acceleration opportunities. In our ongoing work, we are exploring graph transformation techniques to improve the handling of the mutual exclusion problem of FMUs. We also envision to extend these results to the multi-rate co-simulation of FMUs by developing an efficient multi-core scheduling heuristic to handle it.

References

A. Ben Khaled, M. Ben Gaïd, D. Simon, and G. Font. Multicore simulation of powertrains using weakly synchronized model partitioning. In *IFAC Workshop on Engine and Powertrain Control Simulation and Modeling ECOSM*, pages 448–455, Rueil-Malmaison, France, 2012. doi:10.3182/20121023-3-FR-4025.00018.

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. ISSN 1569-190X. doi:http://dx.doi.org/10.1016/j.simpat.2014.05.002. URL `http://www.sciencedirect.com/science/article/pii/S1569190X14000665`.

Z. Benjelloun-Touimi, M. Ben Gaïd, J. Bohbot, A. Dutoya, H. Hadj-Amor, P. Moulin, H. Saafi, and N. Pernet. From physical modeling to real-time simulation: Feedback on the use of Modelica in the engine control development toolchain. In *8th Int. Modelica Conf.*, Dresden, Germany, Mar 2011. Linköping Univ. Electronic Press.

T. Blochwitz, T. Neidhold, M. Otter, M. Arnold, C. Bausch, M. Monteiro, C. Clauß, S. Wolf, H. Elmqvist, H. Olsson, A. Junghanns, J. Mauss, D. Neumerkel, and J.-V. Peetz. The Functional Mockup Interface for tool independent exchange of simulation models. In *8th Int. Modelica Conf.*, Dresden, Germany, Mar 2011. Linköping Univ. Electronic Press. ISBN 978-91-7393-096-3. doi:10.3384/ecp11063105.

H. Elmqvist, S.E. Mattsson, and H. Olsson. Parallel model execution on many cores. In *10th Int. Modelica Conf.*, Lund, Sweden, 2014.

H. Elmqvist, H. Olsson, A. Goteman, V. Roxling, D. Zimmer, and A. Pollok. Automatic gpu code generation of modelica functions. In *11th Int. Modelica Conf.*, Versailles, France, 2015.

FMI development group. Functional mock-up interface for model exchange and co-simulation, July 2014. URL `https://www.fmi-standard.org/`.

M. Gebremedhin, A. Hemmati Moghadam, F. Fritzson, and K. Stavaker. A data-parallel algorithmic modelica extension for efficient execution on multi-core platforms. In *9th Int. Modelica Conf.*, Munich, Germany, 2012.

T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. URL `http://www-rocq.inria.fr/syndex/publications/pubs/codes99/codes99.pdf`.

M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In Linköping Univ. Electronic Press, editor, *3rd Int. Workshop on Equation- Based Object-Oriented Languages and Tools EOOLT*, page 71–80, Oslo, Norway, 2010.

Y. Sorel. Real-time embedded image processing applications using the algorithm architecture adequation methodology. In *Proceedings of IEEE International Conference on Image Processing, ICIP'96*, Lausanne, Switzerland, September 1996. URL `http://www-rocq.inria.fr/syndex/publications/pubs/icip96/icip96.pdf`.

Y. Sorel. Syndex: System-level cad software for optimizing distributed real-time embedded systems. *Journal ERCIM News*, 59:68–69, October 2004. URL `http://www-rocq.inria.fr/syndex/publications/pubs/ercim04/ercim04.pdf`.

Y. Sorel. From modeling/simulation with scilab/scicos to optimized distributed embedded real-time implementation with syndex. In *Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE'05*, Wuhan, China, October 2005. URL `http://www-rocq.inria.fr/syndex/publications/pubs/sosse05/sosse05.pdf`.