A

Automatic Parallelization of Multi-rate Simulink Control Models for Multi-core Architectures

CUMHUR ERKAN TUNCALI, Arizona State University
GEORGIOS FAINEKOS, Arizona State University
YANN-HANG LEE, Arizona State University

This paper addresses the problem of parallelizing model block diagrams for real-time embedded applications on multi-core architectures. Our approach is based on assigning each CPU core, a set of blocks to execute. The challenge is in finding an optimal or feasible mapping so that a synchronized execution of the blocks on different cores can be achieved with respect to the constraints induced by the control model and the target platform architecture. In order to solve the problem, we describe a Mixed Integer Linear Programming (MILP) formulation for finding a feasible mapping of the blocks to different CPU cores. For single-rate models, we use an objective function that minimizes the overall worst-case execution time on the target platform. For the multi-rate models, we solve the feasibility problem for finding a mapping which satisfies given block sampling period constraints. When the model size increases, solving these problems in a reasonable time becomes harder. For addressing this issue, we introduce a set of heuristics for reducing the number of constraints in the MILP formulation. For single-rate models, these heuristics help the MILP solver to find solutions that are closer to the optimal solution given a limited solver execution time. We study the scalability and efficiency of our approach with synthetic benchmarks of randomly generated directed acyclic graphs. We demonstrate applicability of our approach to practical problems using a Diesel engine controller from Toyota as a case study.

CCS Concepts:•**Computer systems organization** → *Embedded software;*

Additional Key Words and Phrases: Multiprocessing, embedded systems, optimization, model based development, Simulink, multi-rate, task allocation

ACM Reference Format:
Cumhur Erkan Tuncali, Georgios Fainekos, and Yann-Hang Lee, 2015. Automatic Parallelization of Multi-rate Simulink Models for Multi-core Architectures. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Model Based Design (MBD) has gained a lot of traction in the industries that develop safety critical systems. This is particularly true for industries that develop Cyber-Physical Systems (CPS) where the software implements control algorithms for the physical system. Using MBD, system developers and control engineers can design control algorithms on high-fidelity models. Most importantly, they can test and verify the system properties before having a prototype of the system. The autocode generation facility of MBD tools provides additional concrete benefit which helps in eliminating programming errors.

This research was partly funded by the NSF awards CNS-1446730 and IIP-1361926, and the NSF I/UCRC Center for Embedded Systems.
Authors' addresses: Arizona State University, Centerpoint Bldg. STE 203, 660 S. Mill Ave. Tempe, AZ 85281

DOI: http://dx.doi.org/10.1145/0000000.0000000

However, currently, the autocode generation processes of commercial tools focus on single-core systems. Namely, at the model level, there is no automatic support for producing code that runs on a multi-core system. This is problematic since advanced control algorithms, e.g., Model Predictive Control algorithms [Huang et al. 2013], are computationally demanding and may not be executed within the limited computation budget of a single-core embedded system. In this paper, we address this problem at the model level. Namely, given a data flow diagram of an embedded control algorithm, the worst-case execution times of the blocks and a computation budget (deadline or sampling period), can we automatically map the blocks of a model onto multiple sub-models which are executed on different cores and the real-time constraints are satisfied?

Depending on system requirements, the controller model can have single or multiple sampling rates. For instance, communicating with different hardware systems may require different subsystems to operate with different sampling periods/rates. In addition, for satisfying the hardware interface and performance requirements or for maintaining stability the system, the system designer has to determine the optimal sampling periods for the subsystems [Åström and Wittenmark 1997]. Most model based design tools, e.g. Simulink from The MathWorks Inc. [2015], support multi-rate design. For multi-rate designs, the problem of mapping the blocks onto the cores requires an analysis for the interaction between blocks with different sample rates and a consideration of the task scheduling algorithm on the target platform.

In particular, we focus on control models built in the Simulink MBD environment. Our goal is to produce a framework which determines the mapping of each block onto a CPU core and an execution order of the blocks inside the tasks. We aim at creating a single task for each sample rate on a CPU core, i.e., a single task on each core for single-rate models and possibly multiple tasks on each core for multi-rate models. A task contains all the blocks with the same sample rate which are mapped on the same core. Our consideration for scheduling the tasks on the target platform is based on the rate-monotonic scheduling algorithm which is a fixed priority partitioned scheduling mechanism (task migration between CPU cores is not allowed). Especially, in safety-critical systems, scheduling in a predictable and deterministic manner is highly important for verification and satisfying the certification requirements that are mandated by regulatory authorities. For example, multi-core architectures are classified as highly complex in the 2011/6 final report of European Aviation Safety Agency, EASA [2012] and in the Certification Authorities Software Team position paper CAST-32 Multi-core processors [2014]. These classifications highlight the difficulty of certifying safety-critical systems that are based on multi-core architectures.

In our previous work [Tuncali et al. 2015], our approach for single-rate models is explained. Our approach is based on having separate executables for each core while Simulink blocks are allocated in each core and executed in the execution order that we compute. In other words, we determine the execution order of the blocks inside each core while respecting the data dependencies between them. In this paper, we are extending our approach for single-rate models to multi-rate models. Our approach for the multi-rate models is based on seeking a mapping of the blocks onto different tasks on the CPU cores and an execution order of the blocks within the tasks in order to satisfy the deadline requirements of all the tasks. After code generation, we have a separate task for each sampling period in a model. The scheduling of the tasks should be taken into account for parallelizing multi-rate models. It must be guaranteed that the preemptions of the low priority tasks by the high priority tasks does not cause deadline misses. In particular our MILP formulation for multi-rate models incorporates scheduling related constraints between the tasks with different rates.

The contributions of this paper are,

1. providing a practical, automated solution to the Simulink model parallelization problem for multi-core architectures while considering the deadline requirements and scheduling of the parallelized application on the target platform, and
2. extending available Mixed Integer Linear Program (MILP) formulations for parallelizing control models to allow multi-rate execution.

2. RELATED WORK

There is a large amount of research being done on the optimization of scheduling multiple tasks on multi-core processors or multiple processors in the literature.

An exhaustive survey on real-time scheduling techniques for homogeneous multiprocessor architectures is provided by Davis and Burns [2011]. That survey evaluates different techniques by discussing their advantages and disadvantages. The authors state the main practical advantage of statically assigning the tasks onto the processors as the ability to apply available uniprocessor scheduling techniques and analyses on each processor in the system.

There are multiple studies on task parallelization. For optimal mapping of tasks to CPU cores, Yi et al. [2009], Bender [1996] and Ostler and Chatha [2007] discuss integer linear programming (ILP) techniques which constitute the foundation for our optimization formulation. The aforementioned approaches can be applied to single-rate Simulink models by substituting the tasks in the formulation with Simulink blocks, i.e, considering the blocks as tasks with dependencies. On the other hand, for multi-rate models, a set of blocks with the same rate should be considered as a single task in those approaches but this eliminates the parallelization opportunities inside a task. For most realistic models which consist of a significant number of blocks, ILP based approaches require introduction of heuristics to find an optimal or sub-optimal solution in a reasonable amount of time. Yi et al. [2009] take use of available loop level parallelism or functional pipelining in the system. An efficient constraint programming approach to the task allocation problem is described by Hladik et al. [2008]. The authors introduce a constraint programming approach to solve the static task allocation to distributed processors problem. Their algorithm can also prove non-existence of a solution when it cannot find one. Another interesting feature of their algorithm is that it separates the allocation problem from the scheduling problem. Their algorithm incorporates a method for learning from the schedulability analysis to remodel the allocation problem and improve performance. However, the only experiment results given in that work is with 40 tasks and scalability of that approach is not studied. Another constraint programming based approach for parallel execution of safety-critical applications is studied by Puffitsch et al. [2015]. The objective of that work is executing the tasks of Prelude applications on multi- and many-core architectures where the tasks are scheduled by non-preemptive offline scheduling. Our work differs from that work in the sense that we are doing the multi-core mapping of the blocks in a model which results in splitting a task into subtasks which runs on different cores where the tasks in a core are scheduled by a preemptive rate-monotonic scheduler. Cotton et al. [2011] discuss the use of SMT solvers and multi-criteria optimization for mapping tasks to multi processors. Application of SMT solvers in many-core scheduling for data parallel applications is discussed by Tendulkar et al. [2014]. Feljan and Carlson [2014] propose a heuristic, which utilizes information on how tasks delay each other, for finding a good solution for task allocation problems in a short solver execution time.

There are also several studies focusing on the parallelization of Simulink models. Kumura et al. [2012] propose methods to flatten Simulink models for parallelization without giving a detailed description of the optimization formulation. In that work,

Simulink blocks are considered as tasks. Canedo et al. [2010] introduce the concepts of strands for breaking the data dependencies in the model to achieve thread level parallelism in multi-core. The authors define a strand as a chain of blocks that are driven by Mealy blocks. The proposed method searches for available strand split points in Simulink models and it is heavily relying on strand characteristics in target models. Cha et al. [2011] have focused on automating code generation for multi-core systems where the parallel blocks are grouped by user-defined parallelization start and end S-functions into the model. An approach for worst-case execution time analysis of Simulink models is described by Kirner et al. [2000]. Although WCET analysis is crucial for going from Simulink models to executables on a multi-core system, we don't focus on WCET analysis in our work but we assume the WCET of the blocks are readily available as an input. A compiler level parallelization of code generated by Simulink is studied by Umeda et al. [2015]. The authors propose an automatic parallelization approach using the OSCAR compiler. While Umeda et al. [2015] propose a compiler level parallelization approach, we approach the problem at the model level. We believe that the model level parallelization provides a better architectural picture for control engineers since they can see the functional partitioning in the model. This helps the engineers to understand the parallel execution better and to debug easier.

Deng et al. [2015] study model-based synthesis flow from Simulink models to AUTOSAR [2015] runnables and runnables to tasks on multi-core architectures. The authors extend the Firing Time Automation (FTA) [Lublinerman and Tripakis 2008] model to specify activations and requested execution time at activation points. They define modularity as a measure of number of generated runnables and reusability as a measure of false dependencies introduced by runnable generation. The authors use modularity, reusability and schedulability metrics for evaluation of runnable generations. They also propose different heuristics and compare their results with the results obtained by utilizing a simulated annealing algorithm. Although that work is targeting a similar problem to our target problem for single-rate models, they are providing experiment results for systems with less than 50 blocks and they are not considering inter-core communication and memory overhead.

A study on multi/many-core execution of multi-rate Simulink models is done by Pagetti et al. [2014]. Authors describe an approach to execute multi-rate Simulink models on multi/many-core architectures. However, for this purpose the authors propose doing a translation from Simulink models to Prelude [Forget et al. 2010]. Our work differs from that since we focus on finding a mapping of blocks on the available CPU cores for meeting deadline and shared memory related constraints. We do code generation directly for execution on the target architecture while Pagetti et al. [2014] focus on translation to Prelude without seeking a feasible mapping of the input model/blocks to the target cores.

A linear programming approach for partition scheduling problem for strictly periodic tasks on multiprocessor integrated modular avionics (IMA) architectures is studied by Al Sheikh et al. [2012]. The authors incorporate available resource constraints like memory limitations along with IMA related constraints into their linear programming formulation. They are also proposing a game theory based, best-response algorithm as a heuristic which is proven to converge. Our work differs from that work by the heuristics we introduce and our model level approach which creates a parallelization before the code which forms the tasks is generated.

A scheduling methodology for multi-core systems is described by Elhossini et al. [2010], where directed acyclic graphs for the tasks are used for task partitioning. The approach in that work groups the tasks in a system as ordinary tasks and multi-rate tasks. The ordinary tasks are defined as the tasks that must be executed at every cycle of the system and the multi-rate tasks are defined as the tasks that do not need to be

executed at every cycle but the tasks that can be scheduled during idle times of the schedule. The problem targeted in that work differs from ours since in our problem tasks cannot be grouped as ordinary and multi-rate tasks as it is done in that work. Instead, we are dealing with a set of tasks consisting of blocks of a model design where all the tasks have different rates and must complete within their periods under an available scheduling algorithm.

The scalability of the constraint programming based approaches is studied by Gorcitz et al. [2015]. The authors experimentally showed that the constraint programming based approaches can be efficiently used for small to medium sized systems but they diverge rapidly when the system becomes larger. The authors also state the necessity of heuristic method for larger systems, which is along the lines of the experiment results in our work.

The use of conditional sporadic directed acyclic graph (DAG) tasks by Baruah [2015] have similarities to our approach for multi-rate models where execution order dependencies of the blocks, induced by the block-dependency graphs of different sample rates, change for different firing times with possible preemptions of the tasks at the firing times.

Our work mainly differs from the other works in literature by

1. providing a complete flow for automatically parallelizing single- and multi-rate Simulink models,
2. incorporating the communication time cost in the optimization problem both in the transmitter and the receiver side,
3. having total available shared memory and task priority constraints, and
4. being able to handle large models with more than 100 blocks in a reasonably short time using the proposed heuristics.

3. PROBLEM DESCRIPTION

3.1. Preliminaries

Model based design platforms like Simulink from The MathWorks Inc. [2015], SCADE Suite from Esterel Technologies [2015] and Ptolemy [Eker et al. 2003] utilize synchronous block diagrams for describing the model of a system. Figure 1 displays a sample block diagram from a simple Simulink model. A clear description of synchronous block diagrams is given by Lublinerman et al. [2009]. Here, we will give a brief summary of this description. A synchronous block diagram contains blocks (possibly inside other blocks), each with nonnegative number of input and output ports, and the connections between the blocks through their input and output ports. A block can either be macro or atomic. A macro block encapsulates a block diagram. An atomic block can be defined as a block that cannot be partitioned into smaller blocks. Flattening operation on block diagrams is to remove the hierarchy hidden inside macro blocks by replacing the macro blocks with the block diagrams they are encapsulating until no macro blocks are left. Macro blocks correspond to the virtual subsystems in Simulink. Details on the flattening operation can be found in [Lublinerman et al. 2009].

Every block in a synchronous block diagram has a sampling rate which describes the rate at which the block is executed during the execution of the system being modeled. If all the blocks have the same sampling rate, the model is referred as a single-rate model. Otherwise the model is referred as a multi-rate model.

For describing the problem that we are targeting, it is convenient to represent the dependencies between the blocks of a synchronous block diagram in a graph structure. For this, we first flatten the given block diagram. Figure 2 gives an example to a flattened block diagram where the “Subsystem” block in the Figure 1 is replaced with the block diagram it encapsulates.

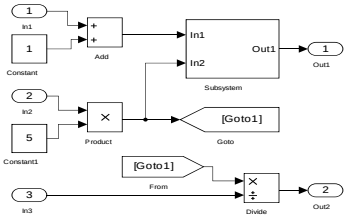

Fig. 1. Simulink Block Diagram

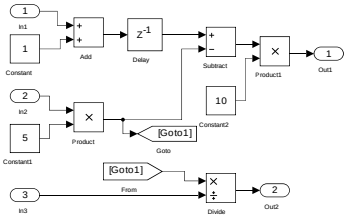

Fig. 2. Flattened Block Diagram

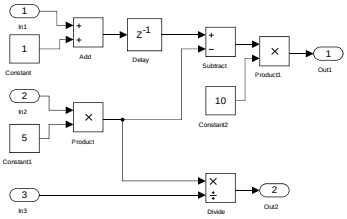

Fig. 3. Main Block Diagram

Model based design tools supply some blocks to the user for routing signals inside the diagram in a visually clear way. An example to this situation is the use of "Goto" and "From" blocks in a Simulink model instead of routing a line from its source to its destination. Since these blocks do not represent any operation done by the system that is being modeled, we call these blocks as virtual routing blocks and replace these blocks with lines representing the data connections between the blocks. Figure 3 gives an example of such a transformation. The "Goto" - "From" pair in Figure 2 is replaced with a line in Figure 3. After flattening and replacing the virtual routing blocks with lines we end up with a block diagram which contains input and output ports of the system, the blocks, where every block is atomic and represents an operation carried by the system, and the interconnections between the blocks. We call such a block diagram as a *main block diagram*.

Definition 1: A *block dependency graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a graph representation of a main block diagram. It is an acyclic directed graph with the vertex set $\mathbf{V} = \{v_i : i \in [1, n]\}$ where $|\mathbf{V}| = n$ and the set $\mathbf{E}$ of directed edges.

Figure 4 illustrates a sample block dependency graph which corresponds to the main block diagram given in Figure 3. Each vertex of a block dependency graph represents a block in the main block diagram. There is a one-to-one correspondence between the vertices in $\mathbf{V}$ and the blocks in the main block diagram. As a notation, we will use v_i in order to refer to the block which the vertex $v_i \in \mathbf{V}$ corresponds to. A directed edge $(i, j) \in \mathbf{E}$ is sourced from the vertex v_i and has the vertex v_j as its destination. Such an edge represents existence of a direct data connection from the output ports of the block v_i to the input ports of the block v_j. In a main block diagram, there can be data connections representing data transfers from previous iterations of their source blocks. We call the blocks with such incoming data connections as *delay introducing blocks*. The edge set $\mathbf{E}$ of a block dependency graph excludes such data connections. Every edge $(i, j) \in \mathbf{E}$ has an associated positive weight $c_{i,j}$ which represents the amount of the data transferred from the block v_i to the block v_j. When the blocks v_i and v_j are executed on different CPU cores, there will be a communication cost in terms of time for transferring $c_{i,j}$ amount of data between the CPU cores. The communication cost for such a connection is divided into transmission and reception parts where $tx_{i,j}$ denotes the time required for transmission part of the communication and $rx_{i,j}$ denotes the time required for

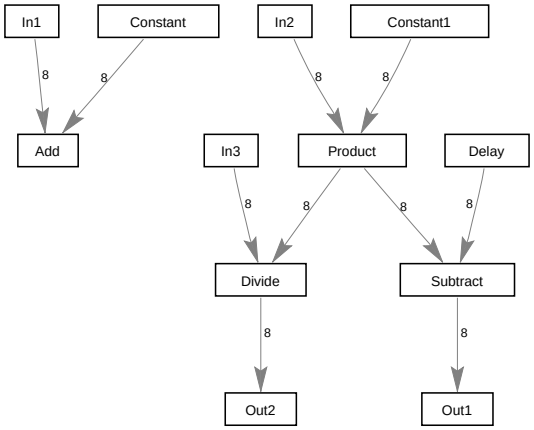

Fig. 4. A Block Dependency Graph

reception part of the communication. For
each block $v_i \in \mathbf{V}$, its worst-case execution time (WCET) is denoted by w_i and its sampling period is denoted by π_i.

3.2. Problem Definition

We are addressing the problem of automatically parallelizing synchronous block diagrams on to multi-core architectures to satisfy WCET constraints on the target platform. We have different problem definitions for single- and multi-rate models.

Assumptions 1:

(a) All of the CPU cores on the target platform are identical.
(b) The time cost for data communication between a pair of blocks on different CPU cores is identical for any pair of CPU cores.
(c) The time cost for transferring some data between a pair of blocks on the same CPU core is zero.
(d) There are no cycles in the given block dependency graph.
(e) The execution order dependencies of the blocks with the same period are only defined by the data dependencies between them.
(f) All the blocks with a period π become available to execute at every π amount of time and they all must complete their execution in π amount of time after they become available.

Assumptions 2:

(a) Distinct sampling periods in the model are harmonic.
(b) On the target platform, for each CPU core, there will be a separate task for each distinct period for the blocks with that period and mapped on that CPU core.
(c) On the target platform, tasks within a CPU core will be scheduled by a rate-monotonic scheduling algorithm.
(d) No protected resource is shared between the blocks with different sampling periods. Any resource sharing between the same-rate blocks is known in advance.

Problem 1 - Single-rate models: Given the number m of available CPU cores on the target platform and a block dependency graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ such that $\pi_1 = ... = \pi_n = \pi$ where $n = |\mathbf{V}|$, compute an optimal mapping of the blocks to the target CPU cores and an execution ordering of these blocks with an objective of minimizing the makespan of all the blocks within their period on the target platform. Report if no feasible solution can be found which allows a makespan shorter than the period π. Here, makespan can be defined as the overall completion time of execution of all blocks. We impose the Assumptions 1 on this problem.

In [Tuncali et al. 2015] we focused on the Problem 1 for single-rate embedded control applications which are modeled in Simulink. In this paper we extend the discussion to the multi-rate models.

Problem 2 - Multi-rate models: Given the number m of available CPU cores on the target platform and a block dependency graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ such that $\pi_i \neq \pi_j$ for some $v_i, v_j \in \mathbf{V}$, compute a mapping of the blocks to the target CPU cores and an execution ordering of these blocks so that execution of every block can be started and finished within its period on the target platform. We impose the Assumptions 1 and 2 on this problem.

3.3. Solution Overview

In this paper we are focusing on the Problems 1 and 2 for Simulink models. Our target platform is Qorivva MPC5675K-based evaluation board from Freescale Semiconductor Inc. [2015]. The processor is a dual-core 32-bit MCU which is targeting automotive

applications. The μC/OS-II from Micrium Inc. [2015] is ported to our target platform and a library to support Simulink code generation is devised for the platform by Bulusu [2014]. Simulink Coder [The MathWorks Inc. 2015] is used for code generation from the models. In multitasking mode, which is the case for multi-rate models, the Simulink Coder combines the blocks of the same rate into a task which is scheduled by a rate-monotonic scheduler on a single core. For this reason, our approach is based on combining blocks of same rate which are mapped on the same core in a single task. Priorities of these tasks increase as their sampling periods decrease. Each core on our target platform has a separate copy of μC/OS-II kernel. The multi-core operation is supported by utilizing inter-core synchronization and communication protocols through a shared memory space as described by [Bulusu 2014]. The operating system kernel on each core uses rate-monotonic scheduling algorithm for scheduling the tasks within the core. Preemption of the tasks is allowed.

An overview of our approach to the Problems 1 and 2 is illustrated in the Figure 5. Although our approach for these two problems have some differences that are explained later in this section, the illustration given in Figure 5 is applicable to both problems. In this section we will first explain our approach for the single-rate models and than extend the discussion to the multi-rate models.

We approach the Problem 1 which is for single-rate models in five steps as described in our previous paper [Tuncali et al. 2015]. **(1)** Creating a block dependency graph from the given block diagram. Description of a *block dependency graph* is given in Definitions 1. Task-data graphs are discussed by Cotton et al. [2011]. We use a similar approach using blocks instead of tasks, the worst-case execution times of blocks instead of the amount of work associated with tasks and using size of data communication between blocks. **(2)** Finding an optimal or near optimal mapping of blocks to different CPU cores by formulating a Mixed-Integer Linear Program (MILP) and solving the resulting optimization problem with off-the-shelf MILP solvers. Details of our MILP formulation are given in Section 4. **(3)** Automatically updating the original Simulink model by adding inter-core communication blocks where necessary in accordance with the most optimal solution. We handle inter-core data communications by utilizing available shared memory and inter-core semaphores which are used for synchronization between tasks across cores and protecting global critical sections as described in [Bulusu 2014]. For the purpose of utilizing this approach in Simulink, we model the transmission and reception of data with two separate S-function blocks which implement inter-core transmission and reception using inter-core semaphores and shared memory. We will refer to these S-function blocks as *inter-core communication blocks*. **(4)** Generating separate code for each target core. We first partition the

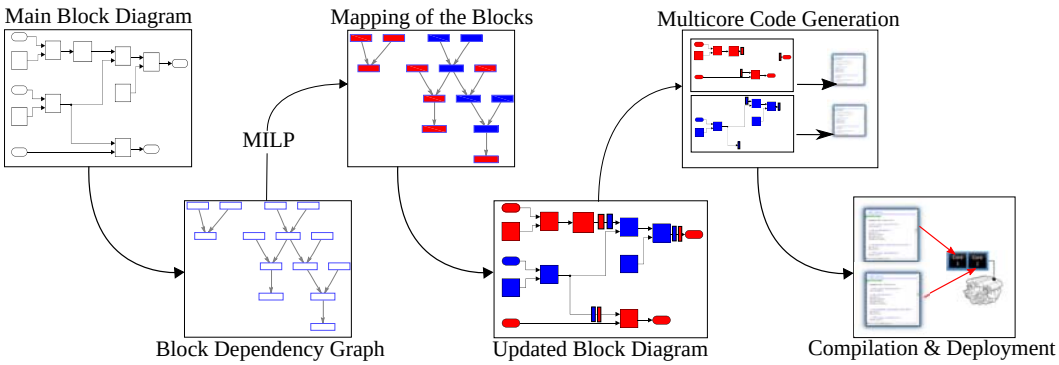

Fig. 5. An Overview of our Approach

model into separate models for each core. For this, we create a copy of the model for each core and automatically comment out the blocks that are not mapped to the corresponding core. Then, we do Simulink code generation from each these model copies. **(5)** Compiling the generated code and deploying it on the target platform. Since we do separate code generation for each core, we compile the generated code for each core and deploy each of the executables on its corresponding core.

Our approach for the Problem 2 which is for multi-rate models follows similar steps as our approach for the Problem 1 with modifications in some of these steps. Firstly, as well as creating a block dependency graph for the model as a whole, we also create a separate block dependency graph for each distinct sampling period in the model. In the second step, instead of optimizing with an objective function, we are utilizing MILP solvers for seeking a feasible solution to satisfy the worst-case execution time limits imposed by the periods of the blocks. For this purpose we first calculate a hyperperiod from the distinct periods in which a block may become available to execute more than once. In our MILP formulation, we are targeting to find a solution for one hyperperiod since the execution of the system on the target platform repeats itself at every hyperperiod. Details of our MILP formulation are described in Section 4. For the multi-rate models, since the transmitter and the receiver block of some communicated data can have different periods, communication between such blocks require rate-transition blocks. As a communication mechanism between different rate blocks on different cores, we utilize asynchronous three-slot mechanism described by Chen and Burns [1997]. Implementation details of this mechanism, which we will refer as *inter-core rate-transition blocks* on our target platform are explained by Bulusu [2014]. For the blocks with same rate which are executed on different cores, we use the same inter-core communication mechanism that we use in the third step of our approach to the Problem 1. Although there is no difference in the final two steps of our approach from our approach to the Problem 1, the generated code from a multi-rate Simulink model has a separate function for each distinct sample time. We place each of these functions in a separate task in the executables for the target CPU cores.

4. MILP FORMULATION

In this section we present our mixed integer linear programming formulation for the parallelization problem of synchronous block diagrams. An MILP formulation for single-rate models was described in our previous work [Tuncali et al. 2015]. In this paper we are extending this formulation for multi-rate models. Our MILP formulation is based on the formulations proposed by Yi et al. [2009], Bender [1996] and Ostler and Chatha [2007]. We introduce an extension to these formulations by dividing the cost of communication to the transmission and reception parts, by adding a constraint on usage of the available shared memory for inter-core communications and extending the formulation to address the multi-rate models. Our MILP formulation for multi-rate models seeks for a feasible solution that satisfies the constraints induced by the desired block periods and available shared memory on the target platform without having any objective function. In Subsection 4.5, we describe our heuristic techniques for reducing the number of constraints for allowing the MILP solvers to find better solutions within a feasible time.

The blocks mapped on the same core with the same period will be executed in the same task. So, there will be a separate task for each distinct sampling period of the blocks that are mapped to the same core.

Since our target platform uses rate-monotonic scheduling algorithm and allows preemption of task executions, a task can be preempted by another task of the same core which arrives later and contains blocks with smaller periods. Also, when low priority and high priority tasks are available at the same time, the tasks with lower priorities

will wait for higher priority tasks to either complete or get blocked. We have constraints for introducing these behaviors into our formulation with some exceptions. In our problem, a high priority task, say T, can be blocked when it is waiting data arrival from the tasks on different cores. In such a case, a lower priority, ready task can start its execution while the task T is blocked. This behavior is not in our MILP formulation which will assume no task will be executed in this period. So, even though the low priority task is computed to be executed after the high priority task according to our formulation, in the actual execution, it can start earlier. Another exception is related to the preemption of a task while it is executing a block. In our formulation, we assume that a block inside a task which starts execution must complete before it can be preempted by a higher priority task. When this may not be possible in the worst case, it should not start its execution until the tasks which can preempt it are completed. Although our formulation does not formulate it, in an actual execution, a block of low priority task can start its execution and be preempted in the middle of its execution. Because of these exceptions, our formulation is pessimistic for completion time of low priority tasks. Here our assumption is no two tasks on different cores with different periods share a protected resource. This assumption must hold for not experiencing any anomaly in the scheduling of the tasks as described by Graham [1969] and Thiele and Kumar [2015]. One can also force this execution ordering in the formulation to be reflected to the target platform by introducing synchronization using global (inter-core) semaphore structures. Details of such mechanisms are not discussed in this paper.

4.1. Notation and Constants

The notation used for the MILP formulation is given in the Table I. The notation given for the problem description in the Section 3 is also valid for the MILP formulation and for convenience we repeat this notation in the Table I.

The block dependency graph $\mathbf{G}$ must be acyclic as it is defined in the Section 3. Since algebraic loops are not allowed in Simulink, a main block diagram of a Simulink model cannot have cycles due to algebraic loops. However, the main block diagram of a Simulink model can have cycles which contain at least one (delay introducing) block which is introducing data dependencies to previous iterations of a model execution (e.g., Unit Delay, Memory, Integrator, etc). When creating a block dependency graph, we discard the incoming connections to the delay introducing blocks from their predecessor blocks. Since those discarded connections represent data dependencies to an earlier iteration of the execution, discarding them does not affect the dependency relations that we are formulating. However, since these deleted connections will not go into our formulation, if no precaution is taken, the formulation will not consider the inter-core communication between a delay introducing block and its predecessor block when they are mapped to different cores. In order to avoid this issue, we force any delay introducing block and its predecessor to be mapped on the same CPU core by introducing a constraint.

We do not allow different rate blocks to share protected resources. Because such a case can create scheduling anomalies. Same-rate blocks accessing shared resources are assumed to be known in advance. We add such blocks into the set Z as if there was a connection between these blocks. This forces such blocks to be on same core and consequently in the same task due to the constraints in the formulation.

For a single-rate model, all block periods are π, $r = 1$, $\mathbf{R} = \{R_1 = \pi\}$, $\mathbf{V^1} = \mathbf{V}$, $\mathbf{E^1} = \mathbf{E}$, $\mathbf{G^1} = \mathbf{G}$, $f = 1$, $\mathbf{H} = \pi$, $\mathbf{F} = \{0\}$ and $\mathbf{M} = \{v_{i,0} : v_i \in \mathbf{V}\}$.

4.2. Variables

Variables used in the MILP formulation can be listed as follows:

$b_{i,p}$: A boolean variable indicating whether the block v_i is mapped to the core P_p or not. It is defined for all $v_i \in \mathbf{V}$ and for all $P_p \in P$. If v_i is mapped to core P_p, then $b_{i,p}$ takes value 1. If v_i is mapped to another core, then $b_{i,p}$ takes value 0.

$d_{i,j}$: A boolean variable indicating whether the block v_i executes before or after the block v_j when both blocks are mapped to the same core. It is defined for all pairs of same period blocks which do not have any data dependency to each other which means, for all i, j such that $(i, j) \notin \mathbf{E}$, $i < j$ and $v_i, v_j \in \mathbf{V^k}$ where $R_k \in \mathbf{R}$. If v_i executes before v_j, then $d_{i,j}$ takes value 1 and if v_i executes after v_j, then $d_{i,j}$ takes value 0 when these blocks are mapped to the same core. The variable $d_{i,j}$ does not have a meaning when the blocks v_i and v_j are mapped to different cores.

$d'_{i\_c,j}$: A boolean variable indicating whether a block finishes its execution before it can be preempted by executions of blocks with smaller period in an upcoming firing time. It is defined for all blocks $v_{i,c} \in \mathbf{M}$ and $F_j \in \mathbf{F}$ such that $c \cdot \pi_i < F_j < (c+1) \cdot \pi_i$. The variable $d'_{i\_c,j}$ takes value 1 if $v_{i,c}$ finishes its execution before the firing time F_j. It takes value 0 if $v_{i,c}$ starts its execution after all of the blocks in $M^{k,z}$ that are mapped to same core with v_i finish their execution where k, z satisfy $R_k < \pi_i$, $(z \cdot R_k) = F_j$.

$s_{i,c}$: The start time for the execution of the c^{th} repetition of the block v_i, i.e, $v_{i,c}$. It is defined for all $v_{i,c} \in \mathbf{M}$.

Table I. Notation used in MILP formulation

Notation	Description
$\mathbf{G} = (\mathbf{V}, \mathbf{E})$	A graph representation of the main block diagram. Namely a block dependency graph, which is a directed acyclic graph
n	The number of blocks in the main block diagram
$\mathbf{V} = \{v_i : i \in [1, n]\}$	The vertex set, where each vertex corresponds to a block in the main block diagram
w_i	The worst-case execution time of the block v_i
π_i	The sampling period of the block v_i
$\mathbf{E}$	The directed edge set. Where $(i, j) \in \mathbf{E}$ corresponds to the data connection from the block v_i to v_j excluding the connections to the delay introducing blocks
$c_{i,j}$	The amount of data transferred from the block v_i to the block v_j
$tx_{i,j}$ / $rx_{i,j}$	The time required for transmission / reception part of the communication from v_i to v_j when the blocks are executed on different cores
$\mathbf{Z}$	Set of the connections to delay introducing blocks which are not included in E
$\mathbb{N}$	The set of natural numbers
m	The number of available CPU cores
$\mathbf{P} = \{P_i : i \in [1, m]\}$	The set of available CPU cores
r	The number of distinct periods
$\mathbf{R} = \{R_i : i \in [1, r]\}$	The set of distinct sampling periods (“sample time” in Simulink)
r_j	the number of blocks with period R_j
$\mathbf{V^j} = \{v_i^j : i \in [1, r_j]\}$	The set of blocks which has a sampling period of R_j where $\mathbf{V^j} \subseteq \mathbf{V}$ and $\bigcup_{j=1}^{r} \mathbf{V^j} = \mathbf{V}$
$\mathbf{G^j} = (\mathbf{V^j}, \mathbf{E^j})$	The induced subgraph of $\mathbf{G}$ on the vertex set $\mathbf{V^j}$
$\mathbf{H}$	The hyperperiod, which is the least common multiple of all distinct periods in $\mathbf{R}$
firing time	Start of each period and its repetitions in the time interval $[0, H)$
$\mathbf{F} = \{c\gamma : c\gamma < \mathbf{H}, \gamma \in \mathbf{R}, c \in \mathbb{N}\}$	The set of distinct firing times
f	The number of distinct firing times
$v_{i,c}$	c^{th} repetition of a block $v_i \in \mathbf{V}$ in the hyperperiod, where $c \in \mathbb{N}$ and $c\pi_i < \mathbf{H}$
$\mathbf{M} = \{v_{i,c} : v_i \in \mathbf{V}, c \in \mathbb{N}, c\pi_i \in \mathbf{F}\}$	The set of the repetitions of the blocks
sSize	The size of a global semaphore structure in bytes
cCount	The number of copies of the data communicated inside the inter-core rate-transition structure
totMem	Size of the total available shared memory in bytes
aSize	Data alignment size in bytes (word size)
MAX	A very large constant which is used in the program formulation to dominate other terms allowing constraints to be ignored under certain conditions (big-M method)

4.3. Constraints

— A block shall be assigned to a single core

$$\forall v_i \in \mathbf{V},\ \sum_{p=1}^{m} b_{i,p} = 1 \tag{1}$$

— Delay introducing blocks and their predecessor blocks (or protected resource sharing blocks) shall be assigned to the same core

$$\forall z_{i,j} \in \mathbf{Z} \wedge \forall P_p \in \mathbf{P},\ b_{i,p} - b_{j,p} = 0 \tag{2}$$

— Start time of every repetition of a block shall be greater than or equal to its firing time

$$\forall v_{i,c} \in \mathbf{M},\ s_{i,c} \geq c \cdot \pi_i \tag{3}$$

— Execution of every repetition of a block shall be completed within the block's period

$$\forall v_{i,c} \in \mathbf{M},\ s_{i,c} + w_i \leq (c+1) \cdot \pi_i \tag{4}$$

— If there is a data connection from a block v_i to a block v_j where both blocks have the same period, then block v_j shall not start execution until **(i)** block v_i finishes execution and transmission of its output data to its successor blocks that are mapped on other cores and **(ii)** block v_j finishes receiving all of its input data that are sent by the blocks on other cores

Considering that the block v_i is mapped to the core P_p and v_j is mapped to the core P_q where p can be equal to q,

$$\forall P_p, P_q \in \mathbf{P}, \forall v_{i,c}, v_{j,c} \in \mathbf{M}\ \textit{s.t.}\ (i,j) \in \mathbf{E},\ v_i, v_j \in \mathbf{V^k},\ \pi_i = \pi_j = R_k \in \mathbf{R},$$

$$s_{i,c} + w_i + \sum_{v_x \in \mathbf{V^k}} [tx_{i,x}(1 - b_{x,p})] \leq s_{j,c} - \sum_{v_y \in \mathbf{V^k}} [rx_{y,j}(1 - b_{y,q})] + (2 - b_{i,p} - b_{j,q}) \cdot MAX \tag{5}$$

— Execution of the independent blocks with the same period that are mapped to the same core shall not overlap

Considering an independent pair of blocks v_i and v_j, are mapped to the core P_p, we have two different constraints for this requirement.

$$\forall P_p \in \mathbf{P}, \forall v_{i,c}, v_{j,c} \in \mathbf{M}\ \textit{s.t.}\ (i,j) \notin \mathbf{E},\ v_i, v_j \in \mathbf{V^k},\ \pi_i = \pi_j = R_k \in \mathbf{R},$$

$$s_{i,c} + w_i + \sum_{v_x \in \mathbf{V^k}} [tx_{i,x}(1 - b_{x,p})] \leq s_{j,c} - \sum_{v_y \in \mathbf{V^k}} [rx_{y,j}(1 - b_{y,p})] + (3 - b_{i,p} - b_{j,p} - d_{i,j}) \cdot MAX \tag{6}$$

$$s_{j,c} + w_j + \sum_{v_y \in \mathbf{V^k}} [tx_{j,y}(1 - b_{y,p})] \leq s_{i,c} - \sum_{v_x \in \mathbf{V^k}} [rx_{x,i}(1 - b_{x,p})] + (2 - b_{i,p} - b_{j,p} + d_{i,j}) \cdot MAX \tag{7}$$

Since MAX is a very large constant, (6) will be valid when block $v_{i,c}$ executes before $v_{j,c}$ i.e., when $d_{i,j} = 1$ and (7) will be valid when block $v_{i,c}$ executes after $v_{j,c}$ i.e., when $d_{i,j} = 0$.

— A block shall **(i)** either finish execution and output transmission before an upcoming firing time where smaller period block executions are fired **(ii)** or start execution after all the blocks with a smaller period in an upcoming firing time are finished

$$\forall P_p \in \mathbf{P}, \forall v_{i,c}, v_{j,c'} \in \mathbf{M}, \forall F_w \in \mathbf{F}\ \textit{s.t.}\ c\pi_i < F_w < (c+1)\pi_i$$

$$\textit{where}\ \pi_i = R_k \in \mathbf{R}\ \textit{i.e,}\ v_i \in \mathbf{V^k}, \pi_i > \pi_j = R_l \in \mathbf{R}\ \textit{i.e,}\ v_j \in \mathbf{V^l}, F_w = c'\pi_j,\ c' \in \mathbb{N},$$

$$s_{i,c} + w_i + \sum_{v_x \in \mathbf{V^k}} [tx_{i,x}(1 - b_{x,p})] \leq F_w + (2 - b_{i,p} - d'_{i\_c,w}) \cdot MAX \tag{8}$$

$$s_{j,c'} + w_j + \sum_{v_y \in \mathbf{V^l}} [tx_{j,y}(1 - b_{y,p})] \leq s_{i,c} - \sum_{v_x \in \mathbf{V^k}} [rx_{x,i}(1 - b_{x,p})] + (2 - b_{i,p} - b_{j,p} + d'_{i\_c,w}) \cdot MAX \tag{9}$$

Since MAX is a very large constant, (8) will be valid when block $v_{i,c}$ executes before F_w i.e., when $d'_{i\_c,w} = 1$ and (9) will be valid when block $v_{i,c}$ executes after low period blocks fired at F_w i.e., when $d'_{i\_c,w} = 0$.

Considering the execution order between a block and an upcoming firing time, the following constraint must also be added to make sure that if the parameter $d'_{i\_c,w}$ is 0 (block $v_{i,c}$ executes after F_w), then $d'_{i\_c,\bar{w}}$ is also 0 for all $F_{\bar{w}} < F_w$ (block $v_{i,c}$ executes after $F_{\bar{w}}$).

$$\forall v_{i,c} \in \mathbf{M}, \forall F_w, F_{\bar{w}} \in \mathbf{F} \textit{ s.t. } c\pi_i \leq F_{\bar{w}} < F_w < (c+1)\pi_i, F_w = c'\pi_j,\ F_{\bar{w}} = c''\pi_k$$
$$\textit{where } \pi_i > \pi_j,\ \pi_i > \pi_k,\ \pi_i, \pi_j, \pi_k \in \mathbf{R},\ c', c'' \in \mathbb{N},$$

$$d'_{i\_c,\bar{w}} < d'_{i\_c,w} \tag{10}$$

— Blocks shall not start execution until other blocks with smaller periods from same or previous firing times finish their execution and transmission of their outputs when these blocks are mapped to the same core

$$\forall P_p \in \mathbf{P}, \forall v_{i,c}, v_{j,k} \in \mathbf{M}, F_w \in \mathbf{F} \textit{ s.t. } F_w = c'\pi_j \leq c\pi_i < (c'+1)\pi_j$$
$$\textit{where } c, c' \in \mathbb{N}, \pi_i = R_k \in \mathbf{R} \textit{ i.e, } v_i \in \mathbf{V^k} \textit{ and } \pi_i > \pi_j = R_l \in \mathbf{R} \textit{ i.e, } v_j \in \mathbf{V^l},$$

$$s_{j,c'} + w_j + \sum_{v_x \in \mathbf{V^k}} [tx_{j,x}(1 - b_{x,p})] \leq s_{i,c} - \sum_{v_y \in \mathbf{V^l}} [rx_{y,i}(1 - b_{y,p})] + (2 - b_{i,p} - b_{j,p}) \cdot MAX \tag{11}$$

— Since the inter-core communications are done over the available limited shared memory space, the total memory needed for semaphores and communication buffers shall be less than or equal to total amount of available shared memory

$$\sum_{p=1}^{m} \Bigg[\sum_{(i,j) \in \mathbf{E},\ \pi_i = \pi_j} \left[\left(sSize + \left\lceil \frac{c_{i,j}}{aSize} \right\rceil \cdot aSize \right) \cdot |b_{i,p} - b_{j,p}| \right] + \sum_{(k,l) \in \mathbf{E},\ \pi_k \neq \pi_l} \left[\left(sSize + cCount \cdot \left\lceil \frac{c_{k,l}}{aSize} \right\rceil \cdot aSize \right) \cdot |b_{k,p} - b_{l,p}| \right] \Bigg] < totMem \tag{12}$$

4.4. Objective Function

For single-rate models (*i.e,* $\pi_i = \pi, \forall v_i \in \mathbf{V}$), we are focusing on the Problem 1 where the goal is to minimize the makespan for one iteration of the model execution. For this purpose, the objective function for the MILP problem is to minimize π as described in our previous paper [Tuncali et al. 2015].

For multi-rate models, i.e, $\pi_i \neq \pi_j$ for some $v_i, v_j \in \mathbf{V}$, we do not impose any objective function on the MILP formulation as it already includes the scheduling constraints. Since single-rate models can be considered as a special case of the multi-rate models, one can also use the multi-rate approach for parallelization of single-rate models. With

this approach, instead of seeking the minimum makespan, one can seek a feasible mapping for a predetermined makespan which can also help in decreasing the solver execution time for single-rate models.

4.5. Improving Solver Execution Time

In this section, we introduce our heuristic techniques for dealing with large models. The heuristics described in this section are targeting single-rate models, i.e, Problem 1. These heuristics can be adapted for Problem 2 as well. That is, for multi-rate models, they can be by applied only to the blocks of the same rate. However, we do not expect them to be as useful as they are for Problem 1 since they can eliminate opportunities to fit blocks with lower rates in between the completion time of the higher rate blocks and the upcoming firing times.

For two same-rate blocks, if there exists a directed path between the corresponding vertices in the block dependency graph, we say these blocks are *dependent* to each other. Otherwise we say these blocks are *independent* to each other.

The majority of the constraints in the MILP formulation are related to execution ordering of the independent blocks, i.e, the inequalities (6) and (7). This is because, for blocks dependent to each other, execution of each block has constraints related the directly connected blocks. These relations also impose lower and upper bound for execution time of the blocks and limit the search space for each block's execution. So, the solver execution time is not dramatically effected by increasing number of dependent blocks. However, there are constraints between each pair of independent blocks. This combinatorial relations result in quadratically increasing number of constraints as the model size increase for most practical models. The heuristics we introduce are targeting to decrease the number of or completely eliminate these constraints for independent blocks. These heuristics impose some restrictions on execution ordering of the blocks. So, given infinite time, the solver will be seeking a less optimal solution. However, since we limit the solver execution time, it becomes very hard, if not impossible, to find an optimal solution without heuristics and even though it is not optimal, the solver can find better solutions with the heuristics.

4.5.1. Partially ordering independent blocks. In order to have more parallelization opportunities in a model, the main block diagram must preferably have a large number of blocks that are independent to each other. Typically, in an industrial size model with a large number of blocks, both the number of blocks that are independent to each other and the number of blocks that are dependent to each other are large. However, if the number of blocks that are independent to each other is very large, when we consider all possible combinations of execution orders between these independent blocks, the number of constraints introduced by inequalities (6) and (7) becomes very large. As a consequence, finding an optimal solution within a feasible time becomes harder.

We address this problem by deciding the execution order between certain pairs of independent blocks, say v_i, v_j, in advance. That is, before formulating the optimization problem, we decide the values of the $d_{i,j}$ variables for these block pairs. Since the $d_{i,j}$ variables become constants in this case, the MILP solver does not need to seek their values. Also, since at least one of the inequalities (6) and (7) in the constraints become invalid (satisfied always), the number of constraints the MILP solver must try to satisfy decreases. Note that, our execution order decision is valid only when these blocks are mapped onto the same core. So this should not prevent these blocks to be mapped on different cores and, hence, be executed in a different order than what we specify.

Our partially ordering heuristic is based on comparing the execution start time frames of independent blocks. The execution start time frame of a block is de-

fined as the time frame between its best and worst-case start time values. The best and the worst-case start time values of a block $v_i \in \mathbf{V}$ are defined in the subsection 4.2 as bs_i and ws_i respectively. The variable bs_i is determined by using the best case completion time for all of the blocks corresponding to the vertices from which there exists a path to $v_i \in \mathbf{V}$ in $\mathbf{G}$. In the best case, all of this workload before the block v_i is distributed equally on all of the cores. The best case start time of v_i is calculated as $bs_i = \big(\sum_{k \in Y_i} w_k\big)/m$ where $Y_i = \{v_k : v_k \in \mathbf{V}, \pi_k = \pi_i$ *and there exists a path from* v_k *to* v_i *in* $\mathbf{G}\}$. The variable ws_i is determined by using the best case completion time for all of the blocks corresponding to the vertices to which there is a path from $v_i \in \mathbf{V}$ in $\mathbf{G}$ and the worst-case execution time of the block v_i itself, subtracted from the deadline. The worst-case start time of v_i is calculated as $ws_i = \pi_i - \big(w_i + \sum_{k \in Y_i} w_k\big)/m$ where $Y_i = \{v_k : v_k \in \mathbf{V}, \pi_k = \pi_i$ *and there exists a path from* v_i *to* v_k *in* $\mathbf{G}\}$.

For all independent block pairs $v_i, v_j \in \mathbf{V}$, if $\big((bs(i) \leq bs(j)) \wedge (ws(i) < ws(j))\big) \vee \big((bs(i) < bs(j)) \wedge (ws(i) \leq ws(j))\big)$ then we decide v_i to execute before v_j and set $d_{i,j}$ to 1. Else if $\big((bs(i) \geq bs(j)) \wedge (ws(i) > ws(j))\big) \vee \big((bs(i) > bs(j)) \wedge (ws(i) \geq ws(j))\big)$ then we decide v_i to execute after v_j and set $d_{i,j}$ to 0. If we compute $d_{i,j}$ as 1, then we replace the $d_{i,j}$ in the equation (6) with 1 and do not add the constraint given by the equation (7) into our formulation. Similarly, if we compute $d_{i,j}$ as 0, then we replace the $d_{i,j}$ in the equation (7) with 0 and do not add the constraint given by the equation (6) into our formulation.

4.5.2. Fully ordering independent blocks. Even though ordering independent blocks using the partially ordering heuristic improves the performance, this may not be enough for models with very large number of blocks. For example we could not find a feasible solution to models with more than 100 blocks with this approach. For dealing with those large models we propose deciding the execution order of all the independent blocks when they are mapped on the same core. The logic in fully ordering heuristic is based on comparing the midpoints of the execution start time frames for these blocks. For independent blocks $v_i, v_j \in \mathbf{V}$, if $(bs_i + ws_i)/2 < (bs_j + ws_j)/2$ then we decide v_i to be executed before v_j and vice versa if otherwise. So, we replace $d_{i,j}$ values in the equations (6) and (7) to the calculated values and do not add the constraint defined by the equation (6) when $d_{i,j}$ is 0 and similarly do not add the constraint defined by the equation (7) when $d_{i,j}$ is 1 into our formulation. Here the decided value for $d_{i,j}$ determines the ordering of the blocks only when they are mapped to the same core and our discussion on the case when these blocks are mapped to different cores in the previous subsection is still valid.

4.5.3. Merging highly coupled blocks. In this heuristic we merge blocks v_i and v_j when block v_j is the only block connected to the output port(s) of block v_i and block v_i is the only block connected to the input port(s) of block v_j. The merging operation copies all incoming and outgoing edges of v_j to v_i except the edge (i, j) between these blocks. Then it updates w_i with $w_i + w_j$ and finally deletes v_j.

4.5.4. Merging small blocks with large blocks. In this heuristic we merge blocks v_i and v_j based on their ratio of execution times. If block v_j is the only block connected to the output port(s) of block v_i and the WCET of block v_i is very small when compared to the WCET of block v_j, then block v_i is merged into block v_j. If block v_i is the only block connected to the input port(s) of block v_j and the WCET of block v_j is very small when compared to the WCET of block v_i, then block v_j is merged into block v_i. We find this technique useful for reducing the number of blocks of concern in a way that parallelization will be focused on blocks with higher impact on execution time. The

ratio between the worst-case execution times of the blocks for determining a merge operation can be defined depending on how much reduction is needed in the number of blocks.

The merging methods described above can be used for decreasing the number of nodes in very large models where the MILP solver can no more find a good solution. These two techniques are also dependent on the structure of the model. Although, in general, they assist in finding better solutions, there can be cases where the number of nodes cannot be reduced to an acceptable level.

5. IMPLEMENTATION

In this section we describe the details of the implementation of our tool in MATLAB. Some of the concepts explained here are described earlier in Section 3.

Our tool accepts as an input block diagram of a Simulink model that is ready to compile. The user can as well input the desired depth of blocks to be parallelized. The desired depth sets an upper bound on the hierarchical depth for the flattening operation done on the model block diagram. Even though the desired depth is set, if there is a delay introducing block which has a larger depth, it will still be discovered and flattening will be done in order to remove hierarchy for such a block. This is because a block containing such a delay introducing block can cause a cycle in the block diagram which must be handled. In Simulink, *public* "Goto"-"From" block pairs can create data dependencies between the blocks in different hierarchical levels. So, for the subsystems which have a public virtual ("Goto"-"From") connection with any block outside the subsystem, the user defined desired depth is ignored and flattening is done. Since determining the WCET of each block is not in scope of this paper, we assume that the WCET for each of the blocks are already determined and given as an input to the tool.

The first step in our approach is to create a block dependency graph from the given model block diagram. Our tool loads the model block diagram, reads specific block information, e.g., block type, parents, sample time etc., and all the relations between blocks along with the width and size of the data on the ports. For data types that are not built-in, the user input is required to define the data size in bytes. The block diagram is flattened by taking blocks inside sub-systems out of their parent blocks and by discarding the remaining blocks like input and output ports of subsystems and the emptied subsystem container blocks. Simulink provides virtual routing blocks like 'Goto' and 'From'. These virtual routing blocks serves for the purpose of virtually adding a line between the blocks. Thus these blocks do not really perform an operation and they are only virtual blocks that help the designer to create connections between the blocks in a cosmetically nice way. Our tool discards these virtual routing blocks and considers them as regular lines connecting the blocks in the model. After these operations we end up with having the main block diagram of the model as illustrated in Figure 3.

Our tool then creates a directed graph representation of the main block diagram. The vertices of this directed graph correspond the blocks in the main block diagram and the directed edges correspond to the connections in the main block diagram, directed from a block to another. Then the tool removes the incoming edges to delay introducing blocks. The delay introducing blocks contain states that are initialized to a value and in an iteration (including the first iteration), their outputs do not depend on their current inputs but only to their internal states. Hence, removing the incoming edges of such blocks remove any possible cycles without violating execution order dependencies. An issue arising here is that since we remove these edges, the data connection from a block to a delay introducing block is no more considered and we can fail to model an inter-core communication if a delay introducing block and its predecessor block are mapped to different cores. For resolving this issue we keep track of such blocks for forcing them

to be mapped on the same core. Finally we have a block dependency graph of the main block diagram as given in Figure 4.

In the case of applying the merge based heuristics given in Subsection 4.5 the merge operations are done on the block dependency graph in order to obtain a smaller sized graph. The details of the merge operations are described in the related subsection.

The block dependency graph and the number of CPU cores on the target architecture are used in generating the MILP formulation presented in Section 4. Our tool takes the MILP solver to be used as an input option and executes this solver with an upper bound on solver execution time which again is taken from the user as an input option. For the multi-rate models, the tool generates the block dependency graphs for every distinct sample rate value. The block dependency graph for a sample rate contains the blocks with the same sample rate and the data dependencies between these blocks. Since the rate-transition blocks have one sample rate for their inputs and another sample rate for their outputs, they are added to both sample rate graphs. The WCET of the rate-transition blocks may be different for different sample rates. Here, the sample rate values can be supplied by the user or read from the properties of the blocks in the input block diagram. When there are more than one sample rates, the problem must be solved to satisfy the constraints for all tasks in a hyperperiod. For this purpose, the hyperperiod is calculated as the least common multiple of the distinct sample periods. During an hyperperiod, a task containing the blocks with a period will be repeated $\frac{hyperperiod}{period}$ times. Starting time of each of these repetitions in the hyperperiod is called a *firing time*. The tool calculates the firing times and determines the periods and the blocks corresponding to each firing time. The block dependency graphs for the whole model and for the distinct period values together with the hyperperiod, the firing times and the periods/blocks corresponding to the firing times are used in the MILP formulation described in the Section 4. The MILP solver seeks a feasible mapping of blocks to the CPU cores satisfying the constraints and returns the mapping found and execution start times for all executions of the blocks during the hyperperiod. The MILP solver returns the solution for mapping blocks to the available CPU cores and the execution order between these blocks, if a feasible solution is found. If no feasible solution is found, MILP solver reports this and our tool as well reports this to the user and exits.

The solution from the MILP solver is used to add inter-core communication blocks between the blocks with the same period which are mapped on different CPU cores. The relevant outputs of a block which are sending data to a block on a different core are connected to inter-core data transmitting S-function blocks. Similarly, the corresponding inter-core data receiving S-function blocks for each transmitter are connected to the relevant inputs of the block which is receiving data on a different core. The inter-core communication blocks are added by setting unique IDs that set each pair of transmitting and receiving blocks to use a dedicated inter-core semaphore and a dedicated shared memory location. For the multi-rate models, the inter-core rate-transition blocks which are stated in the Section 3 are added between the different rate blocks mapped on the different cores which have a data communication between each other.

An example of the transformation of block diagram for inter-core communication of the same rate blocks is given in Figure 6. The output of B1 is connected to the input of B2 in the original model. This connection is then replaced by the inter-core communication blocks. After adding all needed communication blocks, we set the priority attributes of the Simulink blocks using the execution start time values obtained from the optimization solution. This is done in order to guarantee that, for every iteration in the sampling process of the Simulink model, the total order of block execution induced by Simulink is consistent with the partial order of the block dependency graph that we

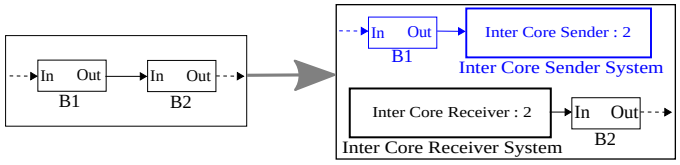

Fig. 6. Inter-core communication blocks

create which imposes the MILP formulation constraints. Simulink Coder uses the priority attributes of the blocks, with respect to the other blocks in the same subsystem, as an execution ordering which is reflected to the generated code as long as the priority settings do not conflict with the data dependencies between the blocks. Because of this behavior of the Simulink Coder, depending on the selected model depth for the parallelization, our tool may be implicitly suggesting a different structure for the model, i.e, splitting subsystems into more than one subsystems, through execution ordering of the blocks. While this can be automated, our tool does not change the structure of the model automatically and leaves this to the control engineer.

As the final step, a copy of the model is created for every CPU core. Each copy of the model corresponds to a CPU core and the blocks which are mapped on other cores are commented out. Code generated from each of these models can be compiled to create separate executables for each core. For the multi-rate models, Simulink generates separate functions for different sample periods. These functions are executed in different tasks and these tasks are scheduled by the rate-monotonic scheduling algorithm on the target platform as described in [Bulusu 2014].

6. EXPERIMENTS

For studying the scalability and efficiency of our approach for the single-rate models, we utilize randomly generated directed acyclic graphs with different number of nodes. We present results of these experiments in the subsection 6.1 and results of our case studies for single-rate models in the subsection 6.3. We illustrate our approach for multi-rate models with a simple example in the subsection 6.2. We use SCIP [Achterberg 2009] from Achterberg as MILP solver which is interfaced with MATLAB through the Opti Toolbox [Currie and Wilson 2012] by Currie and Wilson. Experiments are run on a 64-bit Windows 7 PC with Intel Xeon E5-2670 CPU and 64 GB RAM.

6.1. Randomly Generated DAGs

For evaluating performance of our approach for single-rate models, we generate random DAGs in which the WCET, communication costs and connections between blocks are assigned randomly. We used the random DAG generator tool provided by Gwinner [2011]. Then we solve the Problem 1 for a dual-core system with the basic MILP formulation which is given in Section 4 and with the partially and fully ordering heuristics for deciding the execution order of independent blocks. We set five hours (18,000 sec) as an acceptable upper time limit for the solver run time. We have done 500 runs with different size and completely random DAGs for increasing the confidence level in the benchmarks. Here, we present a comparison of the performance of these three approaches in terms of the average speed-up achieved, the average solver execution time and the ability to find a solution in the given time limit. The speed-up is computed as the overall single-core worst-case execution time of the model divided by the overall worst-case execution time of the parallelized model.

Given infinite solver execution time, the basic MILP formulation is expected to find more optimal solutions than the other approaches do for any problem size. However, when the solver execution time is limited (5 hours in our experiments), it fails to find

satisfactory solutions for large problems. Table II gives a comparison of the performance of the used approaches. Average speed-up achieved by basic MILP formulation, partially and fully ordering heuristics (respectively denoted as *basic*, *partial* and *full*) and corresponding solver run-time values are presented in the table for different problem sizes. We also present the ratio of the solutions found over all the experiments. For a problem size, the lines corresponding to the approaches which could not return any solutions are discarded in the table. As it can be seen from the results presented in Table II, as the number of blocks in a model increases, any heuristic that (partially) sets the execution order performs better both in terms of solver run-time and optimality of solutions. According to our observations, for finding an optimal mapping, the basic MILP formulation performs best when there are less than 30 blocks. The partially ordering heuristic performs best when there are 30 to 50 blocks. For more than 50 blocks in the model, the fully ordering heuristic outperforms other approaches in terms of the achieved speed-up and the ability to return a solution. The basic MILP formulation fails to return any solution for models with 70 or more blocks. The partially ordering heuristic fails to return any solution for models with more than 110 blocks. Although this detail is not illustrated in Table II because of averaging, according to our experimental results, the fully ordering heuristic can occasionally achieve very low speed-up values compared to the other approaches when there are less than 20 blocks in the model. However, this issue is not observed when there are large number of blocks. This behavior is parallel to our expectations since optimization can significantly reduce the effect of possible non-optimal execution order decisions by trying large number of different mapping of blocks to different cores.

In Figure 7, we illustrate the comparison between the two heuristics and the basic MILP formulation in terms of the achieved speed-up over the number of nodes. The solid lines in the plot represent how much average speed-up is achieved by each approach. The dashed lines represent the corresponding minimum and maximum speed-up for each approach. For very small number of nodes, the basic MILP formulation is better than the other approaches. However, when the number of nodes increases, first, the partially ordering heuristic and, then, the fully ordering heuristic perform best.

In Figure 8, we illustrate the comparison between the two heuristics and the basic MILP formulation in terms of the average solver execution time over the number of nodes. Each line in the graph represents the average solver execution time spent for each approach. As it is expected, due to the time limit given to the solver, as the number of nodes increases, the solution times for all approaches converge. However, the experiments on models with less number of nodes suggests that the proposed heuristics can shorten the solver execution time. In the graph it can be observed that the average solver execution time for proposed heuristics (as a function of node count) is smaller than the basic formulation. Combining the data in Figure 7 and Figure 8, we

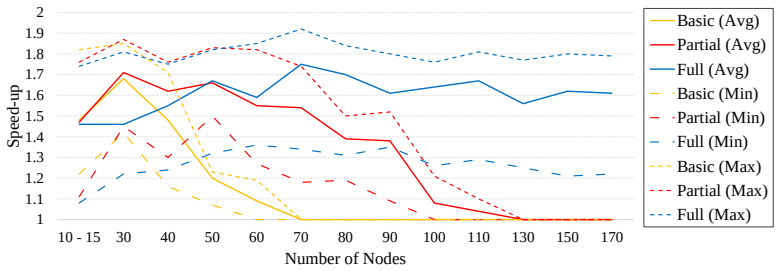

Fig. 7. Comparison of speed-up values between different approaches

Table II. Comparison of different approaches

# Nodes	Approach	Average speed-up	Average solver time (seconds)	% found Solutions
10-15	Basic	1.48	2	100%
	Partial	1.47	1	100%
	Full	1.46	0.5	100%
30	Basic	1.68	2620	100%
	Partial	1.71	1558	100%
	Full	1.46	26	100%
40	Basic	1.48	9256	100%
	Partial	1.62	2091	100%
	Full	1.55	606	100%
50	Basic	1.2	18000	100%
	Partial	1.66	12481	100%
	Full	1.67	5174	100%
60	Basic	1.09	18000	64%
	Partial	1.55	17400	100%
	Full	1.59	11685	100%
70	Partial	1.54	18000	100%
	Full	1.75	18000	100%
80	Partial	1.39	18000	100%
	Full	1.7	18000	100%
90	Partial	1.38	18000	60%
	Full	1.61	18000	100%
100	Partial	1.08	18000	50%
	Full	1.64	18000	100%
110	Partial	1.04	18000	30%
	Full	1.67	18000	100%
130	Full	1.56	18000	100%
150	Full	1.62	18000	100%
170	Full	1.61	18000	100%

can see that the fully ordering heuristic returns better solutions within shorter solver run-time compared to the other approaches.

6.2. An example on multi-rate models

We illustrate our approach for multi-rate models on a simple synthetic example. The Figure 9 illustrates the block diagram of the sample model. We included WCET and sampling period information for each block in the Figure 9. The notation for this information is in the format “block name, (WCET:sampling period)”. For instance the block “A” has a WCET of 1 ms and a sampling period of 20 ms. In can be seen that the blocks “A” to “G” have a sampling period of 20 ms while the blocks “H” to “Y” have

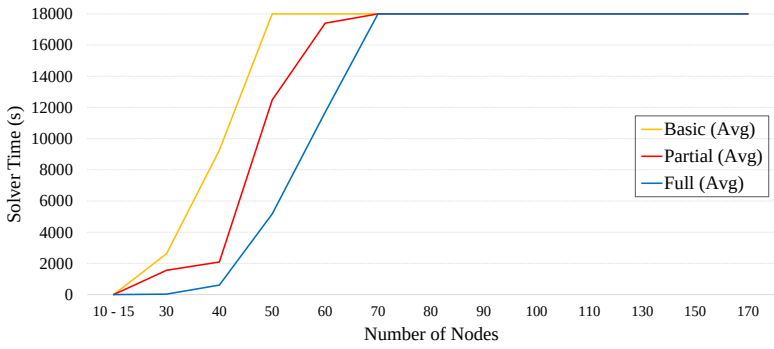

Fig. 8. Comparison of solver execution time between different approaches

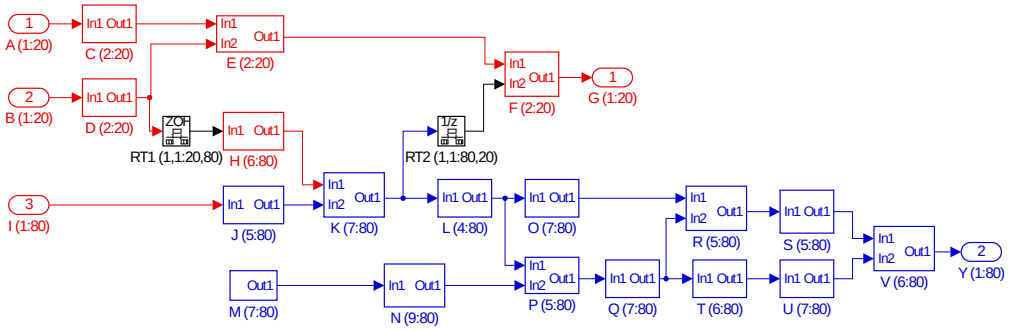

Fig. 9. A multi-rate block diagram sample

a sampling period of 80 ms. The blocks “RT1” and “RT2” are Simulink built-in rate-transition blocks which provide a mechanism for the data transfer between the blocks of different rates. The rate-transition blocks will have their inputs and outputs executing on different sampling periods. In this example, “RT1” has a sampling period of 20 ms for its input and 80 ms for its output and “RT2”, has a sampling period of 80 ms for its input and 20 ms for its output. For rate-transition blocks, we provide WCETs for both sampling periods, separated by a comma respective to the order of sampling periods. The inter-core communication costs are 8 µs for transmission and 8 µs for reception for any data connection between the blocks.

The total WCET time is 13 ms for the blocks with the sampling period of 20 ms including the executions of rate-transition blocks for this sampling period. The total WCET for the blocks with the sampling period of 80 ms is 90 ms. This makes a task generated from the blocks with period 80 ms not schedulable without seeking a feasible partition of the blocks to multiple cores.

Our tool computes the hyperperiod as $H = lcm(20, 80) = 80$ ms. The firing times in the hyperperiod are given as $F = \{0, 20, 40, 60\}$ in milliseconds which are the multiples of the sampling periods during the hyperperiod. Note that the firing time at 80 ms is actually the beginning of the next iteration of the hyperperiod. Copies of the blocks with the sampling period of 20 ms are fired at every firing time. The blocks with sampling period of 80 ms are only fired at the beginning of the hyperperiod. Then, the block dependency graphs for each rate are created. Figure 10 and Figure 11 give the block dependency graphs for the sampling period of 20 ms and 80 ms respectively.

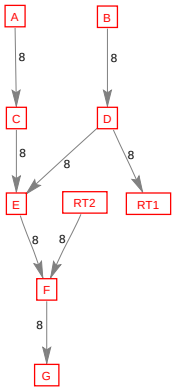

Fig. 10. Block Dependency Graph for 20 ms

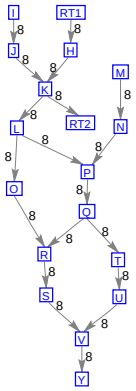

Fig. 11. Block Dependency Graph for 80 ms

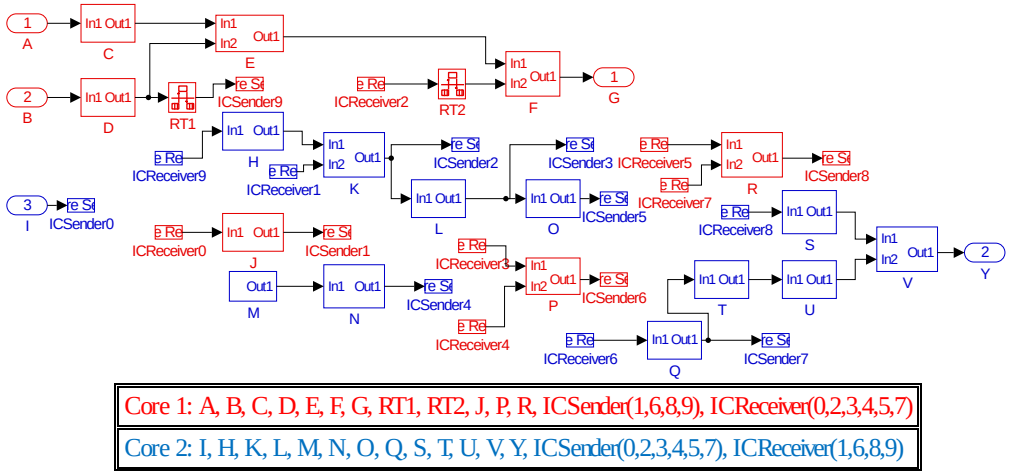

Fig. 12. Updated model for multi-core (contains blocks for both cores)

In the next step, our tool utilizes MILP solver to find a feasible mapping of the blocks. Here we are targeting a dual-core architecture. The updated model with block-to-core mapping is given in Figure 12. For any connection between different cores we add an inter-core sender block to the sending core and an inter-core receiver block to the receiving core. Because of space considerations, we cannot provide the figures of the model for each core here. In brief, the copy of the model for each core has the blocks of other core commented out. That is, in the generated model for Core 1, only the blocks that are mapped to Core 1 are active while the others are commented out and vice versa for Core 2. With this suggested mapping and computed ordering of the blocks, all of the blocks can complete execution in their periods on a multi-core architecture.

The Figure 13 gives the core mapping of the blocks with the execution time information in an hyperperiod. The horizontal lines illustrates the execution of each block. The notation for the labels is (block name, execution copy in hyperperiod : sampling time). For instance (RT1, 0 : 20) corresponds to the execution copy 0 of “RT1” block for sampling period 20 ms. Note that copy counts start from 0 for the first execution. It can be seen that the rate-transition blocks “RT1” and “RT2” executes for both sampling periods. In the Figure 13, we can observe that at every firing time, first the blocks with smaller sampling periods are executed first and after them, the blocks from larger sampling periods are executed. For instance, in the first firing time 0 ms blocks with 20 ms period are completed and “RT1” block for period 80 and block “J” start execution after them and the blocks with 80 ms period complete before the next firing time. Note that the execution information is based on worst-case execution times. An actual execution on the target platform would have same execution ordering with possibly shorter exe-

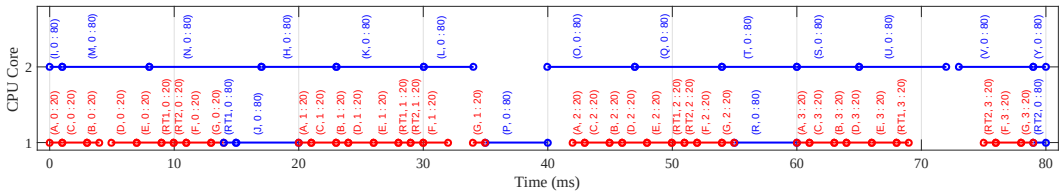

Fig. 13. CPU core mapping and execution information of blocks

cution durations and earlier start times (since an actual execution time may be shorter then the worst-case execution time).

6.3. Case Study: Toyota Diesel Engine Controller

We used the Diesel engine controller model from [Huang et al. 2013] as a single-rate case study from industry. The original model contains both controller and plant parts. The controller part of model has 1004 blocks when flattened as described in Section 5. It has 7 inputs that are multiplexed into a single input bus signal and 6 outputs that are multiplexed into a single output bus signal. Since the model has cycles inside the subsystems, our tool flattens the model by searching all blocks inside virtual subsystems, breaks the cycles as described in Section 5 and merges blocks inside subsystems (when possible) without introducing new cycles. For parallelizing this model we set the target model depth as 2. The block dependency graph capturing a flattened representation of the model up to the target model depth is generated. The generated block dependency graph contains 153 nodes and a total of 184 connections between these nodes. Our target platform for this case study is the dual-core architecture from Freescale which is described in Section 3. In our target hardware setup we have a total of 3.8 KB shared memory available.

For a model of this size, both the basic MILP formulation and the partially ordering heuristic fail in finding a solution in 10 hours. However, by merging blocks of subsystems with depth more than 2 and with our fully ordering heuristic, our tool returned a solution to the given problem within an average of 1.2 hours of solver execution time. Here the average is taken over different sets of worst-case execution time assignments. The suggested multi-core mapping by the tool achieves a speed-up factor of 1.44 on average. This result is parallel with our expectations based on experiments carried on randomly generated DAGs and illustrates applicability of our approach to reasonably large problems in industry.

7. CONCLUSIONS

In this paper we presented our approach for parallelizing single-rate and multi-rate Simulink models on multi-core architectures. We proposed a heuristic for partially deciding execution order of independent blocks when they are mapped to the same core. According to the experimental results for single-rate models with randomly generated DAGs, the MILP solver returns better solutions with this proposed heuristic in a reasonable limited solver execution time for models with around 50 to 60 blocks in our experimental environment. For models with larger number of blocks, we proposed another heuristic in which the execution order of all the independent blocks is decided in advance. With this approach our tool could handle models with larger than 150 blocks. We also presented this heuristic together with block merging methods on a single-rate case study from the industry where our tool reduced 1004 blocks to 153 nodes on the dependency graph by merging blocks deeper than a specified value and solved the problem on this 153 nodes. The results from the case study illustrate how our approach can handle single-rate models which can contain more than 1000 blocks. The introduced heuristics may not be as useful in multi-rate models. Because, these heuristics may eliminate the opportunity to fit small blocks in the idle times of the schedule where no higher priority task is executing.

For the future work, we consider extending this work by introducing heuristic methods for solving the optimization problem for the multi-rate models, studying models with blocks that have priority assignments. In addition, available parallelization under different triggering conditions should be further analyzed for the event-triggered models to get more optimal results. For large models, in order to improve efficiency of the heuristics, investigating control-flow level parallelization opportunities inside

the models is another promising approach to be studied. Similar to the schedule generation approach proposed by Lee et al. [2012], a depth-first traversal on the block dependency graph can be done to explore the model and merging of the blocks can be done in a smarter way based on the parallel paths in the graph. Furthermore, we plan to incorporate worst-case execution time (WCET) estimation tools in our framework.

REFERENCES

Karl J. Åström and Björn Wittenmark. 1997. *Computer-controlled Systems (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Tobias Achterberg. 2009. SCIP: solving constraint integer programs. *Mathematical Programming Computation* 1, 1 (2009), 1–41.

Ahmad Al Sheikh, Olivier Brun, Pierre-Emmanuel Hladik, and Balakrishna J Prabhu. 2012. Strictly periodic scheduling in IMA-based architectures. *Real-Time Systems* 48, 4 (2012), 359–386.

AUTOSAR. 2015. AUTOSAR Specification. (2015). http://www.autosar.org

Sanjoy Baruah. 2015. The Federated Scheduling of Systems of Conditional Sporadic DAG Tasks. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT '15)*. IEEE Press, Piscataway, NJ, USA, 1–10. http://dl.acm.org/citation.cfm?id=2830865.2830866

Armin Bender. 1996. Design of an optimal loosely coupled heterogeneous multiprocessor system. In *European Design and Test Conference, 1996. ED&TC 96. Proceedings*. IEEE, 275–281.

Girish Rao Bulusu. 2014. *Asymmetric Multiprocessing Real Time Operating System on Multicore Platforms*. Master's thesis. Arizona State University.

Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu. 2010. Automatic parallelization of simulink applications. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 151–159.

Certification Authorities Software Team. 2014. *Position Paper CAST-32 Multi-core processors*. Technical Report. Federal Aviation Administration.

Minji Cha, Kyong Hoon Kim, Chung Jae Lee, Dojun Ha, and Byoung Soo Kim. 2011. Deriving high-performance real-time multicore systems based on simulink applications. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*. IEEE, 267–274.

Jing Chen and Alan Burns. 1997. A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. *Report-University of York Department of Computer Science YCS* (1997).

Scott Cotton, Oded Maler, Julien Legriel, and Selma Saidi. 2011. Multi-criteria optimization for mapping programs to multi-processors. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, 9–17.

Jonathan Currie and David I Wilson. 2012. OPTI: lowering the barrier between open source optimizers and the industrial MATLAB user. *Foundations of computer-aided process operations, Savannah, Georgia, USA* (2012), 8–11.

Robert I Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)* 43, 4 (2011), 35.

Peng Deng, Fabio Cremona, Qi Zhu, Marco Di Natale, and Haibo Zeng. 2015. A model-based synthesis flow for automotive CPS. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*. ACM, 198–207.

EASA. 2012. *EASA/2011/6 Final Report*. Technical Report. European Aviation Safety Agency.

Johan Eker, Jörn W Janneck, Edward Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, and others. 2003. Taming heterogeneity-the Ptolemy approach. *Proc. IEEE* 91, 1 (2003), 127–144.

Ahmed Elhossini, John Huissman, Basil Debowski, Shawki Areibi, and Robert Dony. 2010. An efficient scheduling methodology for heterogeneous multi-core processor systems. In *Microelectronics (ICM), 2010 International Conference on*. IEEE, 475–478.

Esterel Technologies. 2015. SCADE Suite. (2015). http://www.esterel-technologies.com/

Juraj Feljan and Jan Carlson. 2014. Task Allocation Optimization for Multicore Embedded Systems. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, 237–244.

Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. 2010. A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems. In *25th ACM Symposium On Applied Computing*. Sierre, Switzerland, 527–534. https://hal.archives-ouvertes.fr/hal-00688490

Freescale Semiconductor Inc. 2015. Qorivva MPC5675K. (2015). http://www.freescale.com/

Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert De Simone. 2015. On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling. In *Formal Modeling and Analysis of Timed Systems*. Springer, 108–123.

Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

Frederik Gwinner. 2011. Transitive reduction of a DAG v1.2. (2011). http://www.mathworks.com/matlabcentral/fileexchange/32723-transitive-reduction-of-a-dag

Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. 2008. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software* 81, 1 (2008), 132–149.

Meng Huang, Hidemoto Nakada, Srinivas Polavarapu, Richard Choroszucha, Ken Butts, and Ilya Kolmanovsky. 2013. Towards combining nonlinear and predictive control of diesel engines. In *American Control Conference (ACC), 2013*. IEEE, 2846–2853.

Raimund Kirner, Roland Lang, Peter Puschner, and Christopher Temple. 2000. Integrating WCET analysis into a Matlab/Simulink simulation model. In *Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems*. 79–84.

Takahiro Kumura, Yuichi Nakamura, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai. 2012. Model based parallelization from the simulink models and their sequential C code. In *Proceedings of the 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 186–191.

Haeseung Lee, Weijia Che, and Karam Chatha. 2012. Dynamic scheduling of stream programs on embedded multi-core processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 93–102.

Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. 2009. Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size. *SIGPLAN Not.* 44, 1 (Jan. 2009), 78–89. DOI: http://dx.doi.org/10.1145/1594834.1480893

Roberto Lublinerman and Stavros Tripakis. 2008. Modular code generation from triggered and timed block diagrams. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 147–158.

Micrium Inc. 2015. μC/OS-II. (2015). http://micrium.com/rtos/ucosii/

Chris Ostler and Karam S Chatha. 2007. An ILP formulation for system-level application mapping on network processor architectures. In *Proceedings of the conference on Design, automation and test in Europe*. EDA Consortium, 99–104.

Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From Simulink specification to multi/many-core execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 309–318.

Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. 2015. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems* 51, 5 (2015), 526–565.

Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. 2014. Many-core scheduling of data parallel applications using SMT solvers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 615–622.

The MathWorks Inc. 2015. Simulink version 8.5 (R2015a). (2015). http://www.mathworks.com/

Lothar Thiele and Pratyush Kumar. 2015. Can real-time systems be chaotic?. In *Proceedings of the 12th International Conference on Embedded Software*. IEEE Press, 21–30.

Cumhur Erkan Tuncali, Georgios Fainekos, and Yann-Hang Lee. 2015. Automatic Parallelization of Simulink Models for Multi-core Architectures. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 964–971.

Dan Umeda, Takahiro Suzuki, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara. 2015. Multigrain Parallelization for Model-based Design Applications Using the OSCAR Compiler. In *Proceedings of the 28th International Workshop on Languages and Compilers for Parallel Computing*. 151–165.

Ying Yi, Wei Han, Xin Zhao, Ahmet T Erdogan, and Tughrul Arslan. 2009. An ILP formulation for task mapping and scheduling on multi-core architectures. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09*. IEEE, 33–38.

Received October 2015; revised ; accepted