Automatic Parallelization of Simulink Models for Multi-core Architectures

Cumhur Erkan Tuncali, Georgios Fainekos, Yann-Hang Lee
School of Computing, Informatics and Decision Systems
Arizona State University
Tempe, AZ, USA
{etuncali, fainekos, yhlee}@asu.edu

***Abstract*— This paper addresses the problem of parallelizing existing single-rate Simulink models for embedded control applications on multi-core architectures considering communication cost between blocks on different CPU cores. Utilizing the block diagram of the Simulink model, we derive the dependency graph between the different blocks. In order to solve the scheduling problem, we describe a Mixed Integer Linear Programming (MILP) formulation for optimally mapping the Simulink blocks to different CPU cores. Since the number of variables and constraints for MILP solver grows exponentially when model size increases, solving this problem in a reasonable time becomes harder. For addressing this issue, we introduce a set of techniques for reducing the number of constraints in the MILP formulation. By using the proposed techniques, the MILP solver finds solutions that are closer to the optimal solution within a given time bound. We study the scalability and efficiency of our consisting approach with synthetic benchmarks of randomly generated directed acyclic graphs. We also use the Fault-Tolerant Fuel Control System demo from Simulink and a Diesel engine controller from Toyota as case studies for demonstrating applicability of our approach to real world problems.**

Keywords—Multiprocessing, embedded systems, optimization, model based development, Simulink, task allocation.

I. Introduction

Model Based Development (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 the 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.

However, currently, the autocode generation process of commercial tools focuses 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 [1], 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), can we automatically partition the blocks onto the different cores so that the real-time constraints are satisfied?

In particular, we focus on control models built in the Simulink [2] MBD environment. Our goal is to produce a framework where non-determinism in the control algorithm is reduced or minimized to the extent possible. 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) [3] and in the Certification Authorities Software Team position paper CAST-32 Multi-core processors [4]. These classifications highlight the difficulty of certifying safety-critical systems that are based on multi-core architectures.

Our approach is based on keeping timing properties of parallelized software as simple as possible. For this purpose, we are aiming at having separate executables for each core while Simulink blocks are allocated in each core and executed in a predetermined order. In other words, we set the priorities of each block inside each core.

The contributions of this paper are,

- providing a practical solution to the Simulink model parallelization problem,
- improving available Mixed Integer Linear Program (MILP) formulations in the literature for finding better solutions within a fixed and practically feasible time for industrial size models,
- solving the multi-core mapping problem while considering the timing predictability of the parallelized application for ease of verification and certification, and
- developing a toolbox for automating parallelization of Simulink models to multi-core architectures.

II. 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. In [5] Anderson

This research was partly funded by the NSF awards CNS-1446730 and IIP-1361926, and the NSF I/UCRC Center for Embedded Systems.

et al. propose a Pfair [6] based scheduling method for real-time scheduling on multi-core platforms where the system has multiple tasks and task migration is allowed. For optimal mapping of tasks to CPU cores, Yi et al. [7], Bender [8] and Ostler et al. [9] discuss integer linear programming techniques which constitute a base for our optimization formulation. Cotton et al. discuss the use of mapping programs to multi processors in [10]. Tendulkar et al. discuss the application of SMT solvers in many-core scheduling for data parallel applications in [11]. In [12], Feljan et al. propose heuristics for finding a good solution for task allocation problems in a short time instead of searching for an optimal solution.

There are studies focusing on parallelization of Simulink models. In [13], Kumura et al. 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. To achieve thread level parallelism in multi-core, Canedo et al. introduce the concepts of strands for breaking the data dependencies in the model. A strand is defined as a chain of blocks that are driven by Mealy blocks [14]. The proposed method searches for available strand split points in Simulink models and it is heavily relying on strand characteristics in target models. In [15], Cha et al. is focusing 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.

There are studies on task parallelization as [9], [7], [8]. However, to apply the similar approaches, Simulink blocks must be considered as tasks. Given that most realistic models may consist of a significant number of blocks, either these methods fail to find an optimal solution in a reasonable amount of time or they rely on available loop level parallelism or functional pipelining as described in [9]. Deng et al. study model-based synthesis flow from Simulink models to AUTOSAR runnables [16] and runnables to tasks on multi-core architectures in [17]. The authors extend the Firing Time Automation (FTA) [18] 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 this work is targeting a similar problem to our target problem, they are providing experiment results for systems with less than 50 blocks and they are not considering inter-core communication and memory overhead.

Our work mainly differs from the other works in literature by

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

III. Problem Description

We are addressing the problem of automatically parallelizing existing Simulink models for embedded control applications on multi-core architectures in an optimal way and in a reasonable time.

We are focusing on single-rate, single-task embedded control applications which are modeled in Simulink and in which the execution order of blocks is determined only by dependencies coming from connections between blocks. Our target models cannot start execution of next iteration before finishing the execution of the current iteration.

Our target platform is Qorivva MPC5675K-based evaluation board [19]. The processor is a dual-core 32-bit MCU from Freescale targeting automotive applications. The µC/OS-II from Micrium [20] is ported on the target and a library to support Simulink code generation is devised for the platform [21]. 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 by Bulusu in [21]. For the purpose of utilizing this approach in Simulink, we model transmission and reception of data between different cores 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*.

A. Solution Overview

We approach the problem in five steps which are illustrated in Fig. 1. First, creating a directed acyclic graph which represents dependencies between blocks. Task-data graphs are discussed in [9]. We use a similar approach using blocks instead of tasks, worst case execution times of blocks instead of amount of work associated with tasks and using size of data communication between blocks. Here we will refer to this kind of graphs as "block dependency graphs". Our second step in approaching the problem is 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. The third step is automatically updating the original Simulink model by adding inter-core communication blocks where necessary in accordance with the most optimal solution. The next step is generating separate code for each target core by automatically commenting out the blocks that are not mapped to the core for which code is being generated. Finally, we compile the generated code and deploy it on the target platform.

IV. MILP Formulation

In this section we present our MILP formulation for the parallelization problem. Our MILP formulation for optimal solution is based on the formulations proposed by [5], [6] and [7]. We introduce an extension to these formulations by dividing the cost of communication to the transmission and reception parts. In Subsection D, we describe our techniques for reducing the number of constraints for allowing the MILP solvers to find better solutions within a feasible time.

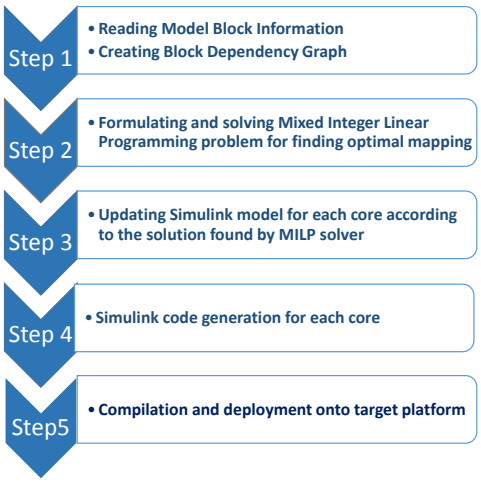

Fig. 1. Steps of going from a single-core Simulink model to multi-core target

A. Notation and Constants

The number of CPU cores available at the target architecture is denoted by $\mathbf{m}$. The set of CPU cores is defined as $\mathbf{P} = \{P_p : p \in [1, m]\}$. The number of nodes in the dependency graph is denoted by $\mathbf{n}$ where each node corresponds to a block in the flattened and merged Simulink model. Merging of blocks is done on the flattened model as described in subsection D. We describe the dependencies between blocks with the block dependency graph. This is a directed acyclic graph **G** = (B, E), where $\mathbf{B} = \{B_1, B_2, , B_n\}$ is the set of nodes and **E** is the set of edges in G. Each node B_i corresponds to a Simulink block with a worst case execution time $\mathbf{w_i}$ and each edge E_{ik} represents a data dependency from block B_i to block B_k. The set of leaf nodes in B, i.e., set of blocks which do not have any output ports is denoted by L and the set of start blocks, i.e., the set of blocks which do not have any input ports is denoted by S. We use Z for the set of deleted connections from the blocks that introduce delays (e.g., Unit Delay, Memory, Integrator, etc) to successor blocks. These connections exist in the original model, but they are deleted when forming the directed acyclic graph for removing cycles from the model. Such a connection is represented by $Z_{ik} \in Z$.

The size of the data transfer from block B_i to B_k in bytes is defined as c_{ik}. When B_i and B_k are mapped on different cores there will be a communication cost for transferring c_{ik} bytes of data between the cores. The communication cost is divided into transmission and receiving parts where t_{ik} denotes the transmission part of the communication time for sending c_{ik} bytes of data from block B_i to block B_k when they are mapped on different cores and r_{ik} denotes the receiving part of the communication time for sending c_{ik} bytes of data from block B_i to block B_k when they are on different cores.

The maximum allowed execution time for one iteration of the model on the target multi-core architecture is given by the deadline. It is either taken as a user input or calculated as the overall worst case execution time on a single-core architecture. The size of a global semaphore structure in bytes is denoted by $\mathbf{sSize}$ and the size of total available shared memory in bytes is defined as $\mathbf{totMem}$. Data alignment size in bytes (word size) is denoted by $\mathbf{aSize}$. A very large value ($\mathbf{MAX}$) is used in the program formulation to dominate other terms allowing constraints to be ignored under certain conditions.

B. Variables

$\mathbf{b_{ip}}$: A Boolean variable indicating whether block B_i is mapped to core P_p or not. It is defined for all $B_i \in B$ and for all $P_p \in P$. If B_i is mapped to core P_p, then b_{ip} takes value 1. If B_i is mapped to another core, then b_{ip} will takes value 0.

$\mathbf{d_{ik}}$: A Boolean variable indicating whether block B_i executes before or after B_k when both blocks are mapped to same core. It is defined for all B_i, $B_k \in B$ with $i < k$. If B_i executes before B_k, then d_{ik} takes value 1 and if B_i executes after B_k, then d_{ik} takes value 0.

$\mathbf{s_i}$: The start time for the execution of block B_i. It is defined for all $B_i \in B$. The lower bound for the variable s_i (best case start time) is denoted by $\mathbf{bs_i}$. It is determined by the best case completion time for all of the blocks from which there is a path to B_i in G. In the best case, all of this workload before B_i is distributed equally on all of the cores. The best case start time of B_i is calculated as $\left(\sum_{k \in K_i} w_k\right)/m$ where $K_i = \{B_k : B_k \in B \wedge there\ exists\ a\ path\ from\ B_k\ to\ B_i\ in\ G\}$. The upper bound for the variable s_i (worst case start time) is denoted by $\mathbf{ws_i}$. It is determined by the best case completion time for all of the blocks to which there is a path from B_i in G and the block B_i itself, subtracted from the deadline. The worst case start time of B_i is calculated as $deadline - \left(w_i + \sum_{k \in Y_i} w_k\right)/m$ where $Y_i = \{B_k : B_k \in B \wedge there\ exists\ a\ path\ from\ B_i\ to\ B_k\ in\ G\}$. For all i, k such that $B_i, B_k \in B$ and for all p such that $P_p \in P$.

$\mathbf{f}$: The completion time after executing all blocks. The lower bound for variable f is 0 and the upper bound is the deadline.

C. Objective Function and Constraints

The objective function for the optimization problem is minimizing f while the constraints for the optimization problem are defined as follows:

1) Every block shall be assigned to a single core:

$$\forall i : B_i \in B, \sum_{P_p \in P} b_{ip} = 1 \tag{1}$$

2) Delay introducing blocks and their first successor blocks shall be assigned to the same core:

$$\forall i, k : Z_{ik} \in Z\ and\ \forall p : P_p \in P,\ b_{ip} - b_{kp} = 0 \tag{2}$$

3) The finishing time of each leaf block shall be less than or equal to the completion time for executing all blocks: This constraint is serving for the purpose of being able to formulate the objective function $minimize(\max_{B_i \in L}(s_i + w_i)))$ as $minimize(f)$.

$$\forall i : B_i \in L, s_i + w_i \leq f \tag{3}$$

*4) If there is a dependency from block B_i to B_k, block B_k shall not start execution until **(i)** B_i finishes execution and transmission of its output data to its successor blocks that are mapped on other cores (which we temporarily define as f_i below) and **(ii)** B_k finishes receiving all of its input data that are sent by the blocks on other cores:* Considering that B_i is mapped to core P_p and B_k is mapped to core P_q where p can be equal to q:

$$\forall i,k : B_i, B_k \in B,\ E_{ik} \in E,\ \forall p,q : P_p, P_q \in P,$$

$$f_i \leq s_k - \sum_{B_l \in B} [r_{lk}(1 - b_{lq})] + (2 - b_{ip} - b_{kq})MAX \quad (4)$$

where $f_i = s_i + w_i + \sum_{B_l \in B}[t_{il}(1 - b_{lp})]$.

5) Execution of independent blocks that are mapped to same core cannot overlap: Considering B_i and B_k are mapped to core P_p, we have two different constraints for this requirement.

$$\forall i,k : i < k,\ B_i, B_k \in B,\ E_{ik} \in E,\ \forall p : P_p \in P,$$

$$f_i \leq s_k - \sum_{B_l \in B} [r_{lk}(1 - b_{lp})] + (3 - b_{ip} - b_{kp} - d_{ik})MAX \quad (5)$$

$$f_k \leq s_i - \sum_{B_l \in B} [r_{li}(1 - b_{lp})] + (2 - b_{ip} - b_{kp} + d_{ik})MAX \quad (6)$$

$$\text{Where, } f_i = s_i + w_i + \sum\nolimits_{B_l \in B}[t_{il}(1 - b_{lp})]$$
$$\text{and } f_k = s_k + w_k + \sum\nolimits_{B_l \in B}[t_{kl}(1 - b_{lp})]$$

Since MAX is a very large constant, (5) will be valid when block B_i executes before B_k i.e., when $d_{ik} = 1$ and (6) will be valid when block B_i executes after B_k i.e., when $d_{ik} = 0$.

6) Total memory needed for semaphores and communication buffers shall be less than or equal to total amount of available shared memory:

$$\forall i,k :\ B_i, B_k \in B,\ E_{ik} \in E,\ \forall p : P_p \in P$$

$$\sum_{B_i, B_k \in B} \left[\left(sSize + \left\lceil \frac{C_{ik}}{aSize} \right\rceil \cdot aSize \right) \cdot |b_{ip} - b_{kp}| \right] < totMem \quad (7)$$

D. Improving Solver Time

The number of variables and constraints in the MILP formulation grows exponentially as the number of blocks in the model increase. Consequently, the MILP solver starts failing in finding optimal or near optimal solutions for the problem in a reasonable time. In this section, we introduce our techniques for addressing this issue.

We say two blocks are *dependent* to each other if there exists a directed path between corresponding nodes in the DAG representation of the model and we say that two blocks are *independent* if there is no directed path between these nodes.

1) Partially ordering independent blocks: In order to reduce the execution time of a model by parallelization, the model must preferably have a large number of blocks that are independent to each other. If all blocks are dependent to each other, then there can be no multi-core mapping that will improve the execution time and, thus, the best solution will be mapping all blocks to the same core.

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 becomes large. In this case, when we consider all possible combinations of execution orders (priorities) between these independent blocks, the number of constraints introduced by inequalities (5) and (6) 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 independent blocks in advance. That is, before formulating the optimization problem, we decide the values of the d_{ik} variables for these block pairs. Since our execution order decision is valid only when these blocks are mapped onto the same core, 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 defined 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 $B_i \in B$ are defined in the subsection IV-B as bs_i and ws_i respectively. For all independent block pairs $B_i \in B$ and $B_k \in B$, if $\big((bs(i) \leq bs(k)) \wedge (ws(i) < ws(k))\big) \vee \big((bs(i) < bs(k)) \wedge (ws(i) \leq ws(k))\big)$ then we decide B_i to execute before B_k and set d_{ik} to 1. Else if $\big((bs(i) \geq bs(k)) \wedge (ws(i) > ws(k))\big) \vee \big((bs(i) > bs(k)) \wedge (ws(i) \geq ws(k))\big)$ then we decide B_i to execute after B_k and set d_{ik} to 0.

2) Fully ordering independent blocks: Even though ordering independent blocks using the partially ordering heuristic improves the performance, this is not 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 $B_i \in B$ and $B_k \in B$ we decide B_i to be executed before B_k if the average of bs_i and ws_i is smaller than the average of bs_k and ws_k. With this approach, d_{ik} variables of MILP formulation change to constant values. Our discussion on the case when these blocks are mapped to different cores in previous subsection is still valid.

3) Merging highly coupled blocks: In this heuristic we merge blocks B_i and B_k when block B_k is the only block connected to the output port(s) of block B_i and block B_i is the only block connected to the input port(s) of block B_k. The merging operation copies all incoming and outgoing edges of B_k to B_i except the edge E_{ik}. Then it updates w_i with $w_i + w_k$ and finally deletes B_k.

4) Merging small blocks with large blocks: In this heuristic we merge blocks B_i and B_k based on their ratio of execution times. If block B_k is the only block connected to the output port(s) of block B_i and the WCET of block B_i is very small when compared to the WCET of block B_k, then block B_i is merged into block B_k. If block B_i is the only block connected to the output port(s) of block B_k and the WCET of block B_k is very small when compared to the WCET of block B_i, then block B_k is merged into block B_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.

V. Implementation

In this section we describe the details of the implementation of our tool in MATLAB.

Our tool accepts as an input a Simulink model that is ready to compile as well as the desired depth of blocks to be parallelized. It loads the model, reads specific block information, e.g., block type, parents, 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. Using this information the model is flattened by taking blocks inside sub-systems out of their parent blocks. The remaining blocks like input and output ports of subsystems, emptied subsystem container blocks and 'Goto' - 'From' pairs, which are converted to line connections, are discarded from the set of blocks.

We represent all these dependencies in a directed graph where a directed edge represents a data communication from its source to its destination. Since determining Worst Case Execution Times (WCET) is not in scope of this paper, we assume that the WCET values for each of the blocks are already determined. If there exists a cycle in the directed graph, this means that there is a corresponding block in the cycle which creates a data dependency from a previous iteration of model execution. We will refer to these blocks as delay introducing blocks. In these cases we break the connection from delay introducing blocks to their successors for transforming a directed graph to a directed acyclic graph. Since the connection from delay introducing blocks to their successor blocks are deleted, our MILP solution can never introduce inter-core communication mechanism between these blocks even if they are mapped on different cores. For dealing with this issue we force the delay introducing blocks and their successor blocks to be mapped on the same core in the MILP formulation.

After all of the cycles are cleared, the blocks that are originally inside subsystems up to the desired model depth are merged together without introducing cycles between blocks. An exception to this is a subsystem including a delay introducing block. In this case, the blocks inside such a subsystem are not merged into a single block since this can cause a cycle in the dependency graph. In such a subsystem, predecessor blocks of a delay introducing block are only merged with other predecessor blocks and successor blocks are only merged with other successor blocks. In other words, a predecessor and a successor of a delay introducing block are never merged. The flow of the process up to this point is illustrated in the simple model in Fig. 2. In the next step, the block dependency graph is annotated with estimates of WCET. Fig. 4 gives an illustration of a simple block dependency graph.

The block dependency graph and the number of CPU cores on the target architecture are used in generating the MILP formulation presented in Section IV. The MILP solver returns the best solution found for mapping blocks to the available CPU cores and the execution order between these blocks.

The solution from the MILP solver is used to add inter-core communication blocks between the blocks 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, 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.

An example of the transformation is given in Fig. 3. The output of B1 is connected to the input of B2 in the original model. This connection is then replaced by inter-core communication blocks. After adding all needed communication blocks, we set the priority attributes of the blocks using the execution start time values obtained from the optimization solution.

As the last 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.

VI. Experiments

For studying the scalability and efficiency of our approach, we utilize randomly generated directed acyclic graphs with different number of nodes. We present results of these experiments in subsection VI-A and results of our case studies in subsections VI-B and VI-C. We use SCIP [22] from Achterberg as MILP solver which is interfaced with MATLAB through the Opti Toolbox [23] by Currie and Wilson. Experiments are run on a 64-bit Windows 7 PC with Intel Xeon E5-2670 CPU and 64 GB RAM.

A. Randomly Generated DAGs

For evaluating performance of our approach, we generate DAGs in which the WCET, communication costs and connections between blocks are assigned randomly. Then we solve

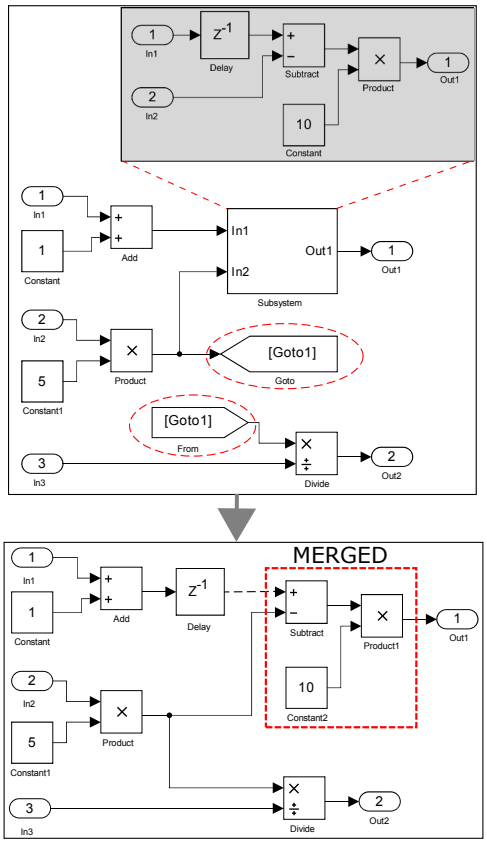

Fig. 2. Flattening models and merging blocks

the problem for a dual-core system with the basic MILP formulation which is given in Section IV 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. Here, we present a comparison of the performance of these three approaches in terms of the average speed-up achieved, the average solver 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 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 time is limited (5 hours in our experiments), it fails to find satisfactory solutions for large problems. Table I gives a comparison of the performance of the used approaches. Average speed-up achieved by basic MILP formulation, partially and fully

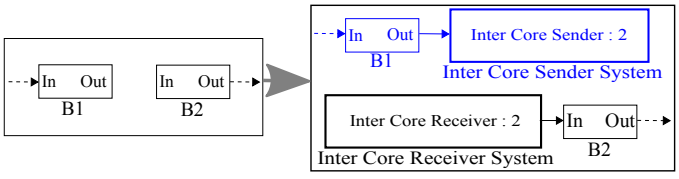

Fig. 3. Inter-core communication blocks

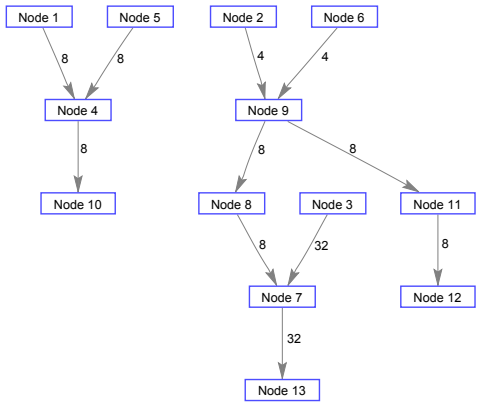

Fig. 4. Block dependency graph for a simple model

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 I, 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 I 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 Fig. 5, 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 Fig. 6, we illustrate the comparison between the two heuristics and the basic MILP formulation in terms of the average solver time over the number of nodes. Each line in the graph represents the average solver time spent for each

TABLE I. COMPARISON OF DIFFERENT APPROACHES

# Nodes	Approach	Average speed-up	Average solver time	% 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%

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 improve the solver time. In the graph it can be observed that the average solver time for proposed heuristics (as a function of node count) is smaller than the basic formulation. Combining the data in Fig. 5 and Fig. 6, we can see that the fully ordering heuristic returns better solutions within shorter solver run-time compared to the other approaches.

B. Case Study: Fault-Tolerant Fuel Control System

As a case study, we used the fuel rate control subsystem of the Simulink Fault-Tolerant Fuel Control System demo. This

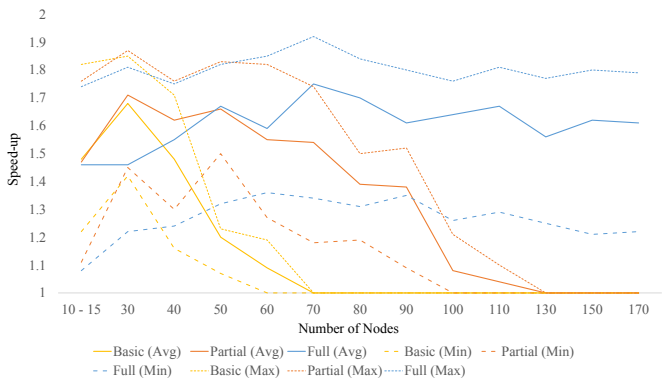

Fig. 5. Comparison of speed-up values between different approaches

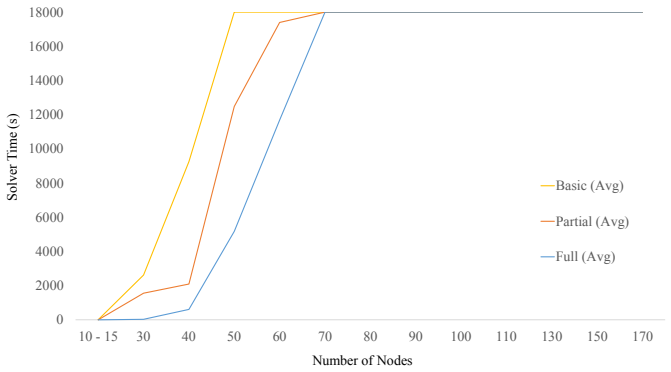

Fig. 6. Comparison of solver time between different approaches

model has 1 input port, 1 output port and 53 blocks after discarding the trivial blocks as described in Section V.

We performed parallelization on a completely flattened graph. The obtained block dependency graph from this model is presented in Fig. 7 where the blocks mapped to core 1 and core 2 are illustrated as the nodes colored with red and blue, respectively.

We achieved a speed-up value of 1.78 with the partially ordering heuristic within 5 hours of solver time. A speed-up value of 1.92 was achieved with the fully ordering heuristic. The basic MILP solution could only achieve a speed-up value 1.19 because it was unable to find the optimum solution within the given time limit of 5 hours. This result is parallel with the outcomes of the experiments carried on randomly generated DAGs.

C. Case Study: Toyota Diesel Engine Controller

We used the Diesel engine controller model from [1] as a 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 V, it has 7 inputs that are merged into a single input bus signal and 6 outputs that are merged into a single output bus signal. Since the model has cycles inside the subsystems, our tool flattens the model by searching all blocks inside subsystems, breaks

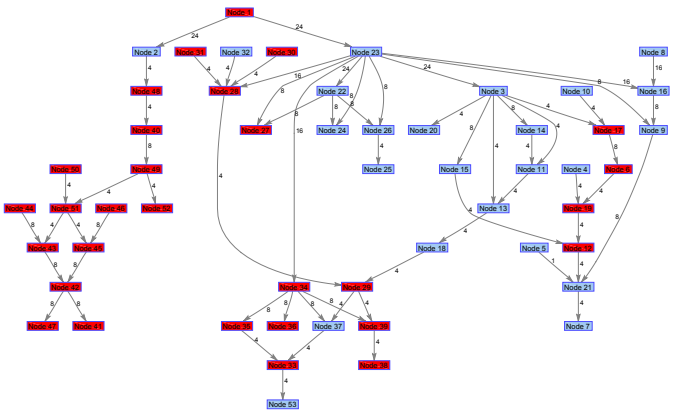

Fig. 7. The block dependency graph and its partition onto two cores for the fuel control system case study

the cycles as described in Section V and merges blocks inside subsystems (when possible) without introducing new cycles. For parallelizing this model we set the target model depth as 2. After merging deep blocks of each subsystem, the block dependency graph is generated from the model with merged blocks. 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 III. 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 time. Here the average is taken over different sets of worst case execution time assignments. The suggested multi-core mapping by the tool achieves 1.44x speed-up 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.

VII. Conclusion

In this paper we presented our approach for parallelizing a single-rate Simulink model on a multi-core architecture. 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 with randomly generated DAGs and our case study with the fuel system controller, this proposed heuristic improves optimality of found solutions in a reasonable time for a realistic size of 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 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 models which can contain more than 1000 blocks.

For the future work, we consider extending this work by introducing heuristic methods for solving the optimization problem, studying multi-rate models and models with blocks that have priority assignments. Furthermore, we plan to incorporate worst case execution time (WCET) tools in our framework.

References

[1] M. Huang, H. Nakada, S. Polavarapu, R. Choroszucha, K. Butts, and I. Kolmanovsky, "Towards combining nonlinear and predictive control of diesel engines," in *American Control Conference (ACC), 2013*. IEEE, 2013, pp. 2846–2853.

[2] Simulink, *version 8.5 (R2015a)*. Natick, Massachusetts: The MathWorks Inc., 2015.

[3] "EASA/2011/6 final report," European Aviation Safety Agency, Tech. Rep., 2012.

[4] Certification Authorities Software Team, "Position paper CAST-32 multi-core processors," Federal Aviation Administration, Tech. Rep., 2014.

[5] J. H. Anderson, J. M. Calandrino, and U. C. Devi, "Real-time scheduling on multicore platforms," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*. IEEE, 2006, pp. 179–190.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[7] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.* IEEE, 2009, pp. 33–38.

[8] A. Bender, "Design of an optimal loosely coupled heterogeneous multiprocessor system," in *European Design and Test Conference, 1996. ED&TC 96. Proceedings*. IEEE, 1996, pp. 275–281.

[9] C. Ostler and K. S. Chatha, "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, 2007, pp. 99–104.

[10] S. Cotton, O. Maler, J. Legriel, and S. Saidi, "Multi-criteria optimization for mapping programs to multi-processors," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, 2011, pp. 9–17.

[11] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-core scheduling of data parallel applications using SMT solvers," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 2014, pp. 615–622.

[12] J. Feljan and J. Carlson, "Task allocation optimization for multicore embedded systems," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, 2014, pp. 237–244.

[13] T. Kumura, Y. Nakamura, N. Ishiura, Y. Takeuchi, and M. Imai, "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)*, 2012, pp. 186–191.

[14] A. Canedo, T. Yoshizawa, and H. Komatsu, "Automatic parallelization of simulink applications," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 151–159.

[15] M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim, "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, 2011, pp. 267–274.

[16] AUTOSAR. (2015) AUTOSAR specification. [Online]. Available: http://www.autosar.org

[17] P. Deng, F. Cremona, Q. Zhu, M. Di Natale, and H. Zeng, "A model-based synthesis flow for automotive CPS," in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*. ACM, 2015, pp. 198–207.

[18] R. Lublinerman and S. Tripakis, "Modular code generation from triggered and timed block diagrams," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 2008, pp. 147–158.

[19] Freescale Semiconductor Inc. (2015) Qorivva MPC5675K. [Online]. Available: http://www.freescale.com/

[20] Micrium Inc. (2015) μC/OS-II. [Online]. Available: http://micrium.com/rtos/ucosii/

[21] G. R. Bulusu, "Asymmetric multiprocessing real time operating system on multicore platforms," Ph.D. dissertation, Arizona State University, 2014.

[22] T. Achterberg, "SCIP: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[23] J. Currie and D. I. Wilson, "OPTI: lowering the barrier between open source optimizers and the industrial MATLAB user," *Foundations of computer-aided process operations, Savannah, Georgia, USA*, pp. 8–11, 2012.