Model-Based Parallelizer for Embedded Control Systems on Single-ISA Heterogeneous Multicore Processors

Zhaoqian Zhong[1], Masato Edahiro[2]

[1]Ph.D. candidate, Graduate School of Information Science, Nagoya University

[2]Professor, Graduate School of Informatics, Nagoya University

[1]zhaoqian@ertl.jp

Abstract

This paper presents a model-based parallelization approach to parallelize embedded systems on single-ISA heterogeneous multicore processors, especially processors with ARM big.LITTLE architecture, wherein the core assignment of the Simulink blocks is determined based on the control design constraints and characteristics of ARM big.LITTLE architecture. The proposed approach uses a hierarchical clustering method on Simulink blocks to reduce the problem scale, and an integer linear programming formulation to determine the core assignment solution, considering load balancing and minimization of the inter-core communication across cores with different performances. Finally, we generate the parallel code of the model based on the core assignment solution for execution on the processors. We evaluate the proposed approach by comparing it with existing methods and generating the parallel code on a single-board computer with ARM big.LITTLE architecture to determine its effectiveness.

Keywords: Single-ISA heterogeneous multicore processor, ARM big.LITTLE architecture, model-based development, MATLAB Simulink, parallelization

1. Introduction

With the evolution of more complex embedded control systems, such as automotive control systems, model-based development (MBD) with platforms such as MATLAB/Simulink [1] is becoming increasingly common in recent years. A Simulink model is a brief and descriptive block diagram that can be automatically translated to a sequential source code for embedded implementation on single-core processors. On the other hand, heterogeneous multicore processors of the single instruction set architecture (ISA) [2] have potential benefits over homogeneous multicore processors. On a single-ISA heterogeneous multicore processor, cores may execute the same instruction set. However, they offer different capabilities and performances, such as different clock frequencies and power consumption. The combination of these heterogeneous cores may lead to better application performance. To implement the control models described in Simulink on a single-ISA heterogeneous multicore processor, the partition of the generated control software based on control design constraints, and the parallelization of the software components based on the characteristics of single-ISA heterogeneous multicore processors for parallel execution is necessary.

In this paper, a model-based parallelization approach is proposed to parallelize embedded systems built in the Simulink MBD environment on single-ISA heterogeneous multicore processors, especially those with ARM big.LITTLE architecture [3]. In this approach, a hierarchical clustering method is proposed to group blocks of the same attribute to top-level clusters. We then use an integer linear programming (ILP) formulation to assign these clusters on heterogeneous cores for the lowest communication cost and proper load balance. Our approach can also generate parallel codes based on the core assignment solution for execution on the processor.

The contributions of our work are as follows.

- We utilize SHIM [4] to estimate the workload of Simulink blocks and evaluate target processors in our approach. Hence, necessary parameters such as block execution time and signal line communication time can be easily acquired without executing the models on the processor in advance.
- An available ILP formulation is proposed to parallelize the model, considering the minimization of the communication cost, load balancing, and the characteristics of single-ISA heterogeneous multicore processors.
- A new structure called a cluster is proposed in our work, where Simulink blocks are gathered based on user configuration or identical attributes. Because building ILP formulations directly on Simulink blocks may lead to a considerable number of ILP variables and constraints, and a rather long solver run-time in complex models, using clusters can considerably reduce the problem scale, which makes the ILP computation much faster in most cases.
- Our proposed approach also contains code generation and user feedback, where model designers can easily implement their models for execution on the processor and comprehend the parallelization of the models in the MBD environment.

2. Related Work

There is a large amount of research being done on parallelizing control models in MBD, which can roughly be divided into code-level parallelization and model-level parallelization. For code-level parallelization, tools such as MATLAB Coder [5] are commonly used to generate sequential C codes from models, followed by a parallel compiler [6, 7] to parallelize the generated C codes. Code-level parallelization can provide higher parallel performance owing to a more fine-grained parallelism than the parallelization at the block level. However, as the generated sequential C codes discard some of the control information, it is difficult to parallelize the model owing to control design constraints. In addition, it is difficult to extract block allocation results for user feedback and model evaluation in code-level parallelization. Meanwhile, for model-level parallelization [8, 9, 11], the extraction of block-level parallelism from models is common. These blocks are partitioned to the cores on the processor. Simulink models can be seen as a block diagram or a dataflow graph, which consists of blocks that represent different parts of a system, and signal lines that define the dependency between the blocks. The model-level parallelization problem is to find the mapping and scheduling of block execution and signal line communication that could minimize the execution time of the block diagram on the target architecture. The mapping and scheduling of blocks are complex optimization problems, which need to be solved simultaneously to maximize the utilization of each core. The communication time between different cores must also be taken into account to solve this problem. In this case, linear programming (LP) introduces an appropriate solution to solve such problems [9, 10, 11]. We can describe Simulink models and target processors in an LP formulation and give it suitable constraints, followed by using LP solvers to solve the formulation for the optimal solution. However, there may be a substantial number of Simulink blocks and signal lines in the complex control model. Thus, the LP solver may run for a long time to solve the parallel problem on the blocks [9]. Therefore, it is necessary to reduce the problem scale to obtain a proper solver time during the parallelization of a large-scale model.

Furthermore, most of these existing studies target homogeneous multicore processors where all cores have the same parameters. It is easier to parallelize a control model for homogeneous multicores because designers do not have to consider the diversity of performance or power consumption, which are identical for all cores. A balanced distribution of the workload is essential for a single-ISA heterogeneous multicore processor composed of cores of varying performances and complexities to achieve high parallel performance [2]. Moreover, owing to the diversity of the cores, the parallelization of blocks becomes a nonlinear problem. The workload and inter-core communication time caused by signal lines may change when blocks are allocated on different cores. Among the single-ISA heterogeneous multi-core architectures, we focus on ARM big.LITTLE architecture [3] where cores are grouped by their parameters and they are homogeneous in all groups, except for the system core, if it exists. On a multi-core processor with ARM big.LITTLE architecture, the cores are marked big or LITTLE owing to their diversity. In this case, the inter-core communication can be described as a data transaction

behavior between the same type of cores, or between big and LITTLE cores, making parallelization a typical LP problem. On a processor with ARM big.LITTLE architecture, we can describe this problem as minimizing the communication time to reduce the time of execution of the whole model, while allocating blocks to big or LITTLE cores according to their workload. Thus, it is possible to achieve a considerably higher parallel performance and lower power consumption, as compared to using only homogeneous cores.

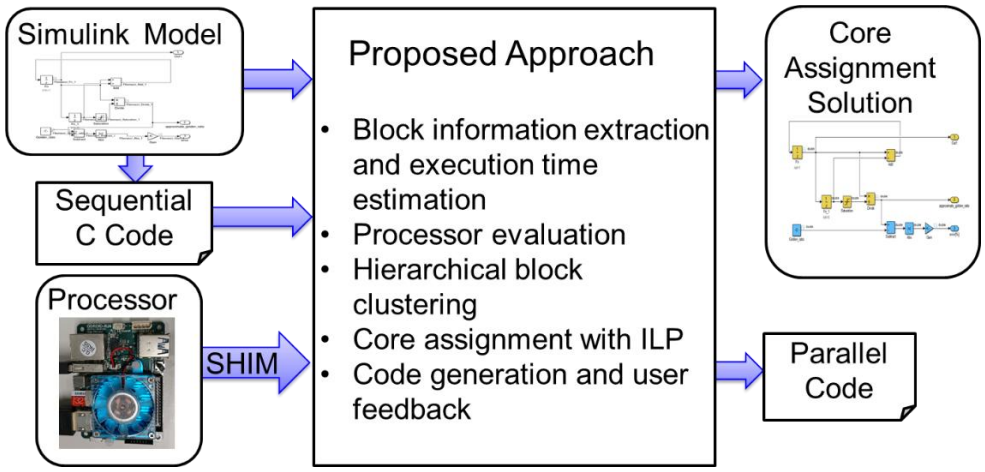


Fig. 1: Overview of model-level parallelization approach in MBD for multicore processor

Fig. 1 shows an overview of our proposed approach for model-level parallelization. It solves the model-level parallelization problems for processors with ARM big.LITTLE architecture and generates static core assignment solution of the Simulink models. Static core assignment, such as ILP [11] and graph partition [12, 13], is more suitable for embedded control applications, as compared to dynamic task allocation. Static core assignment does not need to run a task scheduler to determine the execution of blocks, which leads to a much smaller overhead in the execution of the application. A copy of the input model where the blocks are colored owing to the core assignment is generated as feedback to the model designers. With this graph, the model designers can understand the partitioning of the input models on the target processor and improve the design of the input control models.

3. Proposed Approach

In this section, we present an overview of our proposed parallelization approach, which combines the characteristics of both control design and implementation design, to solve the parallelization problems for ARM big.LITTLE architecture. Our approach consists of the four phases shown in Fig. 2. They are described in the following subsections.

- Data Extraction: extract information from the input Simulink model and the target processor and generate the necessary data file for parallelization.
- Hierarchical Clustering: group blocks according to user configuration and block attributes into high-level clusters.
- Core Assignment: assignment of the clusters to cores with our ILP formulation.
- Code Generation: generate the parallel code according to the core assignment solution.

3.1 Data Extraction

Our proposed approach takes control models designed in MATLAB Simulink, the hardware description of the target processor, and a user configuration file as the initial input. The user configuration file contains demands from model designers regarding implementation, such as the specific cores on the processor to be utilized to parallelize the input model, or whether some special Simulink blocks are preferred to be assigned to a specified core. We utilize tools from SHIM (Software-Hardware Interface for Multi-Many-Core) [4] to evaluate the target big.LITTLE heterogeneous multicore processors. SHIM is a hardware abstraction description standardized by

Multicore Association [14]. SHIM standardizes the interface between the multicore hardware and the software tools. It can be used to describe performance information from the perspective of software design. SHIM provides tools to roughly estimate software performance at the instruction level, which enables us to easily understand the number of clock cycles required by an instruction to be executed on a specific core. In our approach, we set the LITTLE core as the base core and use SHIM tools to generate a SHIM data file that obtains some of the architectural characteristics such as clock cycles for instructions on the base core. We also need to evaluate the processor for performance information such as the communication overhead between different cores and the processing speed of the big and LITTLE cores.

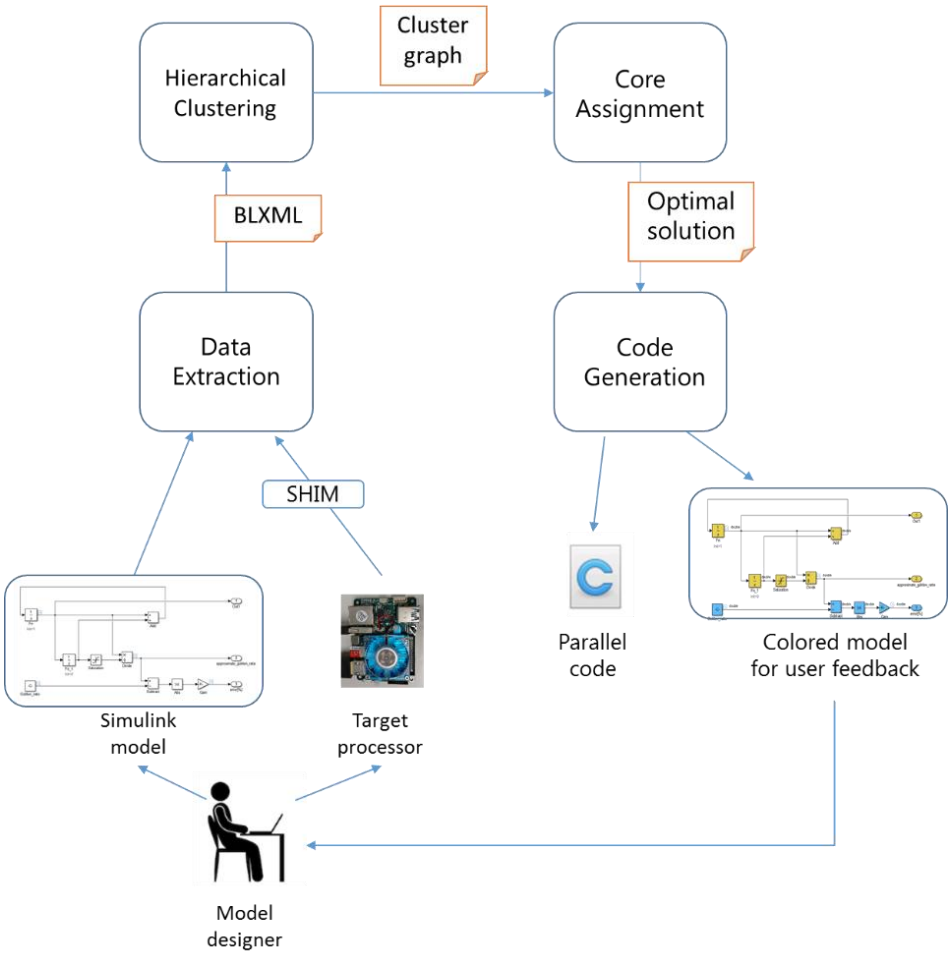


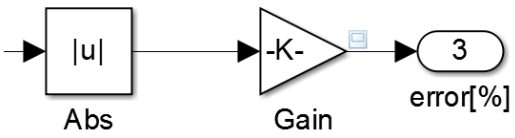
Fig. 2 An overview of each phase in the proposed approach

We standardize a block-level structure XML file (BLXML file) to describe the Simulink model in the proposed approach. The BLXML file mainly contains the following information, which is used in the next phase of our approach:

- Block parameters such as control rate, functional module, data type, etc.
- Block dependency on other blocks.
- Initialization and execution code of each block.
- Estimated execution time of each block.

Block parameters can be obtained from the Simulink model file. Block dependency can be extracted from the block diagram of the models. We generate a sequential C code of the input model with MATLAB Coder. The initialization and execution code of each block can be obtained from the generated code file. The execution time of each block can be estimated by combining block codes and instruction clock cycles in the SHIM data

file. Fig. 3 (a) shows a gain block in a Simulink model and Fig. 3 (b) is the code of the gain block in the BLXML file.



(a) Gain block in a Simulink model

```
<block blocktype="Gain" id="6" name="Gain" rate="-1">
  <input line="Abs_1" port="Gain_1">
    <connect block="Abs" port="Abs_1"/>
  </input>
  <output line="Gain_1" port="Gain_1" usename="true">
    <connect block="error" port="error_1"/>
  </output>
  <var line="Abs_1" mode="input" name="Abs_1" port="Gain_1" type="
   real_T"/>
  <var line="Gain_1" mode="extout" name="Gain_1" port="Gain_1" type="
   real_T"/>
  <param name="Gain_Gain" storage="P" type="real_T"/>
  <code file="Fibonacci.c" line="76" type="task">
                Gain_1 = P.Gain_Gain * Abs_1;
  </code>
  <code file="Fibonacci.c" line="113" type="init">
                Gain_1 = 0.0;
  </code>
  <code file="data.c" line="36" type="param">
                0.61803398874989479
  </code>
  <performance best="12.5" type="task" typical="16.5" worst="22.5"/>
  <performance best="12.75" type="init" typical="12.75" worst="12.75"/
   >
  <forward block="error" type="port">
    <var line="Gain_1" mode="input" name="Gain_1" port="error_1" type=
     "real_T"/>
  </forward>
  <backward block="Abs" type="data">
    <var line="Abs_1" mode="output" name="Abs_1" port="Abs_1" type="
     real_T"/>
  </backward>
</block>
```

(b) Code of gain block in BLXML file

Fig. 3 A sample of Simulink block and BLXML file

3.2 Hierarchical Clustering

In this phase, we group the blocks of the Simulink models into clusters based on the characteristics of control design on different levels to reduce the ILP problem scale.

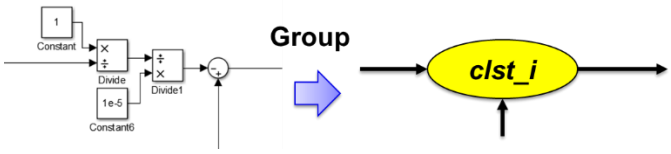


Fig. 4 Grouping continuous blocks into a cluster

Firstly, we group the Simulink blocks that should be assigned to the same core according to user configuration. This is followed by intentionally grouping some of the blocks and assigning them to the same core for design and implementation optimization. Considering the point of view of model design, either If or Switch Case blocks are used when the execution of some blocks is determined by a single input signal. The blocks present on the branches between an If or Switch Case block and the corresponding Merge block can be grouped to avoid branch selection. Considering the point of view of implementation, blocks such as Data Store Read blocks or Data Store Write blocks are used to read or write values to the same memory, and it is advisable to group blocks that are between a pair of Data Store Read block and Data Store Write block to reduce memory access. Moreover, continuous blocks that have a direct connection and share the same attribute should be clustered. For example, if two continuous blocks belong to the same functional module in the control design, they are usually gathered to the same atomic subsystem in the model and should not be separated to different cores during core assignment. In addition, if two blocks have a signal line between them and share the same rate parameter, they have a similar iterative execution behavior and can be seen as a single unit during core assignment. Fig. 4 shows an example of grouping continuous blocks with the same attributes into a cluster. We do such clustering hierarchically. The final generated groups of blocks are called clusters in our approach. ILP is a rigorous, yet heavy, method for the optimization problem. Therefore, parallelizing blocks directly may lead to a substantial number of ILP variables and constraints. This could result in a rather long run-time to obtain the optimal solution. Meanwhile, the utilization of clusters instead of blocks in ILP greatly reduces the complexity of the ILP problem and the time taken by the solver. In actual scenarios, there are commonly several tens of clusters that have the same control rate and belong to the same functional module in an automotive control model [15].

3.3 Core Assignment

In this phase, we assign clusters to cores on the target multicores. To define the model in the ILP formulation, we first generate a cluster graph from the result of the hierarchical clustering and estimate the necessary data from the BLXML file. We fit the forced core assignment according to the user configuration file and assign other clusters to cores with our ILP formulation. This phase is extremely important in our proposed approach. Therefore, we have provided more details about the proposed algorithm using ILP formulation in the next section. We then expand the clusters to the Simulink blocks and assign the blocks to the cores on the target processor. Our approach supports models with multiple control rates and action subsystems, where If or Switch Case blocks are used, which makes it important to ensure that these Simulink blocks will be executed in the right order, or a fatal error can occur in the code generation phase. We perform a path analysis on the block dependency to determine the execution sequence of the blocks on each core. Finally, we generate a copy of the input model where the blocks are colored according to the core assignment solution to feed back the assignment of these blocks to the model designer.

3.4 Code Generation

In the last phase, we generate the parallel code of the input model from the core assignment solution and the BLXML file. The generated code is implemented using POSIX Threads. Firstly, the block diagram of the input model is translated into a graph based on the communicating sequential process (CSP) theory [16], owing to the result of the path analysis where block dependency and execution order can be easily distinguished. We create one thread for each core on the target processor and write the execution code of each block from the BLXML file to these threads according to the core assignment solution and their execution order in the CSP graph. Execution cores for threads are specified using pthread affinity.

4 ILP Formulation for Core Assignment

In this section, we present our ILP formulation for cluster-level core assignment on heterogeneous multicore processors with ARM big.LITTLE architecture. Given the cluster graph, the user configuration, and the parameters of the target processor, our ILP formulation discovers the optimal static core assignment solution, which aims at minimizing the communication transactions between the cores to reduce the overall application execution time. Our ILP formulation also distributes the workload of the clusters to each core owing to the characteristics of ARM big.LITTLE architecture.

4.1 Architecture Definition

In our formulation, the user should specify which cores on the processor are used to parallelize the input model in the user configuration file. The parallelization problem on heterogeneous multicore processors with ARM big.LITTLE architecture is to find the mapping of clusters on the given cores. To solve this problem using linear programming, we divide the target heterogeneous multicore processor into two levels: core groups and cores. In our formulation, the cores in ARM big.LITTLE architecture are grouped into the big group and the LITTLE group based on their performance. We evaluate some of the major tools in ARM big.LITTLE architecture, such as Odroid-XU4 [17] and Jetson TX2 [18] and we observe that the communication overheads between two big cores or two LITTLE cores always remain at a very close range, while the overhead between a big core and a LITTLE core does the same. Therefore, in our ILP formulation, we assume that the inter-group overheads between the cores of different core groups are the same, and the inter-core overhead in both the core groups is also the same.

At the core group level, we define the set of two core groups as $GROUP = \{core\_group_g | g \in [big, LITTLE]\}$. Each core group is defined as a three-tuple: $core\_group_g = (core\_group\_num_g, core\_overhead_g, CORE_g)$, where $core\_group\_num_g$ represents the number of cores, $core\_overhead_g$ denotes the overhead of communication, and $CORE_g$ is the set of cores in $core\_group_g$. We use the communication overhead between a big core and a LITTLE core as the communication overhead between core groups, which is denoted by $group\_overhead$.

At the core level $CORE_g = \{core_{p,g} | p \in [0, core\_group\_num_g - 1]\}$ defines the set of cores in either core group. A core is defined as a five-tuple: $core_{p,g} = (core\_clst\_cpu\_util_{p,g}, core\_util_{p,g}, core\_speed_{p,g}, core\_max\_cpu\_util_{p,g}, core\_min\_cpu\_util_{p,g})$, where $core\_clst\_cpu\_util$ represents the total workload of clusters assigned to this core in the core assignment solution, $core\_util$ represents the utilization ratio of the core to be used in our formulation, and $core\_speed$ represents the processing speed of this core. The value of $core\_util$ is specified in the user configuration if factors such as operating system (OS) influence the utilization of the core, and its default value is set to 1 where all of the core resources can be used. The value of $core\_speed$ should be given in the user configuration or extracted from the evaluation on the real processor. To distribute the workload to each core, we use $core\_max\_cpu\_util_{p,g}$ and $core\_min\_cpu\_util_{p,g}$ as the maximum and minimum of the cluster workload assigned to $core_{p,g}$. They are supposed to be taken from the user configuration file if the model designer prefers to balance the block distribution. In this paper, we set $core\_min\_cpu\_util = 1$ to ensure that all of the cores are used and $core\_max\_cpu\_util = total\_cpu\_util / (\sum_{g \in GROUP} (\sum_{p \in CORE_g} core\_speed_{p,g})) * (1.05 + t)$, where $total\_cpu\_util$ denotes the sum of the estimated execution time of all the blocks and t denotes a value to avoid a non-optimal solution. It is possible that the constraint on $core\_max\_cpu\_util$ may not be satisfied and the formulation may not be solved. Therefore, t is used to increase the value of $core\_max\_cpu\_util$ if the proposed ILP formulation fails to find any solution.

4.2 Model Definition

After hierarchical clustering, the input model is represented by a cluster graph, which is an acyclic directed graph $G = (CLST, CONN)$, where $CLST$ is the set of clusters and $CONN$ is set of the communication edges between the clusters.

The number of clusters is denoted by m and the set of m clusters is defined as $CLST = \{clst_i | i \in [0, m-1]\}$. Each cluster is defined as $clst_i = (clst\_cpu\_util_i)$, where $clst\_cpu\_util_i$ is the estimated workload of the cluster

$clst_i$. The estimated workload $clst\_cpu\_util$ is calculated by the sum of estimated execution time and the control rate of the blocks grouped to $clst_i$. The BLXML file helps us to find the estimated execution time and the control rate parameter of the blocks. The sum of the estimated execution time shows the time taken to execute all blocks in this cluster sequentially on the base core of the processor. The control rate parameter determines the frequency of cluster execution in loop interactions. For example, a very low control rate shows that the blocks in this cluster are not executed frequently. Thus, despite the large value of the sum of the estimated execution time of the blocks in this cluster, the value of $clst\_cpu\_util$ may still be very small. We find the cluster with the lowest control rate $clst\_lowest\_rate$, and use $clst\_execution\_time_i$ to denote the sum of estimated execution time of the blocks grouped in $clst_i$ and $clst\_rate_i$ to denote the control rate of the blocks grouped in $clst_i$. Hence, the estimated workload of $clst_i$ is defined as $clst\_cpu\_util_i = clst\_execution\_time_i * clst\_rate_i / clst\_lowest\_rate$.

The number of communication edges is denoted by n. We define the set of n communication edges as $CONN = \{conn_j | j \in [0, n-1]\}$. Each communication edge is defined as a three-tuple: $conn_j = (conn\_s\_clst_j, conn\_t\_clst_j, conn\_weight_j)$, where $conn\_s\_clst_j$ and $conn\_t\_clst_j$ represent the start cluster and termination cluster of the communication edge $conn_j$, respectively, and $conn\_weight_j$ is the estimated communication time of the communication edge. A communication edge can be seen as a set of signal lines that start from blocks in $conn\_s\_clst_j$ and go to blocks in $conn\_t\_clst_j$. The $conn\_weight_j$ is decided by the number of signal lines and the control rates of $conn\_s\_clst_j$ and $conn\_t\_clst_j$. For example, a signal line from a block in $conn\_s\_clst_j$ to a block in $conn\_t\_clst_j$ indicates that the two clusters have a communication transaction, and the high control rates of $conn\_s\_clst_j$ or $conn\_t\_clst_j$ indicate the high frequency of communication transaction in loop interactions, so a significantly higher value of $conn\_weight_j$ should be set accordingly. In our formulation, the value of $conn\_weight_j$ is set to the product of the number signal lines between $conn\_s\_clst_j$ and $conn\_t\_clst_j$, and the higher of the control rate multiples of $conn\_s\_clst_j$ and $conn\_t\_clst_j$, to quantize the communication behaviour of $conn_j$.

4.3 Variables

Following are the variables used in our ILP formulation to denote to which core group or core a cluster is assigned:

- $x\_group_{i,g}$: equals to 1 if cluster $clst_i$ is assigned to core group $core\_group_g$, otherwise equals to 0.
- $x\_core_{i,p,g}$: equals to 1 if cluster $clst_i$ is assigned to core $core_{p,g}$ in core group $core\_group_g$, otherwise equals to 0.

We also use the following variables in our ILP formulation to denote whether the start cluster and termination cluster of a communication edge are assigned to the same core group or to the same core:

- y_group_j: equals to 0 if both end clusters, $conn\_s\_clst_j$ and $conn\_t\_clst_j$, of communication edge $conn_j$ are assigned to the same group, and equals to 1 otherwise.
- $y\_core_{j,g}$: equals to 0 if both end clusters, $conn\_s\_clst_j$ and $conn\_t\_clst_j$, of communication edge $conn_j$ are assigned to the same core in group $core\_group_g$, and equals to 1 otherwise.

4.4 Objective Function

Our proposed ILP formulation assigns clusters to the specified cores on a heterogeneous multicore processor with ARM big.LITTLE architecture. For all communication edges, $conn_j$, whose $y\_group_j = 1$ or $y\_core_{j,g} = 1$, we use the product of $conn\_weight_j$ and the corresponding overhead as the communication cost of $conn_j$. The sum of all these communication costs represents the communication cost of the whole application and we can minimize this cost to reduce the number of communication edges as well as the communication time between the cores. The objective function for the core assignment problem is as follows:

$$minimize \sum_{j \in CONN} conn\_weight_j * (y\_group_j * group\_overhead + \sum_{g \in GROUP} y\_core_{j,g} * core\_overhead_g)$$

4.5 Constraints

The constraints for the core assignment problem are defined as follows.

- Each cluster shall be assigned to only one core group:

$$\forall i \in CLST: \sum_{g \in GROUP} x\_group_{i,g} = 1 \quad (1)$$

- Each cluster shall be assigned to only one core on the processor:

$$\forall i \in CLST: \forall g \in GROUP: \sum_{p \in CORE_g} x\_core_{i,p,g} = x\_group_{i,g} \quad (2)$$

- Whether the two end clusters of a communication edge are assigned on the same core group is checked by the following calculation:

$$\forall j \in CONN: y\_group_j = (\sum_{g \in GROUP} abs \left(x\_group_{conn\_t\_clst_j,g} - x\_group_{conn\_s\_clst_j,g} \right))/2 \quad (3)$$

- Whether the start cluster and termination cluster of a communication edge are assigned on the same core is checked by the following calculation:

$$\forall j \in CONN: \forall g \in GROUP: y\_core_{j,g} = \left\lfloor (\sum_{p \in CORE_g} abs \left(x\_core_{conn\_t\_clst_j,p,g} - x\_core_{conn\_s\_clst_j,p,g} \right))/2 \right\rfloor \quad (4)$$

- The sum of workload of clusters assigned to each core of different processing speed is calculated as follows:

$$\forall g \in GROUP: p \in CORE_g: core\_clst\_cpu\_util_{p,g} = \sum_{i \in CLST} x\_core_{i,p,g} * core\_util_{p,g} / core\_speed_{p,g} \quad (5)$$

- The sum of workload of clusters assigned to a core shall not exceed its upper limit and it is calculated as follows:

$$\forall g \in GROUP: p \in CORE_g: core\_clst\_cpu\_util_{p,g} \leq core\_max\_cpu\_util_{p,g} \quad (6)$$

- The sum of workload of clusters assigned to a core shall not exceed its lower limit and it is calculated as follows:

$$\forall g \in GROUP: p \in CORE_g: core\_clst\_cpu\_util_{p,g} \geq core\_min\_cpu\_util_{p,g} \quad (7)$$

5 Experiments

To study the scalability and efficiency of our method with the ILP formulation, we use randomly generated cluster graphs with different number of clusters. Furthermore, we parallelize an automotive control evaluation model based on real scenarios with our approach and execute the parallel code on ODROID XU4 single-board computer. Among a variety of ILP solvers, we use IBM ILOG CPLEX Optimization Studio [19] to solve the core assignment problem. The upper time limit of CPLEX execution is acceptably set to 5 hours (18,000 seconds), even though it takes only a few minutes to find the solution for our formulation of 100 clusters. The proposed approach and the CPLEX solver are executed on a PC with an Intel Xeon CPU E5-2695v2 2.40 GHz, in which the cache size is 30720 kB and the main memory size is 32 GB.

5.1 Randomly Generated Cluster Graphs

We use randomly generated directed acyclic graphs of clusters to evaluate the performance of the ILP formulation for the assignment of cores. Ideally, we should use data from real control models but gathering reasonable models is not easy. The artificial generation of well-controlled models is also difficult. Hence, we use randomly generated directed acyclic graphs, such as input cluster graphs, whose parameters, such as cluster

estimated workload and communication time, are generated randomly. The range of these parameters are based on real-scenario models.

In these experiments, we generate 100 cluster graphs of each specified cluster number, followed by using our ILP formulation and a graph partition method called khmetis [20] for core assignment solutions. khmetis is a program provided by hMETIS [21]. It computes a k-way partitioning using multilevel k-way partitioning. hMETIS is commonly used to solve problems such as task allocation for multicores [8]. As khmetis uses random seeds to generate graph partitions, we execute khmetis 100 times with different execution parameters and use the partition with the lowest communication cost as the core assignment result in our experiments. We assume a big.LITTLE heterogeneous multicore system as the target processor, where the $core\_speed$ of big cores is 3 times higher than the $core\_speed$ of LITTLE cores and the $group\_overhead$ is 5 times higher than the $core\_overhead$. We use a user configuration of one big core and one LITTLE core, and another of two big cores and four LITTLE cores in our experiments.

Table 1. Core assignment on randomly generated cluster graphs for one big and one LITTLE cores.

Cluster number	Approach	Average solver time	Average speedup	Average load balance
20	proposed	0.13	2.88	1.90
	khmetis	1.51	1.65	1.37
30	proposed	0.22	3.23	1.91
	khmetis	3.66	1.74	1.34
40	proposed	0.19	3.34	1.90
	khmetis	6.55	1.76	1.33
50	proposed	0.13	3.44	1.89
	khmetis	10.26	1.78	1.33
60	proposed	0.32	3.50	1.88
	khmetis	14.60	1.78	1.33
70	proposed	0.32	3.52	1.86
	khmetis	19.66	1.76	1.32
80	proposed	0.31	3.57	1.87
	khmetis	25.37	1.78	1.32
90	proposed	0.46	3.56	1.86
	khmetis	31.75	1.75	1.31
100	proposed	0.60	3.56	1.86
	khmetis	38.99	1.81	1.34

We present metrics on the core assignment results of randomly generated cluster graphs in Table 1 and 2. The average solver time denotes the average execution time required by our ILP formulation and khmetis to solve core assignment problems of different sizes. Average speedup metric is the ratio between the sequential execution time and the parallel execution time at the cluster level. Here, we use the sum of the workload of the sequential execution time of all clusters. The parallel execution time is the total execution time computed by the core assignment result and the dependency of these clusters. The average load balance metric is the ratio between the sequential execution time and the highest $core\_clst\_cpu\_util$. It indicates the efficiency of core utilization.

For a small number of clusters and cores, our ILP formulation executes much faster than khmetis to obtain the solution. However, when the number of clusters and cores increase, the solver time to solve the core assignment problem in ILP may become much longer, while still remaining within the acceptable upper time limit. The existing method, khmetis, cannot properly balance the weights of partitions on heterogeneous architectures. It fails to parallelize these clusters and cannot use cores more efficiently with respect to the speedup and load balance, as compared to ILP. However, for a smaller number of clusters on six cores, the ILP formulation may

generate solutions of lower speedup and load balance, as compared to a larger number of clusters. We set the upper limit of cluster workload on each core, which makes it difficult to distribute the cluster workload evenly in a scenario where the number of clusters is small, but the workload of most clusters is large. In such cases, we raise the upper limit on the cores to generate a legal solution of poor workload balance where the heaviest clusters are assigned to big cores. Additionally, we use $core\_min\_cpu\_util$ as the lower limit of workload on each core in the ILP formulation to utilize each core. However, this action may lead to too many communication transactions between the cores for a small number of clusters, thereby increasing the whole application execution time.

Table 2. Core assignment on randomly generated cluster graphs for two big and four LITTLE cores.

Cluster number	Approach	Average solver time	Average speedup	Average load balance
20	proposed	14.14	3.01	3.52
	khmetis	36.97	1.82	3.19
30	proposed	1.88	4.25	4.72
	khmetis	40.54	2.96	3.70
40	proposed	8.30	5.16	5.38
	khmetis	45.17	3.89	4.09
50	proposed	9.94	6.16	5.56
	khmetis	50.61	4.42	4.20
60	proposed	22.07	6.67	5.60
	khmetis	57.93	4.59	4.27
70	proposed	54.2	6.91	5.61
	khmetis	66.09	4.77	4.34
80	proposed	89.23	6.91	5.60
	khmetis	75.77	4.70	4.37
90	proposed	214.58	7.15	5.6
	khmetis	89.86	4.74	4.37
100	proposed	462.91	7.03	5.59
	khmetis	102.24	4.67	4.40

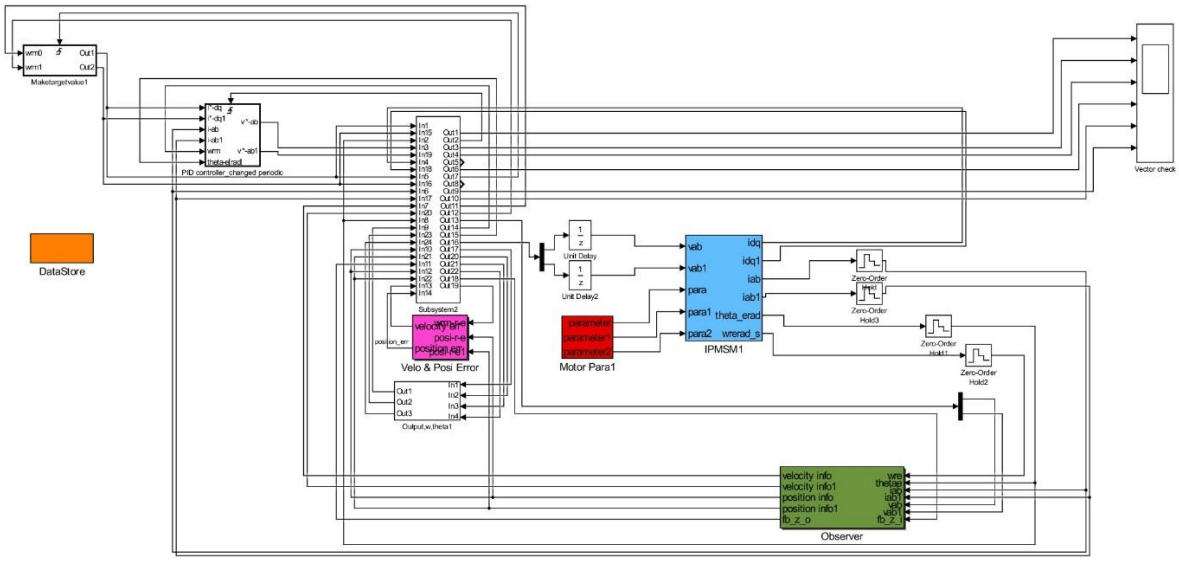


Fig. 5 Motor control model

5.2 Motor Control Model

The motor control model is provided by our research partner, who is an automotive manufacturer. It is abstracted from a real automotive control evaluation model. Fig. 5 provides an overview of the motor control model. This model is a multi-rate Simulink model and contains complex Simulink structures such as triggered subsystems and S-functions. There are a total of 514 Simulink blocks in the motor control model, which reduces to 31 after hierarchical clustering. The number of communication edges between clusters is 83. In this experiment, we perform parallelization on the motor control model with our approach. We then implement it on the Odroid-XU4 [16] board to evaluate the execution time of the generated parallel code with the core assignment solution from our ILP formulation. ODROID-XU4 is one of the latest single board computing devices and equips a Samsung Exynos 5422 processor which includes four Cortex-A15 and four Cortex-A7 cores in a big.LITTLE configuration, as shown in Fig. 6. Ubuntu 16.04.3 is run on Odroid-XU4, and we set the CPUFreq Governor policy to performance mode to ensure that the cores work at the highest clock frequency, where Cortex-A15 is at 2 GHz and Cortex-A7 is at 1.4 GHz. From the performance evaluation of the big cores and the LITTLE cores, we set the LITTLE cores as the base cores, and the $core\_speed$ of the big cores is set to 2.2 times that of the $core\_speed$ of the LITTLE cores. Although we observe the execution time may suffer from influences, such as OS or TDP limit, $core\_util$ of each core is set to 1. We input the motor control model and the description file of Odroid-XU4 to our proposed approach and generate the core assignment solutions and the parallel codes for the specified configurations. Considering the number of clusters in the motor control model, we use at most 4 cores in the experiment. We execute each of these generated parallel codes on the Odroid-XU4 board and record the execution times. Our approach takes only several minutes to generate these parallel codes.

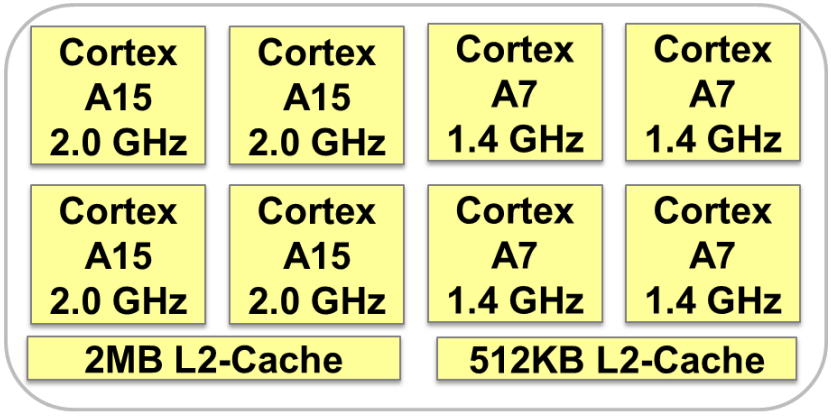


Fig. 6 Overview of Odroid-XU4

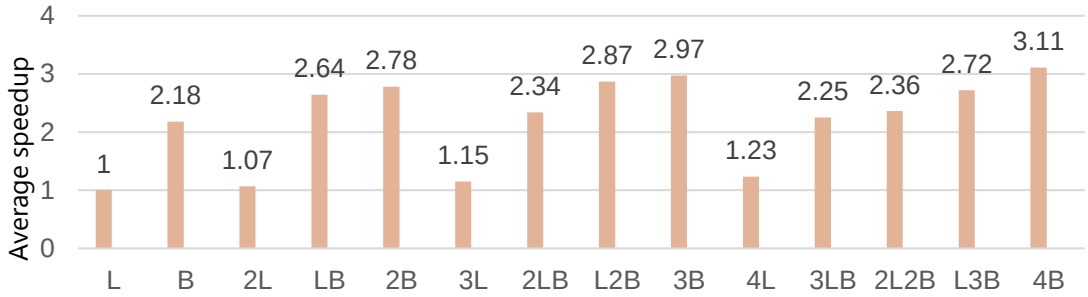


Fig. 7 Speedup performance of generated codes executed on ODROID-XU4

Fig. 7 shows the speedup performance of generated codes of different configurations to be implemented on ODROID-XU4. L and B in the configuration name denote the number of LITTLE or big cores being used to

execute the generated parallel codes. We record the average time taken to execute the generated source code on ODROID-XU4 in seconds. The average speedup is the ratio between the average execution time on a LITTLE core and the average execution time of the given configurations.

The results show that the execution on ODROID-XU4 has lower speedup as compared to the randomly generated cluster graphs in 5. 1, where speedup is calculated ideally at the cluster level. However, the core assignment in the configurations with both big and LITTLE cores achieves a decrease in application execution time and reasonable speedup, as compared to the configurations where the same type of core is used. This shows that for a control model of this size, our approach achieves a reasonably high parallel efficiency and low communication cost on such processors in a short solver time.

However, we observe in the result that in some configurations, more time is taken to execute the model when more cores are being used. For example, as compared to two big and two LITTLE cores, the speedup is lower when two big cores and only one LITTLE core is used. This happens because using more LITTLE cores results in more signal lines across the big and LITTLE cores. These signal lines may lead to heavier inter-group communication during the execution and the whole execution time of the application will increase. Also, the motor control model is a multirate model based on real scenarios. Most of its blocks have higher control rates and only a few blocks have a lower control rate. Owing to the workload constraints in our ILP formulation, if too many LITTLE cores are used for implementation, some high rate blocks are assigned to these slow cores and the threads on the LITTLE cores may suffer a much longer execution time, resulting in a lower speedup. We observe that when using both big and LITTLE cores, using one LITTLE core and two big cores is the best choice to implement the motor control model. In this configuration, most of the low rate blocks are assigned to the LITTLE core, whereas the heavy blocks are distributed to the two big cores to be executed in a well-parallelized manner.

This shows a potential utilization of our approach when multiple models must be executed on a processor simultaneously. By merging multiple models into 1 cluster graph, and avoiding blocks of different models from being grouped into the same cluster, our approach can find the optimal solution to parallelize these blocks on the heterogeneous cores with an optimal execution time of the models and utilization of the cores. However, it also leads to a challenge in parallelizing a Simulink model on a single-ISA heterogeneous multicore processor with ARM big.LITTLE architecture. When several cores of different performances on a processor are specified to implement a control model, it is possible that using only some of the given cores achieves the best parallel performance. Thus, a proper prediction is necessary to build the ILP formulation.

6 Conclusion

In this paper, we addressed a model-based parallelization approach based on ILP to parallelize embedded control systems designed on the MATLAB/Simulink platform for single-ISA heterogeneous multicore processors, especially the processors with ARM big.LITTLE architecture. In comparison to existing methods, our method can minimize the communication cost across the cores to generate a better parallelization solution, while distributing the workload of the input Simulink blocks to the cores having different performances. Moreover, our approach utilizes the characteristics of ARM big.LITTLE architecture to achieve high parallel efficiency and core utilization. The results on randomly generated data have shown that a higher speedup and lower communication cost are achieved by our approach on the assumed architectures. We also implement a real model on the ODROID-XU4 board and observe a reasonable speedup performance. Besides ARM big.LITTLE architecture, our ILP formulation can also be applied to other Single-ISA heterogeneous multi-core architectures where cores can be grouped to heterogeneous clusters by inter-core communication overhead, and the available cores in each cluster share inter-core communication overhead at a close range. In future work, we plan to extend our approach to architectures where cores have more complicated communication behavior.

Conflicts of Interest

The authors declare no conflicts of interest associated with this manuscript.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 16H02800.

References

1. MathWorks, Inc. "Simulation and Model-Based Design." https://jp.mathworks.com/products/simulink.html, 2015.

2. Kumar, Rakesh, et al. "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction." Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2003.

3. Chung, Hongsuk, Munsik Kang, and Hyun-Duk Cho. "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big. LITTLE™ Technology." Samsung White Paper (2012).

4. Gondo, Masaki, Fumio Arakawa, and Masato Edahiro. "Establishing a standard interface between multi-manycore and software tools-SHIM." COOL Chips XVII, 2014 IEEE. IEEE, 2014.

5. MathWorks, Inc. "MATLAB Coder Generate C and C++ code from MATLAB code." https://jp.mathworks.com/products/matlab-coder.html, The MathWorks, Inc, 2012.

6. Dan, Umeda, Youhei, Kanehagi, et al. "Automatic Parallelization of Designed Engine Control C Codes by MATLAB/Simulink. " Journal of Information Processing, Vol.55, No.8, pp 1817-1829, 2014.

7. Cha, Minji, et al. "Deriving high-performance real-time multicore systems based on simulink applications." Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on. IEEE, 2011.

8. Kumura, Takahiro, et al. "Model based parallelization from the simulink models and their sequential C code." Proceedings of the 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012). 2012.

9. Höttger, Robert, Lukas Krawczyk, and Burkhard Igel. "Model-based automotive partitioning and mapping for embedded multicore systems." International Conference on Parallel, Distributed Systems and Software Engineering. Vol. 2. No. 1. 2015.

10. Yi, Ying, et al. "An ILP formulation for task mapping and scheduling on multi-core architectures." Proceedings of the conference on design, automation and test in Europe. European Design and Automation Association, 2009.

11. Tuncali, Cumhur Erkan, Georgios Fainekos, and Yann-Hang Lee. "Automatic Parallelization of Simulink Models for Multi-core Architectures." 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, 2015.

12. Sou, Aburadani, and Masato, Edahiro. "Task Mapping Method for Hierarchical Many-core Processor Architectures." Journal of Information Processing, Vol.56, No.8, pp 1568-1581, 2015.

13. Edahiro, Masato, and Takeshi, Yoshimura. "New placement and global routing algorithms for standard cell layouts." Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE. IEEE, 1990.

14. Multicore Association. "SHIM - Multicore Association." https://www.multicore-association.org/workgroup/shim.php, 2018.

15. Embedded Multicore Consortium. Discussion at Embedded Multicore Consortium, 2015.

16. Hoare, Charles Antony Richard. "Communicating sequential processes." Communications of the ACM 21.8 (1978): 666-677.

17. Hardkernel. "ODROID-XU4 User Manual." http://www.hardkernel.com, 2017.

18. Franklin, D. "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge." NVIDIA Accelerated Computing| Parallel Forall (2017).

19. CPLEX, IBM ILOG. "12.7, User's Manual for CPLEX." CPLEX division, 2016.

20. Karypis, George, and Vipin Kumar. "Multilevelk-way partitioning scheme for irregular graphs." Journal of Parallel and Distributed computing 48.1 (1998): 96-129.

21. Karypis, George. "hMETIS 1.5: A hypergraph partitioning package." http://www.cs.umn.edu/~metis, 1998.