Mercury: Instruction Pipeline Aware Code Generation for Simulink Models

Zehong Yu, Zhuo Su✉, Yixiao Yang, Jie Liang, Yu Jiang✉, Aiguo Cui, Wanli Chang, Rui Wang

***Abstract*—Simulink is a widely used model-driven design environment for supporting the simulation and code generation of embedded applications. To improve the quality of the code generated from Simulink models, state-of-the-art code generators employ various high-level optimizations, like eliminating local variables. However, they overlook the compatibility between code and low-level processor architecture, especially the instruction pipeline. Consequently, instruction pipeline stalls occur frequently, leading to additional delays in instruction execution, as well as limited efficiency for deployed embedded software.**

In this paper, we propose Mercury, an instruction pipeline aware code generator for Simulink models which utilizes data dependencies between actors to decrease the instruction pipeline stalls of the generated code. First, Mercury collects data dependencies through model dataflow traversal and records the property of each actor. Then, Mercury approximately estimates the execution latency of required instructions fetched from corresponding actors and uses a topology-based method to obtain candidate actors for code synthesis. Finally, Mercury adopts the least penalty priority to iteratively select the most suitable actor for code synthesis and releases data dependencies with its subsequent actors. We implemented and evaluated Mercury on benchmark Simulink models [1] as well as a real industrial model. Compared to the official tool Simulink Embedded Coder and the state-of-the-art academic tool DFSynth, Mercury outperformed them by 9.7%-33.4% and 9.2%-59.4% in terms of the execution time of the generated code across different architectures, respectively. The statistics also demonstrate that the generated code of Mercury increases utilization of pipeline slots by 11.0%-37.1% and 10.6%-50.0%, respectively.

***Index Terms*—Simulink Models, Code Generation, Instruction Pipeline.**

I. Introduction

Simulink [2] is a widely used model-driven design environment and has become very popular in embedded scenarios such as vehicle control systems, autonomous driving, and aerospace development [3]–[5]. It offers various toolsets such as model-based simulation, verification, and code generation to facilitate the development of embedded software. Due to the resource limitation of embedded devices, code generation needs to be taken seriously since the quality of generated code directly affects the efficiency of the entire system.

Recently, some noteworthy tools for generating high-quality code have emerged in both industry and academia. In industry, for example, Simulink Embedded Coder [6] is the most widely used commercial tool. It supports the generation of production-quality code for models satisfying its modeling semantics. The quality is ensured by various optimizations, such as reusable data exploitation, local variable elimination, and SIMD (Single Instruction Multiple Data) instruction utilization. As a result, most developers can deploy the generated code into embedded devices directly without any modifications. In academics, for example, DFSynth [1] focuses on generating more efficient code for models containing complex branch actors like BooleanSwitch, while retaining the structure information of original models. Besides, DFSynth supports generating code with well-structured code templates.

However, these state-of-the-art code generators overlook the compatibility between code and low-level processor architecture, especially the **instruction pipeline**. In other words, all these generators translate the model into code containing multiple parts with data dependencies, running in serial without interleaved execution of independent instructions. Consequently, instruction pipeline stalls occur frequently. Specifically, the execution of one instruction needs to wait for the processing of other instructions, leading to extra latency for instruction execution. As a result, the efficiency of the generated and deployed embedded software is limited.

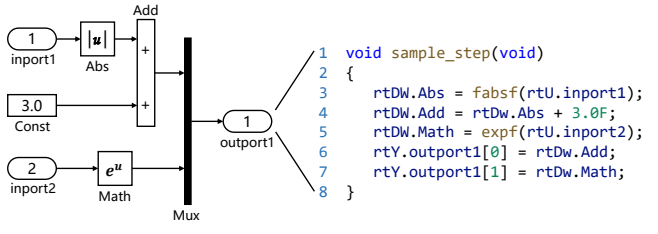


Fig. 1. A sample Simulink model and the corresponding code generated by Simulink Embedded Coder. The model combines the data from *inport1* and *inport2* after they finish relevant operations. Because of data dependencies, the generated code will cause massive instruction pipeline stalls.

For example, Simulink Embedded Coder translates the model shown in Fig. 1 in the following steps: ① Obtains data from *inport1* and calculates the absolute value of it. ② Obtains data from *Const* and adds up the result of Step 1 to

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706203, No2021QY0604) and MIIT Project(Design of intelligent networked vehicle based on SOA central control).

Z. Yu, Z. Su, and Y. Jiang are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mail: yzhddding@gmail.com).

Y. Yang, and R. Wang are with the Information Engineering College, Capital Normal University, Beijing 100048, China (e-mail: rwang04@163.com).

A. Cui is with the HUAWEI Technologies, Co. LTD., Shanghai 200120, China (e-mail: ag.cui@huawei.com).

W. Chang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China. (email: wanli.chang.rts@gmail.com).

Yu Jiang and Zhuo Su are the corresponding authors.

translate *Add* actor. ③ Obtains data from *inport2* to perform corresponding exponential operation. ④ Combines results of Step 2 and 3 into a vector output. However, it leads to a RAW (Read-After-Write) hazard since the execution of Step 2 needs to wait for the result in Step 1 (lines 3-4), which introduces instruction pipeline stalls. They can be avoided by swapping the translation order of *Math* actor and *Abs* actor.

To generate pipeline-friendly code that decreases instruction pipeline stalls during execution, we have to struggle with the following two challenges.

The first challenge arises in *constructing a pipeline-friendly translation sequence of model actors*. A well-designed sequence can reduce the occurrence of instruction pipeline stalls. Specifically, when selecting the next actor in the construction of a translation sequence, there might be multiple available options to choose from. However, a slight difference in the selection may lead to serious performance gaps in the generated code. For example, suppose we translate actors with data dependencies at the same time. Since the data needed by one actor may be occupied by others, it is not possible to execute the actors in parallel, resulting in additional overhead for the embedded software. In particular, due to the lack of processor parameters and other relevant information, the code generator cannot retrieve the execution latency of the corresponding actor and thus cannot determine whether the required data has been released by others. Therefore, it is challenging to iteratively select the most suitable actor for the translation sequence to improve the performance of the generated code.

The second challenge arises in *maintaining the semantic consistency between the pipeline-friendly translation sequence and the model*. For code generation, it is of paramount importance to ensure the consistent semantics of the model and generated code. However, adjusting the translation order of actors into pipeline-friendly code may break the semantic consistency. For example, when swapping the code in line 3 and line 7 of Fig. 1, the instruction pipeline stall can be avoided, but the generated code will output the wrong result. In general, to maintain semantic consistency, code generation is guided by the control logic of the entire model analyzed from the dataflow graph. But the increasing complexity of models and rich modeling semantics result in difficulties in collecting and analyzing data dependencies among actors. For example, circular data dependencies may exist within the Simulink model. They cause infinite loops in fully collecting data dependencies. More importantly, without precise data dependencies, the code generator is unable to generate code that maintains the original semantics.

To address the aforementioned challenges, we propose Mercury, an instruction pipeline aware code generator for Simulink models. Mercury mainly consists of three steps. First, Mercury traverses the model dataflow graph to collect data dependencies between actors and records the property of each actor. Second, Mercury approximately estimates the execution latency of each actor based on the complexity of corresponding instructions and processor parameters. With the analysis of collected data dependencies through a topology-based method, Mercury obtains the candidate actors in each iteration of translation while retaining the original model semantics. Finally, according to the gathered information, Mercury adopts the least penalty priority to iteratively select the most suitable actor in generating a translation sequence, avoiding the occurrences of instruction pipeline stalls, thus improving the execution efficiency of the generated code. After that, Mercury releases data dependencies pointed from the selected actor and assigns the penalty value for actors.

We implemented Mercury and evaluated it on benchmark Simulink models [1] as well as a real industrial model. The results demonstrate that Mercury gains great performance. Compared to the official tool Simulink Embedded Coder and the state-of-the-art academic tool DFSynth, the code generated by Mercury achieved an improvement of 10.0%-33.4% and 12.1%-59.4% in terms of execution time on Intel x86 architecture processor, respectively. Furthermore, improvements to other tools in extended experiments on ARM architecture processors show that Mercury is also cross-architecture compatible. More importantly, according to statistics from Intel VTune profiler, compared with Simulink Embedded Coder and DFSynth, Mercury increases the utilization of pipeline slots in 11.0%-37.1% and 10.6%-50.0%.

In summary, this paper makes the following contributions.

- We identify that state-of-the-art code generators do not take full advantage of the instruction pipeline, resulting in frequent pipeline stalls in the generated code, which leads to additional execution time and limited performance of embedded software.
- We implement Mercury, an instruction aware code generator for Simulink models. Mercury traverses model dataflow to collect data dependencies and estimates the execution latency of each actor. Leveraging the collected information, Mercury generates instruction pipeline-friendly code.
- We apply Mercury on a benchmark of 10 commonly used Simulink models and a real industry model. The results show that Mercury outperformed Simulink Embedded Coder and DFSynth across different architectures.

II. Background

A. Code Generation for Simulink Model

Code generation is of vital importance in model-driven design tools, which releases the developers from error-prone manual coding. For the constructed Simulink model, developers can adopt various code generators to generate production-level code that can be directly deployed to embedded devices, facilitating the efficiency of embedded software development [1], [7], [8]. The code generation process of most code generators can be divided into the following steps [9]. First, they transfer the model file into a customized intermediate representation to record the corresponding information, such as actor details, data dependencies, inports, and outports. Then, they extract the scheduling relations between actors. Finally, they perform code synthesis for each actor and integrate them according to the scheduling relations.

Since there exists a trade-off between capacity and economics for embedded devices, the quality of the generated code is critical to improving the overall performance of

the embedded system [10]. The execution efficiency of the generated code is one of the primary indicators of code quality, and there have been some noteworthy efforts trying to improve it. They attempt various optimization techniques for code generation, such as reusable data exploitation, SIMD instruction utilization, and branch elimination [6]. These techniques mainly focus on the code itself, resulting in a bottleneck in making further improvements. However, with the development of processor architecture, numerous processor features may bring opportunities to improve the performance of code generation [11]. For example, if we can break some data hazards within the generated code, it will improve the utilization of processors and reduce instruction pipeline stalls, thus improving the overall efficiency of the embedded system.

B. Instruction Pipeline

Instruction pipeline is a subtle design inside the processor to improve the efficiency of the processor in executing instructions [12], [13]. In general, depending on the operations, it can be divided into two parts, namely frontend and backend. The frontend consists of two stages, namely instruction fetching and instruction decoding. The backend mainly contains three stages, namely execution, memory access, and write-back. Each stage has a special processing unit, allowing the parallel execution of different operations in the multi-stage pipeline architecture [11]. Modern processors divide one of the stages into more smaller stages to introduce architecture optimization, which makes the pipeline more complex. For example, to implement speculative execution, reading/writing register file and calculating branch jump address are separated from the instruction decoding stage. Furthermore, the multiple instruction issue mechanism is introduced to achieve higher performance by increasing the width of the instruction pipeline [14]. It allows multiple instructions to be fetched and decoded simultaneously at the frontend of the instruction pipeline. In this way, several instructions with no data dependency among them can be issued and completed on the corresponding functional units at the backend of the instruction pipeline at every clock cycle.

The instruction pipeline stall represents that the execution of some instructions needs to wait for the processing of other instructions [11]. The instruction pipeline stall inside the deployed code will prevent subsequent instructions from being issued to the corresponding processing units for parallel execution, thus decreasing the overall performance. Data hazard is one of the main factors causing the instruction pipeline stall, occurring when instructions with data dependencies access or modify the same data in different stages of the instruction pipeline [15]. It can be divided into three kinds of data hazards: RAW (Read-After-Write) hazard, WAR (Write-After-Read) hazard, and WAW (Write-After-Write) hazard [16]. For example, RAW hazard occurs when the latter instruction attempts to read the data before the former instruction writes it. Therefore, it is important to decrease the data hazards in code generation, which is supposed to be able to improve instruction pipeline utilization during the execution of the generated code.

III. Motivation

State-of-the-art code generators such as Simulink Embedded Coder generally translate the Simulink model into embedded code based on the data dependencies among actors strictly, as shown in Fig. 1. Basically, they iteratively select an actor without any data dependency and its subsequent actors for translation until there are no subsequent actors or the subsequent actors have additional data dependencies. Then they search for other available actors and repeat the aforementioned step. We observe that the aforementioned steps for code generation lead to the execution of latter actors heavily depending on the result produced by the former actors, introducing a significant amount of instruction pipeline stalls. Instruction pipeline stalls inside the generated code cause less instructions to be issued simultaneously by the processor, which decreases the utilization of pipeline slots, thus bringing a negative impact on the performance of the generated code. More specifically, when deployed in embedded devices with limited capability, the generated code experiences decreased performance.

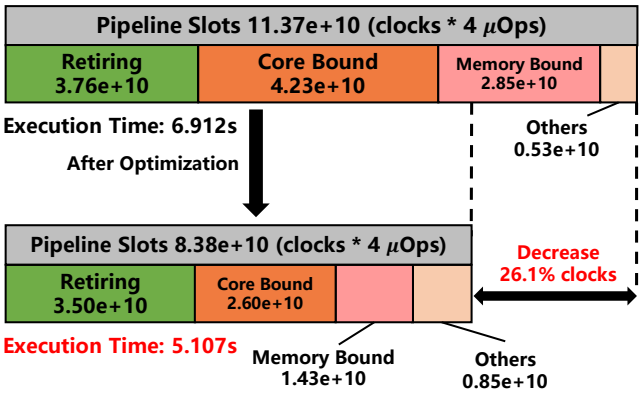


Fig. 2. Performance profiling results of the code generated by Simulink Embedded Coder and the code optimized by the idea of Mercury from Intel VTune Profiler. For processors under CISC architectures, the macro instructions are broken down into several micro-operation (μOps) for execution. The pipeline slots represent the number of μOps that can be issued simultaneously at all clock cycles. The retiring chunk represents the fraction that the issued μOps get executed. That is, it represents the fraction of pipeline slots utilized by useful work. Memory bound chunk represents the fraction with stalls caused by the corresponding LOAD or STORE instructions. Core Bound chunk mainly results from the shortage in hardware devices. The others chunk represents frontend bound and bad speculation. The result illustrates that avoiding the instruction pipeline stalls can improve the efficiency of the generated code greatly.

To quantitatively understand the severity of the aforementioned problem, we analyze the pipeline slots utilization of code generated by Simulink Embedded Coder, according to the performance metrics obtained from Intel VTune Profiler. The pipeline slots are divided into the following chunks: retiring, memory bound, core bound, and others. Since the retiring chunk represents the fraction of pipeline slots that the issued μOps eventually get executed, we use it as the indicator to evaluate the utilization of pipeline slots. Take a sample Simulink model as an example, Fig. 2 demonstrates the performance profiling results of the code generated by Simulink Embedded Coder. The statistics show that only roughly one-third of pipeline slots eventually retired the issued μOps, incurring significant overhead for execution.

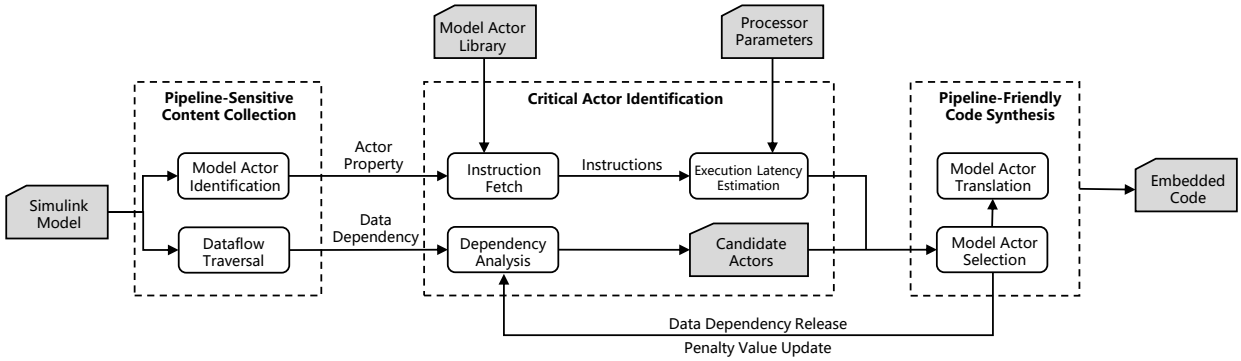


Fig. 3. Overall Framework of Mercury. 1) In the first step, Mercury obtains the corresponding property of each actor and data dependencies between actors. 2) In the second step, Mercury fetches the needed instructions which are used to execute the corresponding actor and approximately estimates the execution latency of it. Then, Mercury obtains the candidate actors without data dependency pointing from others. 3) In the third step, Mercury adopts the least penalty priority to iteratively select the most suitable actor for translation. After that, Mercury releases data dependencies pointing from the selected actors and updates the penalty value of actors. Finally, Mercury performs code synthesis for each actor in translation sequence for generating pipeline-friendly code.

Since Mercury avoids pipeline stalls by utilizing data dependencies, it increases μOps that can be processed simultaneously. However, when the number of instructions exceeds the capacity of the pipeline's frontend, it leads to excessive fetch latency without sending μOps in time or insufficient decode capacity to fully utilize the fetch bandwidth, thus increasing the fraction of others chunk.

Basic Idea of Mercury. The Simulink model contains abundant data dependencies. Mercury addresses the aforementioned problem by fully utilizing those information to avoid data hazards, improving the execution efficiency of the generated code. The main idea is that, after selecting an actor for translation, it will not select its subsequent actors directly but the other actors without data dependency for translation to decrease the occurrence of instruction pipeline stalls. The statistics in Fig. 2 illustrate that based on the optimization of Mercury, the execution time of the code generated by the optimized engine decreases by 26.1% compared with the original code generated by Simulink Embedded Coder.

IV. Mercury Design

In this section, we detail the design of Mercury. Mercury takes Simulink model, model actor library, and target processor parameters as input and generates pipeline-friendly embedded code, which can be directly deployed on embedded devices with less instruction pipeline stalls. Fig. 3 shows the overall framework of Mercury. It mainly consists of three steps: Pipeline-Sensitive Content Collection, Critical Actor Identification, and Pipeline-Friendly Code Synthesis.

In the first step, Mercury identifies and records the property of each actor in the Simulink model. Then, Mercury traverses the model dataflow to obtain data dependencies between actors. In the second step, Mercury fetches the necessary instructions which are used to execute the corresponding actor from the model actor library. Based on the execution latency of each instruction obtained from processor parameters, Mercury approximately estimates the execution latency of actors. Then, Mercury adopts a topology-based method to analyze data dependencies to obtain the candidate actors in each iteration of the translation process. The candidate actor means that there is no data dependency pointing to it. In the third step, according to the collected information in the second step, Mercury adopts the least penalty priority to iteratively select the most suitable actor in generating translation sequences. Then, Mercury releases data dependencies pointing from the selected actor and updates the penalty value for its subsequent actors and remaining candidate actors. Finally, Mercury performs code synthesis for each actor in translation sequence for generating pipeline-friendly embedded code, decreasing the occurrences of instruction pipeline stalls.

A. Pipeline-Sensitive Content Collection

Mercury first parses the given model into a well-structured IR (Intermediate Representation) as the preparation step. The IR contains the inports, outports, actors, relations between actors, etc. Based on the information contained in IR, especially relations, Mercury extracts the model dataflow. After that, Mercury traverses the model dataflow to get data dependencies between actors and filters the properties of actors in IR, such as actor type, parameters, inports, and outports. Furthermore, the detailed information inside each element of IR will be retained for further analysis. For example, as shown in Fig. 4, the element "Actor" in IR contains the information of its name, type, inports, and outports, which will be used for generating code for this actor.

The overall procedure of collecting pipeline-sensitive content inside Simulink models is shown in Algorithm 1. Before collecting data dependencies and identifying the corresponding actors, Mercury needs to break the circular data dependencies inside the Simulink model. This causes infinite loops in fully collecting data dependencies. Since there exists a data delay actor for each back edge of the loop in the Simulink model, we can simply break the loop by dividing the actor into two parts: data storage and data fetch. For example, suppose $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_0$ is a loop inside dataflow, and a_2 is the data delay actor. Mercury divides a_2 into a_2' and a_2'', a_2' points to a_0 and a_2'' is pointed by a_1. After that, the loop becomes a sequence $a_2' \rightarrow a_0 \rightarrow a_1 \rightarrow a_2''$. Then, Mercury adopts BFS method for each *inport* of the model to traverse the entire dataflow (lines 4-17). Mercury obtains the subsequent actors

```
<Root>
   <!-- Inports and Outports-->
   <Inport Name="In1" Type="i32" ArraySize="32"/>
   <Outport Name="Out1" Type="i32"/>
   <Outport Name="Out2" Type="i32"/>
   <!-- Actors -->
   <Actor Name="Add" Type="Sum">
      <Inport Name="Inport1" Type="i32"/>
      <Inport Name="Inport2" Type="i32"/>
      <Outport Name="Outport1" Type="i32"/>
      <Parameter Name="Inputs" Value="+-"/>
   </Actor>
   <!-- Relations -->
   <Relation>
      <Src Src="Product.#DefaultOutport1"/>
      <Dst Dst="Add1.#DefaultInport2"/>
   </Relation>
   <!-- Other model elements-->
   ...
</Root>
```

Fig. 4. A fragment of IR. It contains the inports and outports, actors, relations, and other useful elements related to corresponding Simulink models. The inside of each element in IR contains detailed information.

of model's *inport* by searching in dataflow and pushes them into *queue* which stores actors that need to be traversed. After that, Mercury selects the top actor in the *queue* and obtains its subsequent actors for collecting data dependencies until the *queue* is empty (lines 6-7). Mercury collects data dependencies by adding a former actor into the corresponding entry of *Dependency*. For instance, assume that $\{a_1, a_2\}$ is a collection of a_0's subsequent actors, then Mercury adds a_0 to the entry $Dependency[a_1]$ and $Dependency[a_2]$, respectively. For each actor, Mercury uses *ActorProp* to record the properties of it by filtering the content of IR for further usage (lines 19-22). To perform the aforementioned steps efficiently, *ActorProp* and *Dependency* are designed as *HashMap* structures. Finally, after all the actors have been traversed, the collected information will be further analyzed by Mercury to avoid the occurrence of instruction pipeline stalls, thus improving the execution efficiency of the generated code.

B. Critical Actor Identification

In order to iteratively select the most suitable actor for code synthesis, Mercury should approximately estimate the execution latency of each actor and adopt a topology-based method to obtain the candidate actors that have no data dependency pointing from others. Algorithm 2 shows the overall process of Critical Actor Identification.

Instruction Fetch. Mercury supports a well-structured library of actors represented in XML format to fetch the required instructions for executing actors. Mercury first filters the model actor library according to the type of each actor. Then, Mercury matches the type of inports and outports with filtered information and obtains the instructions inside it. More specifically, for the array type of inports and outports, Mercury repeats the obtained instructions based on the length of I/O ports. Taking the actor in Fig. 4 as an example, assume Mercury needs to fetch the instructions of it. Since the actor type is "Sum", the required operations should be "Add" or

Algorithm 1: Pipeline-Sensitive Content Collection

Input: $Dataflow$: Dataflow of the model
$Inports$: Inports of the model
IR: Intermediate Representation of the model
Output: $ActorProp$: HashMap (actor → actor's property)
$Dependency$: HashMap (actor → actors that points to key actor)

```
1  breakLoop(Dataflow)
2  actors = ∅
3  // Dataflow traversal
4  for inport in Inports do
5      queue.push(getSubsequentActors(inport, Dataflow))
6      while queue ≠ ∅ do
7          current = queue.pop()
8          subsequents = getSubsequentActors(current,
             Dataflow)
9          for actor in subsequents do
10             if actor ∉ actors then
11                 queue.push(actor)
12             end
13             // Collect data dependency
14             Dependency[actor].add(current)
15         end
16     end
17 end
18 // Model actor identification
19 for actor in actors do
20     // Filter properties of actor in IR
21     ActorProp[actor] = IR.filterProperty(actor)
22 end
```

"Sub". The type "i32" demonstrates that the input value and output value of the actor are 32-bit integers. The parameter value "+-" reflects that the operation of the first inport is "Add" and the operation of the second inport is "Sub". Therefore, the operation of this actor is to output the value of subtracting its two inports. After that, Mercury matches the actor's operation and data type with the model actor library to fetch the required instructions, as shown in Fig. 5.

```
<Root>
   <Actor Name="Sub" Type="i32">
      <Instruction Name="MOV" Operand="reg,mem"/>
      <Instruction Name="SUB" Operand="reg,mem"/>
      <Instruction Name="MOV" Operand="mem,reg"/>
   </Actor>
</Root>
```

Fig. 5. An example of the model actor library. This example represents the required instructions for a Sub actor. It contains two MOV instructions and a SUB instruction for the execution of the Sub actor.

Execution Latency Estimation. The statistics from [17] is used as processor parameters to assist Mercury to approximately estimate the execution latency of obtained instructions. Table I shows the latency of some typical instructions in x86 architecture. The latency required for each instruction may be different due to the usage and operands of the corresponding instruction. For example, the two MOV instructions shown in Table I have different latencies. The operand of the latter instruction needs to access memory for the required data, resulting in extra overhead than accessing registers, thus having more latency than the former instruction. Besides, the execution latency of the same instruction may vary in different architectures. Therefore, Mercury should precisely match instruction's operands and target processor architecture

with processor parameters to obtain the required execution latency. After that, for each fetched instruction, Mercury adds up the estimated latency and stores it into a corresponding data structure for further usage (lines 5-9).

TABLE I
THE LATENCY OF TYPICAL INSTRUCTIONS.

Instruction	Latency	Operands
MOV reg, i	1	register, immediate data
MOV reg, mem	3	register, memory
ADD reg, reg	1	register, register
SUB reg, reg	1	register, register
PUSH mem	3	memory
MUL reg	3	register
DIV reg	19	register

Data Dependency Analysis. Each data dependency inside the Simulink model has its direction representing the data transfer from the outport of one actor to the inport of another actor. However, the mistake order of translation sequence without conforming to these data dependencies results in functional inconsistencies between the model and generated code. For example, suppose Mercury translates the *Abs* actor after the *Add* actor for the model shown in Fig.1. The meaning of the generated code becomes calculating the absolute value of *inport1* after adding up the value from *Abs* with a constant. This will lead to inconsistent results between the execution of the generated code and the simulation of the model. Therefore, it is of paramount importance to confirm the data dependencies inside Simulink models for generating a translation sequence correctly with retaining the original modeling semantics. Mercury adopts a topology-based method to obtain the candidate actors to address the aforementioned problem. In each iteration of generating translation sequence, Mercury traverses the $Dependency$ mentioned in Algorithm 1. For actors that are without any data dependencies, Mercury marks it as a candidate and adds it to the actor list for selecting the most suitable actors in model translation (lines 13-17). In this way, Mercury ensures that the actor is unable to execute until obtaining the required data, thus guaranteeing the correctness of the translation sequence.

C. Pipeline-Friendly Code Synthesis

Mercury applies the least penalty priority to generate the translation sequence. The objective of Mercury is to iteratively select the most suitable actor that has the minimum penalty value. After that, Mercury updates the penalty value for candidate actors and assigns the penalty value for subsequent actors of the selected actor, used in the next iteration. Finally, Mercury performs code synthesis to generate pipeline-friendly code for the target Simulink model according to the order of obtained translation sequence.

Model Actor Selection. Before performing code synthesis for each actor, Mercury iteratively selects the most suitable actor for generating the translation sequence. Algorithm 3 presents the overall procedure of model actor selection. It primarily consists of the following steps: Suitable Actor Selection, Penalty Value Update, and Data Dependency Release.

Algorithm 2: Critical Actor Identification

Input: $ActorList$: Actors that required translation
$ActorProp$: HashMap (actor $\rightarrow$ actor's property)
$ProcessorInfo$: Target processor information
$Dependency$: HashMap (actor $\rightarrow$ actors that are dependent on key actor)
Output: $Latency$: HashMap (actor $\rightarrow$ actor's execution latency)
$Actors$: Candidate actors

```
1  for actor in ActorList do
2      latency = 0
3      // Fetch corresponding instructions from actor
4      Instructions = loadActorLibrary(ActorProp[actor])
5      for instruction in Instructions do
6          // Estimate required clock cycles of instruction
7          latency+ = getLatency(instruction,ProcessorInfo)
8      end
9      Latency[actor] = latency
10 end
11 // Find actors that have no dependencies pointing by others
12 Actors = ∅
13 for actor in Dependency do
14     if Dependency[actor] = ∅ then
15         Actors.add(actor)
16     end
17 end
```

1) Suitable Actor Selection: In most cases, there may exist multiple candidate actors that have no data dependencies pointing from others in each iteration of generating a translation sequence. Mercury adopts the least penalty priority to iteratively select the most suitable actor, i.e. selecting an actor with the minimum penalty value. First, Mercury obtains the candidate actors based on data dependency analysis according to the topology-based method in Algorithm 2. Then, Mercury defines a variable to store the minimum penalty value of candidate actors. For each candidate actor, if its penalty value is lower than the current minimum penalty value, Mercury will replace the most suitable actor with the current one and reassign the minimum penalty value (lines 11-17).

2) Penalty Value Update: Since the data required by the actor may be occupied by others, Mercury introduces the penalty value to measure the needed clock cycles to obtain the required data. Before assigning the penalty value for candidate actors and subsequent actors respectively, Mercury updates the execution latency of the selected actor. Since the selected actor may have a positive penalty value representing the required data that has not been released, the execution latency should be added to the penalty value of the selected actor to measure the clock cycles for execution precisely (lines 20-22). For each candidate actor, Mercury subtracts the execution latency of the selected actor from its penalty value to eliminate the impact of the selected actor on it (lines 24-26). As for the subsequent actor, since the data occupied by the selected actor may be released before other required data, i.e. $Penalty[actor] > Latency[suit]$, Mercury should compare the penalty value with the execution latency of the selected actor and assign the larger one as the penalty value of the subsequent actor. Based on the penalty value of candidate actors and subsequent actors, Mercury can apply the least penalty priority to generate a translation sequence that makes full use of the instruction pipeline.

3) Data Dependency Release: The data dependency inside

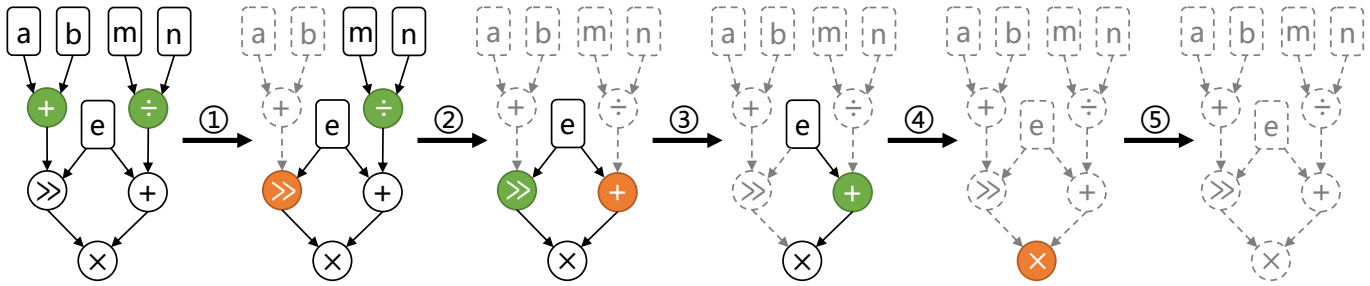


Fig. 6. An example illustrates the process of generating the translation sequence. The green actor represents the candidate actor for generating the translation sequence. The orange actor means that it has a penalty value. In each step of selecting the most suitable actors, Mercury picks an actor with the minimum penalty value among the green actors and assigns a penalty value for its subsequent actors turning the color to orange.

Simulink models represents the data transfer from one actor to another, resulting in the execution of the latter actor requiring the data generated by the former actor. Therefore, Mercury needs to generate a translation sequence confirming the order of data dependencies. In each iteration, after selecting the suitable actor, Mercury obtains the subsequent actors of it. Then, for each subsequent actor, Mercury removes the selected actor from the corresponding entry of *Dependency* to release the useless data dependencies (line 29). The removed data dependencies will contribute to different results in the next iteration of obtaining the candidate actors.

According to the aforementioned collected information, Mercury can use the penalty value as the guidance to select the most suitable actor for translation and generate pipeline-friendly embedded code. Besides, Mercury follows the data dependencies to ensure the correctness of generated code. Based on the least penalty value priority, Mercury can reduce the occurrences of actors waiting for the required data to execute, thus minimizing the frequency of instruction pipeline stalls and improving the efficiency of the generated code.

We take the example shown in Fig. 6 to illustrate the process of generating a translation sequence. The green actors in the figure represent the candidate actors while the orange actors represent candidate actors with a penalty value. When the green actor has been selected to translate, Mercury releases data dependencies pointing from this actor. Then Mercury assigns a penalty value for the subsequent actors and updates the penalty value of candidate actors. For example, in step ②, Mercury selects the *Div* actor for translation and releases the data dependency with the subsequent *Add* actor. After that, Mercury updates the penalty value of *Shift* actor turning the color to green and assigns a penalty value to *Add* actor. Finally, Mercury will iteratively perform the aforementioned steps until all of the actors and data dependencies have been released. Code generation with such translation sequence can effectively decrease the occurrence of instruction pipeline stalls, thus improving the execution efficiency of the generated code.

Model Actor Translation. The code synthesis for each actor in Mercury is basically consistent with Simulink Embedded Coder and DFSynth. Based on the obtained translation sequence, Mercury iteratively matches each actor with model IR to obtain detailed information such as actor type, inports, and outports. Then, Mercury supports well-defined DLL (Dynamic Link Library) files for different actors to generate the corresponding code. Since the same type of actors may have different detailed information, resulting in differences in the generated code. For example, the generated code of *Add* actors with different inport types are not the same, e.g., int and float. Therefore, Mercury should configure the aforementioned information as parameters of the corresponding DLL file to obtain the required code precisely. After that, according to the order of the translation sequence, the generated code of actors is synthesized to form the function code of the model. Other relevant information is encapsulated into some header files for calling by the main file.

Algorithm 3: Model Actor Selection

Input: $ActorList$: Actors that required translation
$Latency$: HashMap (actor → actor's execution latency)
$Dependency$: HashMap (actor → actors that point to key actor)
Output: $Sequence$: Translation sequence

```
1  // Assgin initial penalty value
2  Penalty = ∅
3  for actor in ActorList do
4  |  Penalty[actor] = 0
5  end
6  while ActorList ≠ ∅ do
7  |  // Obtain candidate actors
8  |  Actors = getCandidateActors()
9  |  min = MAX
10 |  suit = Actors[0]
11 |  for actor in Actors do
12 |  |  penalty = Penalty[actor]
13 |  |  if penalty < min then
14 |  |  |  min = penalty
15 |  |  |  suit = actor
16 |  |  end
17 |  end
18 |  Actors.remove(suit)
19 |  // Update the exection latency of the selected actor
20 |  if min > 0 then
21 |  |  Latency[suit] += min
22 |  end
23 |  // Update penalty value for available actors
24 |  for actor in Actors do
25 |  |  Penalty[actor] -= Latency[suit]
26 |  end
27 |  // Update data dependency and penalty for subsequent actors
28 |  for actor in getSubsequentActors(suit) do
29 |  |  Dependency[actor].remove(suit)
30 |  |  Penalty[actor] = min(Penalty[actor], Latency[suit])
31 |  end
32 |  Sequence.push(suit)
33 |  ActorList.remove(suit)
34 end
```

The detailed procedure of code synthesis is as follows. Mercury first encapsulates the Simulink model into a well-defined function, while the inports and outports of the model are stored as structs. For example, a model named *Kalman* will be encapsulated in the following function header: *void Kalman_step(void)*. The structs named *Kalman_In* and *Kalman_Out* are used to store the model's inports and outports, respectively. Then, Mercury uses several lines of code to describe the type of each actor according to its complexity and adds the generated code to the function. The inports of the corresponding actor are used as operands in the generated code. For example, an *Add* actor accepts data from a *Shift* actor and a *Mul* actor. Then it performs the addition operation. The corresponding code is as follows: *Kalman_Add = Kalman_Shift + Kalman_Mul*. The code for each actor is synthesized according to the order of translation sequence and data dependencies between actors to conform the original modeling semantics. Next, Mercury adopts global variables to describe the states of the model and other relevant information varies for different models. Two functions are supported to initialize and release these variables respectively. A header file is used to store the global variables, utility functions, and other definitions such as struct definition. Finally, Mercury declares a main function to execute the aforementioned functions and includes the header files to compose the main file.

V. Evaluation

Tool Implementation: We implemented Mercury using over 20000 lines of C++ code. We have supported code generation for Stateflow modeling semantic and Dataflow modeling semantic. We support almost all algebraic actors, e.g., *Add* actor, *Product* actor, and *Abs* actor. For each supported actor, we define a corresponding DLL template for code generation. It describes the functionality of the actor and other relevant information, i.e., actor name, data type, actor parameters, inports and outports. For subsystems of Simulink models, we define a unique function and perform code generation for each actor inside the subsystem to implement the defined function.

Experiment Setup: We evaluated the performance of Mercury on a benchmark of 10 commonly used Simulink models [1] as well as a real industrial model. For benchmark models, ABS is a safety anti-skid system model used on vehicles. BandStop, HighPass, and LowPass are filter models which are mainly used to filter undesired parts of the signal. PID is a controller model that is widely used in industrial control systems; Simpson is a numerical integration model using Simpson's rule [18]. Besides, NLGuida is a nonlinear algorithm model for aviation guidance with 16 subsystems and 282 actors, while those models distributed with Simulink contain about 30 actors. AN, Gamma, and Hybrid are multi-rate models for different scenarios. For the industrial model, it is an automotive temperature control module in a vehicle control system. To investigate the effectiveness, we compared Mercury with two state-of-the-art code generators, the official Simulink Embedded Coder [6] and the academic DFSynth [1]. We first collected the execution time of the generated code on different architectures such as Intel x86 and ARM Cortex. Second, the utilization improvement of pipeline slots was also analyzed in-depth. Since Simulink Embedded Coder is a built-in tool for Simulink, we use Simulink for short.

A. Effectiveness on Benchmark Models

To validate the performance of the generated code by Mercury, we first conducted the experiment on the benchmark models. The experiment environment was an industrial machine running Ubuntu 20.0 x64 (Intel J3160 1.6GHz, 4GB RAM). The compiler we used was the most commonly used GCC (v9.4.0) with default compile options (`gcc *.c -o out`). The code generated by Simulink, DFSynth, and Mercury were executed with the same times of 10000 for average results to establish the statistical significance of the results. While conducting the experiments, we also verified the correctness of the generated code by comparing the output values between the simulation and execution.

Table II shows the experiment results and the percentage of improvement. Overall, there is a 10.0%-33.4% and 12.1%-59.4% performance improvement for the execution of the code generated by Mercury, when compared with the code generated by Simulink and DFSynth, respectively.

TABLE II
Comparison of the code execution time on Intel

Model	Simulink	DFSynth	Mercury	Improvement	
				Simulink	DFSynth
ABS	0.519s	0.551s	0.448s	13.7%	18.8%
BandStop	0.315s	0.396s	0.243s	22.5%	38.5%
HighPass	0.600s	0.683s	0.510s	15.0%	25.3%
LowPass	0.579s	0.646s	0.484s	16.3%	25.0%
PID	0.486s	0.908s	0.381s	21.6%	58.1%
Simpson	0.782s	0.986s	0.521s	33.4%	47.2%
Hybrid	0.984s	1.158s	0.855s	13.1%	26.2%
NLGuida	4.089s	5.049s	3.216s	21.4%	36.3%
AN	7.521s	7.696s	6.767s	10.0%	12.1%
Gamma	2.024s	4.288s	1.741s	14.0%	59.4%

The performance of the code generated by DFSynth is relatively limited, mainly because it generates variables to store data for the parameter and the output of each actor in the model. Each time the code is executed, these variables are created in the stack space, resulting in performance degradation, especially when there are a large number of input and output ports and actors in the model. As for Simulink, it can fold some expressions and reuse some output variables. Although the code is optimized, the data hazards between expressions incur massive instruction pipeline stalls, thus resulting in the inefficiency of the generated code. In contrast, higher code performance can be obtained by adjusting the actor translation order to reduce the instruction pipeline stalls. For example, the Simpson model calculates the function value of the midpoint of the interval and two endpoints for numerical integration. These operations can be performed in parallel, which allows for a high degree of parallelism at the actor level of the model. We can generate code from actors without dependencies together, which allows us to better utilize the pipeline slots. Therefore, Mercury can get performance improvements of 33.4% and 47.2% compared to Simulink and DFSynth.

Furthermore, to demonstrate the effectiveness of Mercury on different architectures, we also compiled the code with GCC on a machine with an ARM processor (ARM Cortex A72 1.5 GHz) and repeated the above experiments. The result on ARM is shown in Table III. We can see that Mercury can achieve 9.7%-21.5% and 9.2%-37.2% performance improvement compared to Simulink and DFSynth. Comparing Table II and Table III, that is, the experimental results on Intel and ARM, we can find that the execution efficiency of the code generated from the same model on different processor architectures is different. This is mainly because ARM processors and Intel processors have different memory access strategies. ARM uses a weakly-ordered memory model, while Intel has a strongly-ordered memory model. Besides, Mercury is not only able to achieve the improvement on Intel and ARM processors but also on other architectures. The only thing we need to do is to prepare processor-specific instruction information for code generation to make Mercury extend to the target architecture.

TABLE III
COMPARISON OF THE CODE EXECUTION ON ARM

Model	Simulink	DFSynth	Mercury	Improvement	
				Simulink	DFSynth
ABS	0.674s	0.657s	0.588s	12.7%	10.5%
BandStop	0.498s	0.571s	0.437s	12.2%	23.5%
HighPass	0.488s	0.638s	0.427s	12.5%	33.0%
LowPass	0.460s	0.544s	0.394s	14.2%	27.5%
PID	0.510s	0.572s	0.400s	21.5%	30.0%
Simpson	0.794s	0.999s	0.627s	21.0%	37.2%
Hybrid	0.824s	1.095s	0.737s	10.5%	32.6%
NLGuida	4.829s	4.688s	4.257s	11.8%	9.2%
AN	9.213s	8.676s	7.334s	20.3%	15.4%
Gamma	2.063s	2.917s	1.863s	9.7%	36.1%

Besides, we compared our approach with corresponding GCC optimization (`-fschedule-insns`), as shown in Fig 7. That is, for code generated by Simulink and DFSynth, we compiled them with this flag. The statistics with suffix (`-insns`) means the performance improvement after adding the optimization flag. Compared with the code generated by DFSynth, Mercury still can achieve superior results, mainly because of the way it generates code, as mentioned above. Compared with the code generated by Simulink, Mercury achieved better performance on 6 models, while performing similarly on the other 4. According to the GCC documentation and implementation, it focuses on reordering memory-load instructions to avoid pipeline stalls due to required data being unavailable. However, Mercury considers not only this, but also the data-processing instructions, which may avoid more pipeline stalls and achieve better performance.

B. Utilization of Pipeline Slots

We adopted the performance analysis engine Intel VTune Profiler to analyze the utilization of pipeline slots, to further validate the effectiveness of our method, i.e. whether it can decrease the occurrence of instruction stalls to improve the performance of deployed software.

Table IV demonstrates the utilization of pipeline slots of the generated code by different tools. It shows that Mercury

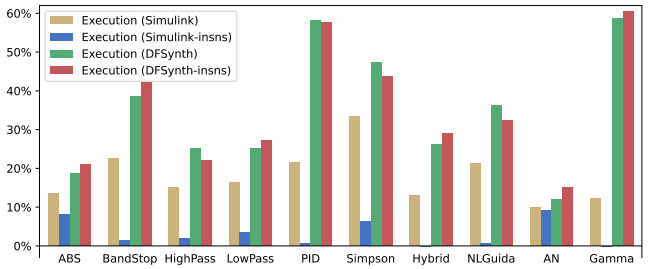


Fig. 7. The comparison of Mercury with the corresponding optimization implemented by GCC. The suffix (`-insns`) means the generated code is compiled with `-fschedule-insns` flag. The histogram shows the improvement percentage in execution time.

TABLE IV
COMPARISON OF THE UTILIZATION OF INSTRUCTION PIPELINE

Model	Simulink	DFSynth	Mercury	Improvement	
				Simulink	DFSynth
ABS	44.6%	40.2%	51.0%	14.6%	27.1%
BandStop	47.8%	40.4%	58.4%	22.2%	44.6%
HighPass	46.1%	42.7%	54.2%	17.6%	27.0%
LowPass	47.7%	45.7%	54.5%	14.3%	19.3%
PID	48.2%	41.7%	62.5%	29.7%	50.0%
Simpson	24.5%	22.5%	33.6%	37.1%	49.3%
Hybrid	49.9%	48.8%	55.4%	11.0%	13.5%
NLGuida	24.2%	22.1%	29.3%	21.1%	32.6%
AN	27.0%	27.3%	30.2%	11.9%	10.6%
Gamma	25.1%	24.4%	29.2%	16.3%	19.7%

achieved better performance than Simulink and DFSynth, and the code generated by Mercury improved the utilization of pipeline slots by 11.0%-37.1% and 10.6%-50.0%, respectively. Specifically, compared to other models, the model named Simpson mainly consists of floating-point operations that consume lots of clock cycles and introduce massive instruction pipeline stalls at the backend of the instruction pipeline. Consequently, it has relatively low utilization of pipeline slots.

To confirm that the root cause of the performance improvement brought by Mercury is decreasing the occurrence of instruction pipeline stalls, we further performed a correlation analysis on the results of the execution improvement and utilization improvement, as demonstrated in Fig. 8. We obtain correlation coefficients of 0.94 and 0.60 between the execution performance improvement values and the pipeline slots utilization of pipeline slots improvement values for Simulink and DFSynth, respectively. The correlation coefficient of 0.94 shows that Mercury actually works to improve the execution efficiency due to the improved pipeline slots utilization.

As for the correlation coefficient of the comparison with DFSynth, it is mainly weakened by the slight difference in code generation for constants. Using the PID model as an example, the huge execution time improvement is not only due to the increased pipeline slots utilization, but also the replacement of time-consuming instructions with faster instructions. Analyzing the compiled assembly code, we found that the division operations in the code generated by Simulink and Mercury are compiled to faster operations such as multiplication, subtraction, and bit shift when the divisor is a constant. In contrast, DFSynth translates the constant actor into a variable,

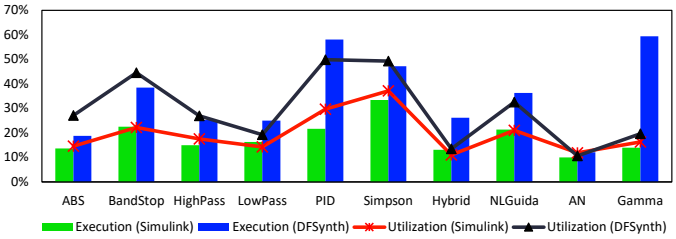


Fig. 8. The comparison of correlation between execution improvement and utilization improvement. The histogram shows the percentage the improvement in execution time. The line chart shows the percentage improvement in the utilization of pipeline slots.

causing the division code to be compiled as an inefficient DIV instruction. In addition, some multiplication operations also have been compiled as left shift instructions and additional instructions. To eliminate the effects caused by the compiler, we manually performed constant replacement on the code generated by DFSynth. Then the experiments were repeated for execution and pipeline slots utilization of the code. The correlation between the new code performance improvement and the pipeline slots utilization improvement is shown in Fig. 9, and the new correlation coefficient is improved to 0.91 from the previous 0.60, which can demonstrate their positive relativity. The rest of the gap mainly comes from the model named ABS which contains a relatively large number of ports. Mercury and Simulink use global variables to store the value of inports and outports. However, DFSynth represents these ports as function parameters in the generated code, incurring additional overhead in parameter passing.

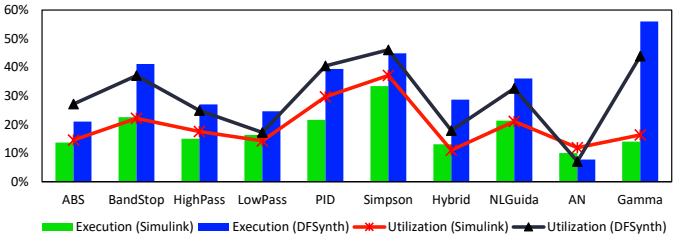


Fig. 9. Correlation between execution improvement and utilization improvement after constant replacement of the code generated by DFSynth.

C. Industrial Case Study

We applied Mercury on a real project from our industry partner to demonstrate the practical capabilities of Mercury. The model shown in Fig. 10 is an automotive temperature control module from an autonomous driving project. We divide the translation sequence obtained in Algorithm 3 into five translation layers to better elaborate the process of code generation. The experimental results in Table V show that Mercury brings 10.2% and 20.5% performance improvements of the generated code over Simulink and DFSynth, respectively. The pipeline slots utilization results of the generated code also confirm the effectiveness of Mercury on the industrial model.

For better illustration, we presented the codes generated by Simulink and Mercury, in Fig. 11. For space limitation, we have simplified the code, and the comments are mapped to the translation layer 1-5 of the actors.

TABLE V
COMPARISON ON INDUSTRY MODEL OF AUTOMOTIVE TEMPERATURE CONTROL

Item	Simulink	DFSynth	Mercury
Execution Time	0.616s	0.696s	**0.553s**
Improvement	+10.2%↑	+20.5%↑	-
Pipeline Utilization	48.8%	46.5%	**54.2%**
Improvement	+11.0%↑	+16.5%↑	-

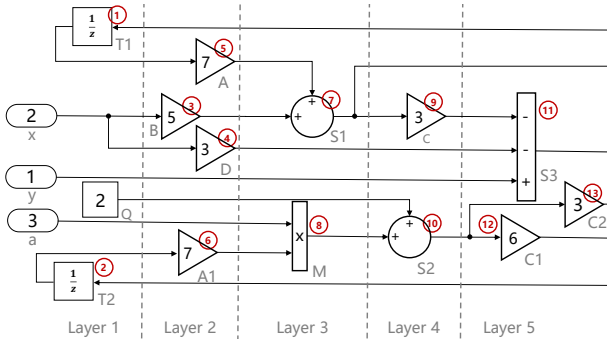


Fig. 10. Part of the model of automotive temperature control from our industrial partner. The obtained translation sequence indicated in red circles is divided into five translation layers.

```
1  k.T1 = k.T1_DSTATE; //......1
2  k.A = 7 * k.T1; //..........2
3  k.B = 5 * k.x; //...........2
4  k.S1 = k.A + k.B; //........3
5  k.C = 3 * k.S1; //..........4
6  k.D = 3 * k.x; //...........2
7  k.S3 = (k.y - k.C) - k.D; //5
8  k.T2 = k.T2_DSTATE; //......1
9  k.A1 = 7 * k.T2; //.........2
10 k.M = k.a * k.A1; //........3
11 k.S2 = k.M + 2; //..........4
12 k.C1 = 6 * k.S2; //.........5
13 //.......................6-11
14 k.C2 = 3 * k.S2; //.........5
15 //...
```

(a) Code generated by Simulink

```
1  k.T1 = k.T1_DSTATE; //......1
2  k.T2 = k.T2_DSTATE; //......1
3  k.B = 5 * k.x; //...........2
4  k.D = 3 * k.x; //...........2
5  k.A = 7 * k.T1; //..........2
6  k.A1 = 7 * k.T2; //.........2
7  k.S1 = k.A + k.B; //........3
8  k.M = k.a * k.A1; //........3
9  k.C = 3 * k.S1; //..........4
10 k.S2 = k.M + 2; //..........4
11 k.S3 = (k.y - k.C) - k.D; //5
12 k.C1 = 6 * k.S2; //.........5
13 k.C2 = 3 * k.S2; //.........5
14 //.......................6-11
15 //...
```

(b) Code generated by Mercury

Fig. 11. The code snippets is based on the model in Fig. 10. The code on the left is generated by Simulink, and the code on the right is generated by Mercury. The comments indicate the translation layer of the actor corresponding to that line of code.

Comparing these two codes, we can see that the calculations in the code generated by Simulink are result-continuous, that is, the results of the current statement are used by the next statement. In contrast, the code from Mercury is generated layer by layer according to the translation layer, which avoids the dependence of the next statement on the results of the previous one as much as possible. For example, in lines 3-5 of the code generated by Simulink on the left side, the variable k.C in line 5 depends on the operation result k.S1 in line 4, and k.S1 in line 4 also depends on the variable k.b in line 3. In these three lines, the operations in lines 4 and 5 are blocked by the operations that precede them, which causes the instruction pipeline to work inefficiently. In particular, the impact is greater when multiplication operations are blocked, because multiplication operations generally require more clock cycles, as shown in Table I. On the other hand, as for lines 3-5

of the code generated by Mercury on the right side, they are three multiplication operations without any data dependencies. So they will not cause the instruction pipeline stall and can be executed in parallel when three and more Arithmetic Logic Units (ALU) exist in the processor.

VI. Discussion

Extensibility of Mercury. Mercury currently supports parsing the representation of Simulink models to generate embedded code. Generally, different model-driven design tools have their own representations for the corresponding models, resulting in difficulties for Mercury to perform code generation. A possible way to solve this problem is to build a well-structured IR to support compatibility with different model-driven design tools. Then, we can apply Mercury to generate pipeline-friendly code for these models without modifications to the approach of generating translation sequence. For example, to support code generation for models of Ptolemy [19], we need to parse "entity", "port", "link" and other relevant Ptolemy's elements into corresponding elements in IR and extract data dependencies from it for further usage. Besides, each model-driven design tool may have unique modeling semantics. To support code generation for models using such modeling semantics, Mercury needs to extend the model actor library for estimating execution latency and support corresponding well-defined DLL files for code synthesis. Furthermore, we support code generation not only for Intel and ARM architectures, but also for optimized code of other architectures, simply by configuring the corresponding processor parameter files for estimating execution latency.

Utilization of Semantics inside Simulink Model. A Simulink model has rich and valuable semantics. Mercury mainly utilizes data dependencies to decrease the occurrence of instruction pipeline stalls to speed up the execution of the generated code. Besides that, there still exist other valuable semantics that can be used to optimize the generated code. Mercury can fully utilize them in two steps. First, parse the related model files and record valuable semantics for optimization. Second, for each deficiency, customize a special optimization approach to solve it. For example, for oversized Simulink models, Mercury can derive mutually insensitive composite actors with data dependencies and branching information. Then Mercury will adopt the OpenMP library to parallelize time-consuming composite actors. In the future, we plan to proactively analyze more valuable modeling semantics for improving the performance of the generated code.

Tradeoffs of our approach. Indeed, generating code in a result-continuous manner like Simulink can reuse register data to save some overhead. However, it is still worthwhile to avoid pipeline stalls due to time-consuming operations. For example, IDIV instruction in the Intel Skylake processor requires more than 40 clocks for execution. This period can be utilized to execute other data-independent instructions. Moreover, due to the mechanisms of cache, the overhead of loading and storing data is not as large as expected. Our experiments also show that Mercury achieves better performance.

Discussion of code generation with least pipeline stalls. In our approach, we adopt a greedy algorithm to decrease the pipeline stalls inside the generated code, i.e, the least penalty priority. Since generating the code with the least pipeline stalls is indeed an NP-hard problem, designing a more sophisticated heuristic by considering more hardware factors may achieve higher performance, e.g., number of ALUs. Our approach supports a simple but effective method to decrease pipeline stalls. It not only achieves higher performance in the generated code, but also does not bring much burden on the code generator. Besides, we did an experiment to investigate and verify the performance of our approach by traversing all possible cases of actor scheduling on a benchmark model (2016 cases in total). The statistics show that although our approach does not reach the global optimal solution, it still exceeds 96% cases while Simulink only exceeds 5% cases.

Correctness of Generated Code. The data dependencies in the Simulink model indicate the direction of data transfer between actors. The mistake order of translation sequence without conforming to the data dependencies will result in inconsistent functionality between the model and generated code. Mercury adopts a topology-based method to obtain the candidate actors to address the above problem. In other words, Mercury iteratively selects an actor without any data dependency for translation and releases data dependencies pointing from the selected actor, thus the data is transferred to the corresponding actors. Therefore, Mercury ensures the correctness of the translation order. Furthermore, we also compare the execution results of the code generated by different tools (i.e. Simulink Embedded Coder, DFSynth, and Mercury), as well as the simulation results of the original Simulink model, and they are all consistent.

VII. Related Work

Model-driven Design. Model-driven design is widely used in the field of embedded control systems [20]. First, it uses a low-code approach to build the system. Then, it uses dynamic simulation to check the correctness of the model. Finally, it generates high quality code by code generation function. There are a lot of existing works focusing on the design of model-driven development tools [2], [21]–[23]. For example, Simulink is one of the most widely used tools in industry [2]. It supports dataflow and stateflow modeling and has good support for code generation of discrete systems. There are also some works that focus on the development of certain domain-specific systems [24], [25]. For example, Tsmart tool is dedicated to model construction, simulation, and hardware code generation for vehicle bus control chips through state machine modeling [26]. A dataflow model usually consists of actors and data connections. Data connections are usually used to link the input and output ports of different actors to represent the flow of data. The function of the actor is to calculate the input data and write the result to the output port accordingly. In model-driven design tools, dataflow models are generally hierarchical, meaning that a model with external input and output ports can be used as a composite actor after encapsulation. This method of encapsulating a complex system into a composite actor is also commonly used to embed state machine models within dataflow models.

Code Generation. Code generation is the most important part of model-driven design [9], [27]. It converts the constructed model into deployable code. For dataflow models, the code generator often translates the actor one by one through calculating the data dependencies of the actors in the model [9]. The complete code is generated by assembling the translated code from all actors and the configurable code such as function headers, global variables, and type definitions [1]. Under the premise that the translation is correct, the efficiency of the code is crucial. Especially in high-speed scheduling task systems, the efficiency of the generated code can affect the performance of the entire control system [10]. Simulink Embedded Coder is a built-in code generator of Simulink [6]. It can perform code optimization in many aspects, such as reusable data exploitation and local variable elimination. In addition, the ability to generate code across platforms and architectures has made Simulink widely popular.

Main Differences. Mercury differs from these work by utilizing the feature of instruction pipeline with code generation. Unlike other code generators, Mercury proactively estimates the execution latency of actors. Based on the execution latency, Mercury can iteratively determine the most suitable actor without data dependencies in generating translation sequence, which decreases the occurrence of instruction pipeline stalls during the execution of the generated code.

VIII. Conclusion

In this paper, we propose Mercury, an instruction pipeline aware code generator for Simulink models. Mercury approximately estimates the execution latency of each actor and adopts the least penalty priority to iteratively select the most suitable actor without data dependencies for generating a translation sequence. Then, Mercury performs code synthesis for each actor to generate instruction pipeline-friendly embedded code based on the obtained translation sequence. We evaluate the effectiveness of Mercury on benchmark and real industrial Simulink models. Compared with state-of-the-art code generators such as Simulink Embedded Coder and DFSynth, Mercury reduces the execution time of the generated code by 9.7%-33.4% and 9.2%-59.4% across different architectures and improves the pipeline slots by 11.0%-37.1% and 10.6%-50.0%, respectively. These results demonstrate that decreasing the occurrence of instruction pipeline stalls can significantly improve the execution efficiency of the generated code.

References

[1] Z. Su, D. Wang, Y. Yang, Y. Jiang, W. Chang, L. Fang, W. Li, and J. Sun, "Code synthesis for dataflow based embedded software design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[2] Simulink and Matlab, *Simulink Documentation*. [Online]. Available: https://www.mathworks.com/help/simulink/index.html

[3] A. A. Babikian, "Automated generation of test scenario models for the system-level safety assurance of autonomous vehicles," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 1–7.

[4] D. Goswami, M. Lukasiewycz, M. Kauer, S. Steinhorst, A. Masrur, S. Chakraborty, and S. Ramesh, "Model-based development and verification of control software for electric vehicles," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2013, pp. 1–9.

[5] P. Pampagnin, P. Moreau, R. Maurice, and D. Guihal, "Model driven hardware design: One step forward to cope with the aerospace industry needs," in *2008 Forum on Specification, Verification and Design Languages*. IEEE, 2008, pp. 179–184.

[6] Simulink and Matlab, *Simulink Documentation*. [Online]. Available: https://www.mathworks.com/solutions/embedded-code-generation.html

[7] H. Hanselmann, U. Kiffmeier, L. Koster, M. Meyer, and A. Rukgauer, "Production quality code generation from Simulink block diagrams," in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*. IEEE, 1999, pp. 213–218.

[8] H. Bourbouh, P.-L. Garoche, T. Loquen, É. Noulard, and C. Pagetti, "CoCoSim, a code generation framework for control/command applications An overview of CoCoSim for multi-periodic discrete Simulink models," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[9] T. Miyazaki and E. A. Lee, "Code generation by using integer-controlled dataflow graph," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. IEEE, 1997, pp. 703–706.

[10] M. Vucha and A. Rajawat, "A case study: Task scheduling methodologies for high speed computing systems," *arXiv preprint arXiv:1501.01370*, 2015.

[11] D. A. Patterson and J. L. Hennessy, "Computer organization and design: the hardware/software interface, (rev. ed. of: Computer organization and design/john l. hennessy, david a. patterson. 1998.)," 2012.

[12] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 272–282, 1989.

[13] M. Dubois, M. Annavaram, and P. Stenström, *Parallel computer organization and design*. cambridge university press, 2012.

[14] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 266–275, 1991.

[15] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[16] P. P. Chaudhuri, *Computer organization and design*. PHI Learning Pvt. Ltd., 2008.

[17] A. Fog, *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[18] K. E. Atkinson, *An introduction to numerical analysis*. John wiley & sons, 2008.

[19] C. Ptolemaeus, *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014, vol. 1.

[20] T. Z. Asici, B. Karaduman, R. Eslampanah, M. Challenger, J. Denil, and H. Vangheluwe, "Applying model driven engineering techniques to the development of contiki-based IoT systems," in *Proceedings of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things*. IEEE Press, 2019, pp. 25–32.

[21] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," in *Readings in Hardware/Software Co-Design*, ser. Systems on Silicon, G. De Micheli, R. Ernst, and W. Wolf, Eds. San Francisco: Morgan Kaufmann, 2002, pp. 527–543.

[22] V. I. GmbH, *DaVinci Developer*. [Online]. Available: https://www.vector.com/us/en-us/products/solutions/autosar-classic/

[23] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[24] Y. Jiang, H. Song, H. Kong, R. Wang, and L. Sha, "Safety-assured model-driven design of the multifunction vehicle bus controller," *IEEE Transactions on Intelligent Transportation Systems*, 2017.

[25] R. Ahmadi, E. Posse, and J. Dingel, "Slicing UML-Based Models of Real-Time Embedded Systems," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 346–356.

[26] Y. Jiang, H. Zhang, H. Zhang, X. Zhao, H. Liu, C. Sun, X. Song, M. Gu, and J. Sun, "Tsmart-galsblock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 711–714.

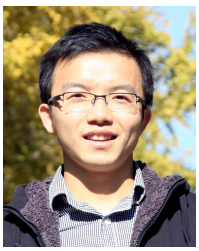
[27] H. Zhang, Y. Jiang, H. Liu, H. Zhang, M. Gu, and J. Sun, "Model driven design of heterogeneous synchronous embedded systems," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 774–779.

Zehong Yu received the B.S. degree in software engineering from Southeast University in 2021. He is pursuing a M.S.E. degree in software engineering at Tsinghua University, Beijing, China. His research interests are in the areas of model driven development and embedded software engineering.

Zhuo Su received the B.S. degree in software engineering from Northeastern University in 2018. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests are in the areas of model driven development and embedded software engineering.

Yixiao Yang received the B.S. degree in software engineering from Nanjing University, Nanjing, China, in 2014. He received the Ph.D. degree in software engineering from Tsinghua University, Beijing, China. He is currently working as a assistant researcher in the College of Information Engineering, Capital Normal University, Beijing, China. His research interests include code completion, test case generation, model driven design and their applications to industry.

Jie Liang received the BS degree in computer science from Beijing University of Posts and Telecommunications in 2017, and the PhD degree in software engineering from Tsinghua University in 2022. He currently works as a Postdoc researcher in School of Software, Tsinghua University. His current research interests include DBMS security and program analysis.



Yu Jiang received the B.S. degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He was a Postdoc researcher with the Department of Computer Science, University of Illinois at Urbana Champaign, Champaign, IL, USA, in 2016, and is now an associate professor in Tsinghua University. His research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

Aiguo Cui is responsible for the software architecture and key technology innovation of Huawei's intelligent vehicle solutions, as well as the innovation and development of Huawei's intelligent vehicle operating system. His main research interests are heterogeneous real-time scheduling, real-time security systems, cyber-physical systems, and intelligent vehicle software architecture.

Wanli Chang received his bachelor's degree (First Class Honours) from Nanyang Technological University, Singapore, in 2008, and the Ph.D. degree in electrical and computer engineering from the Technical University of Munich (TUM), Germany, in 2017, and won the Departmental Best Dissertation Award. He is currently a professor with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research spans across all aspects of real-time and embedded systems.

Rui Wang Rui Wang received the B.S. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2004, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2012. She is currently a professor with the College of Information Engineering, Capital Normal University, Beijing, China. Her research interests include formal verification and their applications in embedded systems.