KNIGHT: Optimizing Code Generation for Simulink Models with Loop Reshaping

Zehong Yu*, Yixiao Yang†, Zhuo Su ✉*, Rui Wang†, Yang Tao‡ and Yu Jiang ✉*
*KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China
†Information Engineering College, Capital Normal University, Beijing 100089, China
‡HUAWEI Technologies, Co. LTD. Shanghai 200120, China

***Abstract*—Simulink has become a pivotal infrastructure in embedded scenarios, including automotive systems and aerospace designs. To improve the performance of the code generated from Simulink models, state-of-the-art code generators employ various optimization techniques, such as expression folding, variable reuse, and parallelism. However, they struggle to generate efficient code for loop-semantic models which are crucial in substantial data processing tasks. This inefficiency manifests in numerous redundant calculations, such as array calculations and conditional statements. As a result, the performance of the generated code is limited.**

This paper proposes KNIGHT, an efficient code generator for loop-semantic Simulink models with loop reshaping. KNIGHT first parses the Simulink model to extract essential content such as block functionalities and connections. KNIGHT then identifies blocks with internal states, and implements the specific interaction rules to discern those that are state-dependent. For state-dependent blocks, KNIGHT conducts forward inference to obtain their preceding blocks, which influence the internal state calculations. Subsequently, KNIGHT isolates blocks that are optimizable and irrelevant to the internal states. These blocks are strategically relocated outside the loop semantics, while preserving critical semantics related to code generation. We implemented and evaluated KNIGHT on benchmark Simulink models across different compilers and architectures. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by KNIGHT is 16.58× faster, 16.89× faster, and 15.38× faster in terms of execution duration on average, without incurring additional overhead of memory usage.

***Index Terms*—Simulink Models, Loop-Semantic, Code Generation**

I. INTRODUCTION

Simulink [1], the most widely used model-driven design tool, serves as the fundamental infrastructure for various embedded scenarios, including automotive systems, aerospace design, and DSP systems [2], [3], [4], [5]. Developers can use the powerful toolkit supported by Simulink, such as simulation, verification, and code generation, to facilitate embedded software development. Among them, code generation is a popular technique for automatically converting the target model into deployable source code, which saves significant labor efforts and receives major adoption in embedded software development. However, the code efficiency should be ensured, as it directly affects the efficiency of the whole system and most embedded devices have limited performance [6].

Simulink supports loop-semantic blocks for modeling that take arrays as input and output, and perform intensive calculations on them, to meet the demands of processing datasets or signal streams. These blocks, such as the `For-Iterator` subsystem, are widely used in both industrial production pipelines and academic benchmarks [7], [8], [9] to effectively manipulate large-scale data. The code efficiency is highly dependent on the comprehension and optimization of loop-semantic blocks within the target model. This is because these blocks are directly related to loop constructs in the generated code, including *for* statement and *while* statement, which are calculation-intensive and time-consuming.

State-of-the-art code generators employ several strategies to ensure code efficiency. For example, Simulink Embedded Coder [10], the built-in tool, supports various optimization options, including expression folding and variable reuse. Typically, those optimization techniques are aimed at generating improved code for individual blocks, without considering the loop semantics. Academic works also have made efforts to improve code efficiency. For example, HCG [11] synthesizes SIMD (Single Instruction Multiple Data) instructions for parallel computation in the model to enhance performance. However, at high-level optimization flags, e.g., `-O3`, compilers also synthesize SIMD instructions within the assembly code, and optimization methods provided by HCG become less effective and may even bring a negative impact.

In fact, the bottleneck of code generation for loop-semantic models arises from redundant calculations within these models, overlooked by the above code generators. The interconnected nature of the blocks means that the execution behavior of one block can significantly impact others, due to the underlying data relationships and modeling semantics. These interactions between model blocks can lead to redundant calculations, which simultaneously occur in the generated code. For instance, an `Assign` block within the `For-Iterator` subsystem, can cause its subsequent blocks to operate on an array of data rather than single elements, thereby leading to extensive redundant calculations. More critically, these redundant calculations, nested within loop semantics, result in repetitive execution and a performance gap. A detailed description of this example will be illustrated in Section III. On the other hand, the optimization techniques employed by state-of-the-art compilers, including GCC and Clang, also fail to effectively eliminate these redundant calculations during the compilation process. Without in-depth knowledge of the target model semantics, such as inport/outport and block functionality, compilers struggle to identify and eliminate invariant variables. Moreover, the intricate data types within the generated code,

such as pointers, pose substantial analytical challenges for compilers. Consequently, these limitations prevent compilers from optimizing the generated code effectively, thereby impinging on the overall performance.

However, it is challenging to effectively eliminate redundant calculations in code generation. There are two main challenges: (1) **The first challenge** is to precisely analyze interactions for discovering redundant calculations. Interactions between blocks encompass not only those that are directly connected but also those indirectly linked. The space of possible interactions is vast, while most interactions are meaningless or equivalent for optimization. Therefore, it is of vital importance to discern the specific characteristics behind interactions that lead to redundant calculations and to provide a general method for identifying them. (2) **The second challenge** is to effectively optimize redundant calculations while maintaining correctness. Since redundant calculations may be distributed across multiple different blocks, it requires a carefully-crafted optimization method to fully eliminate these redundant calculations. Besides, the original model semantics, such as data relationships and block functionalities, should be maintained to guarantee the correctness of the generated code.

To address the aforementioned challenges, we propose KNIGHT, an efficient code generator for loop-semantic Simulink models with loop reshaping. It mainly focuses on fully utilizing model semantics to avoid redundant calculations in code generation. Firstly, KNIGHT parses the Simulink model to obtain essential content as a preparation step, including block functionalities, connections, inports/outports, etc. Then, based on the collected content, KNIGHT identifies blocks with internal states, and then implements the specific interaction rules to determine if they are state-dependent. For state-dependent blocks, KNIGHT conducts forward inference to identify preceding blocks of these state-dependent blocks, which influence the internal state calculations. Subsequently, KNIGHT obtains optimizable blocks irrelevant to the internal states, and then strategically relocates them outside the loop semantics, while preserving critical semantics related to code generation. Finally, KNIGHT performs code synthesis for each block to generate high-efficiency embedded code.

We implement and evaluate the effectiveness of KNIGHT on benchmark Simulink models [12], [13], across different compilers and architectures. The results demonstrate that KNIGHT gains pronounced performance improvement. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by KNIGHT is on average 16.58× faster, 16.89× faster, and 15.38× faster, in terms of execution duration. We also conducted experiments in terms of memory usage. The data shows that KNIGHT improves code performance without incurring additional overhead. In summary, this paper makes the following contributions:

- We identify that state-of-the-art code generators fall short of generating efficient code for loop-semantic Simulink models, leading to numerous redundant calculations.
- We propose KNIGHT, an efficient code generator for loop-semantic models. It first identifies state-dependent blocks within the target model, and then relocates irrelevant blocks outside the loop semantics for generating efficient embedded code.
- We implement KNIGHT and evaluate it on widely-used benchmark Simulink models. The results demonstrate that KNIGHT outperforms the state-of-the-art code generators across different compilers and architectures, without incurring additional overhead.

II. BACKGROUND

A. Model-Driven Design

Model-driven design has emerged as a widely adopted approach in the embedded systems domain, offering a systematic and structured method for designing complex systems[14], [15], [16], [17], [18]. This approach usually encompasses four critical stages: model construction, model simulation, model verification, and code generation[19], [1], [20], [21]. Each stage serves a unique purpose and contributes to the overall efficiency and effectiveness of the model-driven design process. ① Model construction entails creating a comprehensive representation of the system's behavior based on its functional requirements. By constructing a detailed model, developers can visualize the inner workings of the system and facilitate developer understanding. ② Model simulation allows developers to identify and address potential issues early in the development process. ③ Model verification utilizes static analysis techniques to verify whether the model adheres to predefined modeling specifications. ④ Code generation is responsible for transforming the model into executable code suitable for the target embedded system. This stage is crucial because the efficiency and reliability of the generated code affect the performance and success of the embedded system.

B. Simulink and Dataflow Model

Simulink, a part of the MATLAB software suite developed by MathWorks, is a powerful graphical programming environment that provides a block diagram editor and a customizable set of block libraries. Simulink enables users to design, simulate, and analyze dynamic systems, such as control systems, signal processing, and communication systems. It is particularly useful for modeling linear and nonlinear systems, discrete-event systems, and multi-domain systems, which include mechanical, electrical, hydraulic, and thermal components. Simulink also offers advanced tools for analysis, code generation, and other tasks.

The dataflow model is a widely recognized paradigm for modeling and analyzing dynamic systems[22], [23], [24]. It visually represents how data moves and changes within a system, using a graph where each node represents different functions and the lines between them represent the flow of data. Each node processes incoming data based on specific rules or equations, transforming it into output data. These equations or rules are typically expressed using a mathematical or programming language, depending on the system's functionality. Functional elements within the model can be aggregated to create subsystems, which may be further integrated to construct larger, more complex systems. This modular, hierarchical structuring facilitates the development of organized and coherent models, simplifying both analysis and modifications.

C. Code Generation

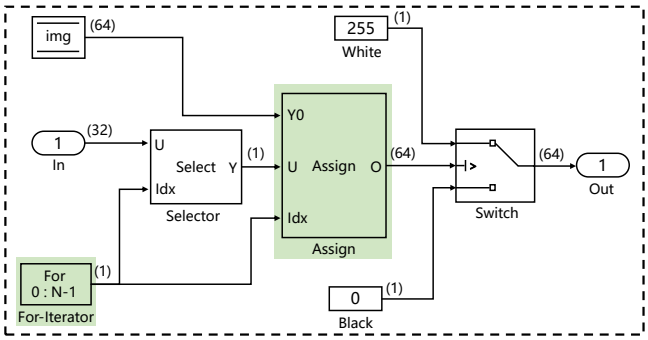
Code generation is of vital importance for model-driven design, which releases the developers from error-prone coding tasks and improves the efficiency of software development. It primarily consists of four essential steps: model parse, dataflow analysis, scheduling, and code synthesis [25], [26], [27].These steps work together seamlessly to convert the user-constructed Simulink models into efficient and deployable code for target hardware platforms. ① First, model parse, as a preparation stage, analyzes and interprets the target model into customized IR (Intermediate Representation) to extract the critical information for further usages, such as the model structure, blocks, and connections. ② Then, dataflow analysis is conducted on the obtained IR. By examining the connections between actors, it derives the sequential relationship and connectivity between blocks. ③ Subsequently, scheduling infers the translation sequence of model blocks, based on the sequential relationship obtained from dataflow analysis. It adopts a topological-based method to iteratively select candidate blocks for translation. ④ Finally, for each block, code synthesis generates corresponding code based on its functionality and other properties, and then assembles them into deployable code according to the translation sequence. As long as the code generators ensure correctness and maintain the original model semantics, they can customize their own implementation for each block to enhance performance.

III. Motivation

State-of-the-art code generators, such as Simulink Embedded Coder [10] and HCG [11], employ various optimization techniques to ensure the efficiency of the generated code, including expression folding, variable reuse, and instruction parallelism. These approaches have demonstrated impressive results in numerous scenarios, and the generated code has been deployed in many critical embedded systems [28], [29]. However, they still fall short of generating efficient code for loop-semantic Simulink models. Specifically, the code generated by these tools contains numerous redundant statements, such as repetitive initialization, leading to unsatisfactory performance. Moreover, such redundant statements bring challenges for compilers to optimize. For instance, the complex conditional statements hinder the potential for loop unrolling and SIMD instruction utilization. Consequently, code generators must address these issues effectively.

A. Illustrative Example

Figure 1 illustrates an example to demonstrate the severity of the aforementioned problems. Figure 1a shows the internal structure of a `For-Iterator` subsystem, which serves as a sample binarization part [30] for the image processing application. This model converts a grayscale image into a binary image (black and white) by setting a threshold value. Pixels with intensity values greater than the threshold are set to 255 (white), while those below the threshold are set to 0 (black). Specifically, it first uses the pixels restored in the `img` block as initialization. Then, `Selector` block iteratively reads pixels from `In` block and sends the obtained pixels to `Assign` block. `For-Iterator` block is the index block



(a) Motivation Model

```
1 void binarize(uint8_t In[32],
2    uint8_t Out[64]) {
3   int i;
4   int j;
5   uint8_t Assign[64];
6   memcpy(&Assign[0], &img[0],
7      sizeof(uint8_T) << 6U);
8   for (i = 0; i < 32; i++) {
9     Assign[i] = In[i];
10    for (j = 0; j < 64; j++) {
11      Switch[j] = Assign[j] >
12          threshold ? 255U : 0U;
13    }
14  }
15
16
17
18
19  memcpy(&Out[0], &Switch[0],
20    sizeof(uint8_t) << 6U);
21 }
```

(b) Original Code

```
1 void binarize(uint8_t In[32],
2    uint8_t Out[64]) {
3   int i;
4   int j;
5   uint8_t Assign[64];
6   memcpy(&Assign[0], &img[0],
7      sizeof(uint8_T) << 6U);
8   for (i = 0; i < 32; i++) {
9     Assign[i] = In[i];
10    for (j = 0; j < 64; j++) {
11      Switch[j] = Assign[j] >
12        Threshold ? 255U : 0U;
13    }
14  }
15  for (i = 0; i < 64; i++) {
16    Switch[i] = Assign[i] >
17        threshold ? 255U : 0U;
18  }
19  memcpy(&Out[0], &Switch[0],
20    sizeof(uint8_t) << 6U);
21 }
```

(c) Optimized Code

Fig. 1: A Simulink model to illustrate the motivation of our approach. It represents the binarization part of an image processing application, which converts a grayscale image into a binary image (black and white). The data length emitted from each outport is denoted by the numbers above the connections. The left code is generated by Simulink Embedded Coder, while the right code is optimized by loop reshaping. The differences lead to the time complexity from $O(n)$ to $O(n^2)$.

within the `For-Iterator` subsystem, which determines the pixels read in each iteration. Finally, the obtained pixels are sent to the `Switch` block to determine whether they exceed the threshold value. If the threshold is exceeded, 255 is assigned to the `Out` block; otherwise, 0 is assigned to it. Note that, the data object transferred and manipulated by `Assign` block and its subsequent blocks is a 64-length array since both the `img` block and `Out` block handle 64-length arrays.

By comparing the code generated by Simulink Embedded Coder (Figure 1b) with the optimized code (Figure 1c), we found that the root cause of the code inefficiency is the redundant *for* statement (line 10-13 in Figure 1b). Since the data object of `Switch` block is a 64-length array, Simulink Embedded Coder should generate *for* statement to represent this semantic. However, the presence of the `For-Iterator` and `Assign` blocks indicates that the execution of the `Switch` block is required only after `Assign` block's result is finalized. By considering this interaction, the aforementioned *for* statement can be relocated outside the outer *for* statement (line 15-18 in Figure 1c), reducing time complexity of the entire function from $O(n^2)$ to $O(n)$.

To quantitatively understand the severity of the aforementioned issues, we compile both the code generated by Simulink

Embedded Coder and its optimized counterpart across C-Compilers, i.e., GCC and Clang. Subsequently, the compiled versions with different optimization levels are deployed on an experimental machine (Win11, AMD Ryzen 7 5800X, 32GB memory) to collect execution duration statistics. The statistics as shown in Table I reveal that there exists a significant performance gap between the original code and the optimized code among various optimization levels employed by compilers.

Moreover, the results show that the carefully-crafted optimizations implemented by compilers are rendered ineffective in addressing this issue without high-level knowledge of model semantics. Some optimization techniques employed by compilers may even exacerbate this problem. For instance, the original code compiled with GCC takes a longer execution time at the `-O2` flag compared to the `-O1` flag. Besides, although the original code compiled with Clang employing `-O3` flag achieves a satisfactory result, taking 0.163s for execution, there remains a more than 10× performance gap compared to the optimized code compiled under the same settings. In other words, the time complexity issue (line 8-14 in Figure 1b) remains unresolved even with the highest optimization level of Clang. More importantly, considering the constrained performance capabilities of embedded devices, such a pronounced performance gap is deemed unacceptable [31]. In summary, these issues emphasize the importance of code generation for loop-semantic models and the urgent need for an effective approach to enhance performance.

TABLE I: Comparison of execution duration between the original code generated by Simulink Embedded Coder and optimized code with loop reshaping across different compilers and optimization levels.

Optimization Level		Original Code	Optimized Code
GCC	`O0`	10.950s	0.468s
	`O1`	4.558s	0.138s
	`O2`	5.386s	0.129s
	`O3`	2.094s	0.014s
Clang	`O0`	13.235s	0.524s
	`O1`	4.545s	0.138s
	`O2`	0.173s	0.014s
	`O3`	0.163s	0.014s

B. Observation

The root cause of the code inefficiency, as illustrated in Figure 1, is that code generators generate code for each block in isolation, without carefully considering their interactions, thereby leading to a marked escalation in execution duration.

We observed that the redundant *for* statements are present in blocks following the `Assign` block, such as `Switch` block. With the in-depth analysis of `Assign` block's functionality and its interactions with other blocks, we identified two key characteristics behind `Assign` block: (1) **Presence of Internal State.** The internal state of `Assign` block saves the execution results and is utilized for subsequent execution, which represents a pre-defined variable in the code level. During the iterative execution of the model, this internal state is dynamically updated in response to the outcomes of each execution. (2) **Dependency on Previous States.** During each iteration, `Assign` block utilizes the value obtained from "U" port to update the index element specified by "Idx" port, i.e., Assign[Idx] = U. Due to the presence of `For-Iterator` block, this process is a continuous accumulation. For instance, line 8-14 in Figure 1b demonstrate this. When the loop reaches its 32nd iteration, i.e., iter = 31, the values assigned in previous iterations continue to be preserved in the Assign array, without being overwritten. We define blocks that exhibit both of these characteristics as **State-Dependent Blocks**. Note that whether a block contains an internal state is dictated by its specific functionality. In Simulink, numerous blocks, such as `Delay` block, `Data Memory` block, and `Accumulator` block, contain internal states. The dependence of a block's internal state on its previous states, however, is determined by its interactions with other blocks.

The characteristics mentioned above inspire us to understand the code inefficiency from the perspective of model semantics, rather than the code level. Specifically, since the execution result of the `Assign` block depends on its previous state, the preceding blocks connected to the `Assign` block must be situated within the scope of the outer *for* statement. This is because the previous and current states of the `Assign` block are ascertained from the execution outcomes of these preceding blocks. On the other hand, for subsequent blocks that are directly or indirectly connected to the `Assign` block, if they do not contain internal states and are not preceding blocks of other state-dependent blocks, then their execution results are solely determined by the final state of the `Assign` block. Therefore, such blocks can be relocated outside the outer *for* statement.

In summary, our observation reveals that redundant calculations within loop-semantic models are highly related to state-dependent blocks, while existing code generators do not take them into account. This motivates us to design KNIGHT, an efficient code generator for loop-semantic models. KNIGHT precisely analyses the interactions between blocks to identify state-dependent blocks, and then effectively optimizes the redundant calculations caused by state-dependent blocks, thereby improving overall performance.

IV. DESIGN

In this section, we detail the design of KNIGHT, which generates efficient code for loop-semantic models. Figure 2 shows the overall framework of KNIGHT. It mainly contains the following two key components. (1) **Interaction-Oriented Analysis:** KNIGHT first parses the Simulink model to obtain essential information as the preparation step, including block functionalities, connections, inports/outports, etc. Then, KNIGHT analyses the functionality and semantics of each block to determine whether it contains internal states. For those blocks containing internal states, KNIGHT implements corresponding interaction rules to analyze their interactions with other blocks, subsequently determining if these components are state-dependent. (2) **Loop Reshaping:** For state-dependent blocks, KNIGHT, based on connection relationships, performs forward inference, identifying blocks that are either directly or indirectly connected to state-dependent blocks, as these are related to the calculation of internal states. Conversely, for other blocks that are optimizable but irrelevant to the

internal state, KNIGHT strategically relocates them outside the loop semantics, while preserving the semantics related to code generation to ensure correctness. Finally, KNIGHT performs code synthesis for each block.

A. Important Definitions

Before introducing the design of KNIGHT in detail, we first define important concepts used in this paper as follows:

Definition 4.1 (Dataflow Graph): Dataflow graph G is denoted as a tuple $\{B, I, O, C\}$. $B = \{b_1, b_2, ..., b_n\}$ is the set of blocks of G. $I = \{i_1, i_2, ..., i_n\}$ and $O = \{o_1, o_2, ..., o_n\}$ represent the set of inports and outports of G. Inports accept the data required by the target block, where outports output the result of the block's execution. $C = \{c_1, c_2, ..., c_n\}$ is the set of connections within G, where each $c_i \in O \times I$ represents a connection from an output o_i (data source) to an inport i_i (data destination). Notably, each block b_i can either be a basic block, denoted as $\dot{b}_i$, or a subsystem, denoted as $\ddot{b}_i$.

Definition 4.2 (Basic Block): Basic block $\dot{b}$ is defined as a tuple $\{f, I, O, \Sigma\}$. $f : V_I \rightarrow V_O$ represents the functionality of $\dot{b}$, which takes values V_I from inports I to calculate the values V_O at outports O. I and O represent the set of inports and outports of $\dot{b}$. $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$ represents the set of internal states of $\dot{b}$. Specifically, some blocks preserve the current execution results for other usages. We refer to variables that save these results as internal states Σ. If a basic block $\dot{b}$ has no internal states, then $\Sigma = \varnothing$.

Definition 4.3 (Composite Block): Composite block $\ddot{b}$ is defined as a tuple $\{f, B, I, O, C, \Sigma\}$. f represents the functionality of $\ddot{b}$. B represents the set of blocks within $\ddot{b}$. I and O represent the set of inports and outports of $\ddot{b}$. C represents connections within $\ddot{b}$. Σ represents the internal states of $\ddot{b}$; Note that, functionality f of composite block $\ddot{b}$ is synthesized by the functionalities $\{f_{b_1}, f_{b_2}, ..., f_{b_n}\}$ of the contained blocks B, i.e., $f = f_{b_1} \oplus f_{b_2} \oplus \cdots \oplus f_{b_n}$. Internal states Σ of composite block $\ddot{b}$ is the union of the internal states $\{\Sigma_{b_1}, \Sigma_{b_2}, ..., \Sigma_{b_n}\}$ of the contained blocks B, i.e., $\Sigma = \Sigma_{b_1} \cup \Sigma_{b_2} \cup \cdots \cup \Sigma_{b_n}$.

Definition 4.4 (Loop-Semantic Subsystem): Loop-semantic subsystem s is a specialized type of composite block $\ddot{b}$, denoted as a tuple $\{f, \bar{b}, B, I, O, C, \Sigma\}$. Among them, f represents the functionality of s. $\bar{b}$ is the index block of s, which records the total number of iterations required by the loop-semantic subsystem s and tracks the current iteration round. During each iteration, the count maintained by $\bar{b}$ is incremented by one, and it is checked against the total number of iterations to determine if further iterations are needed. B is the set of blocks within s except index block $\bar{b}$. I and O represents the set of inports and outports of s, respectively. C includes all internal connections within the subsystem s. Σ represents the set of internal states of $\bar{b}$. Specifically, in each iteration, s first activates the index block $\bar{b}$ to determine whether execution should continue. If the condition is met, s retrieves the data from the inports I, passes this data to contained blocks B, and obtains the execution results, which are then sent to the outports O.

Definition 4.5 (State-Dependent Block): State-dependent block $\hat{b}$ is denoted as a tuple $\{f, I, O, \Sigma\}$. $f : V_I \times \Sigma \rightarrow V_O$ represents the functionality of $\hat{b}$, which takes values V_I from inports I and internal states Σ to calculate the values V_O at outports O. I and O represent the set of inports and outports of $\hat{b}$. Σ represents the set of internal states of $\hat{b}$. The state-dependent block $\hat{b}$ must satisfy the following conditions simultaneously: (1) It contains internal states, i.e., $\Sigma \neq \varnothing$. (2) Its current state Σ_c depends on both the current values of inports $V_c = \{v_{i_0}, v_{i_1}, ..., v_{i_n}\}$ and the previous states $\Sigma_p = \{\sigma_0^p, \sigma_1^p, ..., \sigma_n^p\}$, i.e., $\Sigma_c \leftarrow f(V_c \times \Sigma_p)$.

Due to the interactions between blocks, the current values of the internal states often relies on its previous state. We will introduce these interactions and how to determine state-dependent blocks in Section IV-B in detail.

B. Interaction-Oriented Analysis

Interaction-oriented analysis is to identify state-dependent blocks $\hat{B}$ for optimization. In this way, KNIGHT can obtain blocks irrelevant to internal states and relocate them outside the loop semantics, thereby avoiding redundant calculations.

Model Parse. KNIGHT first parses the given model to extract the essential information. Specifically, the Simulink model is a `ZIP` file that contains several crucial components, including model structure, parameters, configuration, and other properties. These components are recorded in the corresponding `XML` files. KNIGHT systematically parses these files from the given model to construct dataflow graph G and gather crucial information, including block functionalities, connections, and inports/outports.

Internal State Perception. During this stage, KNIGHT performs dataflow analysis examining each block in the model. It focuses on identifying if each block has internal states by analyzing their specific functionalities. Blocks with different types have distinct functionalities. Through a detailed analysis of the block library supported by Simulink, we identified that certain blocks contain internal states, and these blocks can be divided into the following types.

- Memory type. This type of blocks requests and initializes a memory space for data storage, and supports read and write operations on this memory, such as `Data Store` block, `File` block, etc.
- Delay type. This type of blocks is designed to output the acquired data after a specific delay period. Simulink supports various delay type blocks, which serve different purposes, such as `Unit Delay` block, `Tapped Delay` block, `Transport Delay` block, etc.
- Discrete-Calculation type. These blocks save the input values distributed over discrete time and perform specific operations on these values, such as `Accumulator` block, `Integrator` block, etc.
- Complex type. These blocks are used in complex modeling scenarios and utilize the internal state to implement the corresponding functionalities, such as `Assign` block, `Bus Assign` block, `Difference` block, etc.

Interaction Rules. For the dataflow model, the connections define data dependencies between blocks, which similarly exist among blocks that are indirectly connected. These dependencies, called interactions, are crucial for identifying blocks that are state-dependent. However, due to the extensive number of

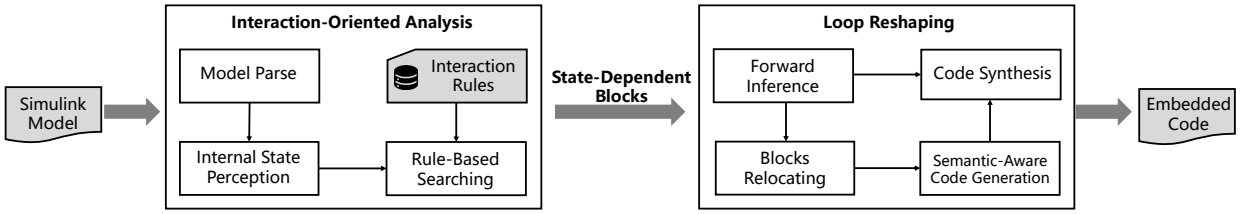


Fig. 2: An overview of KNIGHT. It mainly contains two key components. (1) Interaction-Oriented Analysis: KNIGHT first parses the target model to extract critical contents for preparation. Then, KNIGHT identifies blocks with internal states in accordance with their functionalities, and implements interaction rules to determine state-dependent blocks. (2) Loop Reshaping: For state-dependent blocks, KNIGHT conducts forward inference to identify their preceding blocks. Then, KNIGHT obtains optimizable blocks irrelevant to the internal states, and then strategically relocates them outside the loop semantics, while preserving critical semantics related to code generation. Finally, KNIGHT performs code synthesis for each block.

blocks and the complexity of the connections, the space of possible interactions is vast, while most interactions are meaningless or equivalent for analysis. KNIGHT makes the following efforts to figure out this problem. First, KNIGHT mainly concentrates on interactions that are relevant to blocks that contain internal states. Through dataflow analysis, KNIGHT identifies the preceding blocks connected to these blocks with internal states. For example, consider the `Assign` block in Figure 1a. By analyzing connections related to inports of `Assign` block, KNIGHT can determine four preceding blocks to `Assign` block, i.e., `img` block, `In` block, `For-Iterator` block, and `Selector` block. These blocks determine the data source of `Assign` block.

Second, KNIGHT implements specific interaction rules for analyzing blocks with internal states, focusing on their interactions to determine if they are state-dependent. More concretely, given a block b contains an internal state σ, the carefully-crafted rule $\mathcal{R}(b, G) \rightarrow \{true, false\}$ is utilized to analyze the related interactions extracted from the dataflow graph G, using the following constructs. If $\mathcal{R}(b, G) = true$, this implies that the target block b is state-dependent; otherwise, it is not considered state-dependent.

For each type of block with internal states, the interaction rules are as follows: ① For memory type blocks, KNIGHT determines whether read and write operations on the memory data occur simultaneously within the same loop-semantic subsystem s. If so, the memory data changes across the iterations, indicating that the block is state-dependent. ② For delay type blocks, KNIGHT determines whether these blocks cause circular dependency in the model. If so, it indicates that the values assigned to the delay block will be used in subsequent iterations for calculation, thereby inferring that the block is state-dependent. ③ For discrete-calculation blocks, the functionality of these blocks inherently determines that they are state-dependent, that is, they save values distributed over discrete time and perform calculations on these values. ④ For complex blocks, KNIGHT should design interaction rules in accordance with their functionalities and modeling semantics. Take the `Assign` block in Figure 1a as an example. KNIGHT evaluates if the value of the "idx" port is a variant value. This port determines which specific elements the Assign block assigns values to. Therefore, if this port obtains a variant, then the elements assigned in each iteration are different,

Algorithm 1: Rule-Based Searching

```
Input: G: Dataflow graph of target model
       R : Interaction rules
Output: B̂: State-Dependent blocks
1  Function RuleBasedSearching(G, R):
2      B̂ ← ∅
3      // traverse blocks within the graph
4      for b in G do
5          dependent ← false
6          // determine if the block b contains internal state σ
7          if b contains σ then
8              t ← b.type
9              // interaction rules
10             if t is memory then
11                 dependent ← R.memory(b, G)
12             else if t is delay then
13                 dependent ← R.delay(b, G)
14             else if t is discrete calculation then
15                 dependent ← R.discreteCalculation(b, G)
16             else if t is complex then
17                 dependent ← R.complex(b, G)
18             if dependent = true then
19                 B̂.append(b)
20     return B̂
21 End Function
```

signifying state-dependent. `For-Iterator` block represents the iteration rounds, which is a variant value and connects to the "idx" port. Consequently, the `Assign` block updates different elements in each iteration based on the previous ones, confirming its state-dependent nature. Besides, when a block contains multiple internal states, KNIGHT determines whether each internal state is state-dependent. If at least one internal state is state-dependent, then it is a state-dependent block. For the subsystem within the model, whether they are state-dependent depends on the inner blocks. That is, if there exist state-dependent blocks, then it is state-dependent.

Rule-Based Searching. Leveraging the interaction rules, KNIGHT employs a search method to traverse the dataflow graph G, effectively identifying state-dependent blocks $\hat{B}$. The process of rule-based searching is detailed in Algorithm 1. First, KNIGHT traverses the dataflow graph to identify blocks that contain internal states (line 7). Specifically, KNIGHT determines whether the type of the block b meets the requirements defined by internal state perception. If so, it indicates

that b contains the internal state σ, allowing for further operations. Then, KNIGHT utilizes interaction rules $\mathcal{R}$ and dataflow graph G to determine whether b is state-dependent (line 10-17). Note that, KNIGHT only considers the preceding blocks of b, as these blocks can affect the input data of b. In other words, only the interactions between them can affect the internal state σ of b. Moreover, KNIGHT selects the corresponding rules in accordance with the type of b to determine whether it is state-dependent. Finally, KNIGHT appends the identified state-dependent blocks into $\hat{B}$ for further optimization.

C. Loop Reshaping

To eliminate redundant calculations within loop-semantic models, KNIGHT proposes an effective technique, called **Loop Reshaping**, to ensure the efficiency of the generated code. The key idea is to identify blocks associated with internal state calculations by analyzing the connections of state-dependent blocks. Subsequently, KNIGHT can discern blocks irrelevant to internal state calculations and relocate them outside the loop semantics, thereby enhancing the overall performance.

Specifically, KNIGHT classifies the blocks within the target model into three distinct categories based on the connections of state-dependent blocks for thorough analysis. ① **Preceding Blocks B_p.** These blocks are connected, either directly or indirectly, to the inports of state-dependent blocks $\hat{B}$, thereby providing essential data for their execution. The current and previous states of these state-dependent blocks $\hat{B}$ rely on these preceding blocks B_p. As a result, the calculation of these preceding blocks B_p should be maintained within loop semantics. ② **Subsequent Blocks B_s.** These blocks have connections, either directly or indirectly, to the outports of the state-dependent blocks $\hat{B}$, from which they obtain the data needed for their execution. As they are not state-dependent, their execution results are solely determined by the final state of state-dependent blocks $\hat{B}$. Therefore, subsequent blocks B_s can be effectively relocated outside the loop semantics for execution. ③ **Irrelevant Blocks B_i.** These blocks have no direct or indirect connection to the inports and outports of the state-dependent blocks $\hat{B}$. They repeat the same calculations within loop semantics, and therefore can be relocated, similar to subsequent blocks B_s.

Forward Inference. The goal of forward inference is to identify the preceding blocks B_p of state-dependent blocks $\hat{B}$. Algorithm 2 presents the overall procedure. First, KNIGHT utilizes a $stack$ to preserve state-dependent blocks $\hat{B}$ obtained previously, and initializes the preceding blocks B_p as a `NULL` set (line 2-3). Then, KNIGHT iteratively pops the front element of $stack$ to identify its preceding blocks until it is empty (line 4-6). Specifically, for each obtained block b, KNIGHT determines whether b falls within the loop semantics, as blocks outside the loop semantics do not need to be relocated (line 7). In other words, KNIGHT determines if b is within a loop-semantic subsystem s, i.e., if there exists a loop-semantic subsystem s, such that $b \in B_s$. If so, KNIGHT performs dataflow analysis on each inport i of b to identify the data source block $source$ (line 9-10). This can be achieved by analyzing the corresponding connection in the model, as a connection defines both the data source block and the data destination block. For instance, consider the model shown in Figure 1a. By analyzing the connection related to the "U" port of the `Assign` block, KNIGHT can determine the data source block is `Selector` block. In this way, KNIGHT can identify all the preceding blocks of b, and appends them into B_p.

Algorithm 2: Forward Inference

```
Input: G: Dataflow graph of target model
       B̂ : State-dependent blocks
Output: B_p: Preceding blocks
1  Function ForwardInference(G, B̂):
2      B_p ← ∅
3      stack ← B̂
4      while stack ≠ ∅ do
5          b ← stack.top()
6          stack.pop()
7          if b is in loop semantic then
8              // traverse each inport of block b
9              for i ∈ b.I do
10                 source ← G.findSourceBlock(i)
11                 stack.push(source)
12                 B_p.append(source)
13     return B_p
14 End Function
```

Blocks Relocating. With the preceding blocks identified, KNIGHT can easily obtain the subsequent blocks and irrelevant blocks, and then relocate them outside the loop semantic. First, KNIGHT employs a topological-based method to obtain the translation sequence for the target model. It defines the translation order of each block in code generation. In this way, KNIGHT is able to identify the first block after the end position of the loop semantic. After that, KNIGHT relocates both the subsequent blocks and irrelevant blocks outside the loop semantic. Specifically, KNIGHT moves the position of these blocks in the translation sequence before the identified first block. Note that, in the process, KNIGHT only changes the translation order of these blocks without modifying any other critical contents, especially the data relationship.

We take the example shown in Figure 3 to illustrate the process of blocks relocating in detail. Each block inside the model is indicated by a circular marker, while the `For-Iterator` subsystem (loop semantic) is surrounded by dashed lines. First, KNIGHT identifies the state-dependent block (`Assign` block with **orange** marker) by the forward inference. Then, KNIGHT obtains the translation sequence of the model by the topological method, i.e., block ① to block ⑪, and then identifies the first block after the end position of the loop semantic, i.e., block ⑪. It should be emphasized that the lines in the translation sequence only represent the order of block translation and do not correspond to the connections in the dataflow graph. Finally, KNIGHT relocates the subsequent blocks and irrelevant blocks (blocks with **green** marker) outside the loop semantic (before block ⑪).

Semantic-Aware Code Generation. For the relocated blocks, it is essential to maintain the critical modeling semantics, thereby ensuring the correctness of the generated code. First, KNIGHT should maintain the connections within the dataflow graph for relocated blocks. These connections are essential as they determine the data sources for blocks. For

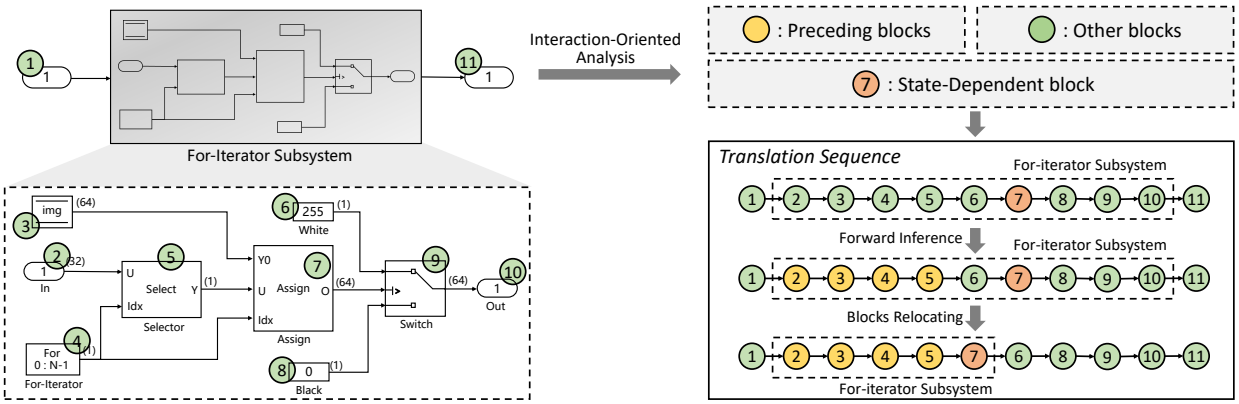


Fig. 3: An example illustrates the process of blocks relocating in detail. Each block is indicated by a circular marker, while the `For-Iterator` subsystem is surrounded by dashed lines. The translation sequences represent the translation order of each block during code generation.

example, in Figure 3, the connection between `Assign` block and `Switch` block defines the data source of the `Switch` block, and is preserved after relocating. Then, the translation sequence between relocated blocks should be maintained. This ensures that the relocated block is executed only after obtaining the required data. For instance, in Figure 3, after relocating, the translation sequence between block ⑥, block ⑧, and block ⑨ is constant. Finally, the specific functionalities of each relocated block are recorded, and the corresponding code will be generated in accordance with these functionalities during code generation.

Code Synthesis. In KNIGHT, the process of generating code for blocks within the dataflow graph is similar to Simulink Embedded Coder and DFSynth. Firstly, KNIGHT employs a topology-based approach to obtain the translation sequence of the target model. For each block, KNIGHT defines the corresponding DLL (Dynamic Link Library) to generate the appropriate code. Note that, blocks of the same type can exhibit varying details, resulting in differences in the generated code. For example, the code generated for `Add` block varies depending on the input type (e.g., float versus double). Therefore, KNIGHT adjusts essential parameters within the DLL files to capture the specific code requirements. Subsequently, the generated code of each block is synthesized together in accordance with the translation sequence.

V. IMPLEMENTATION

We have developed KNIGHT[1] in C++, consisting of 26,142 lines of code. KNIGHT supports code generation for a wide range of blocks, such as the math operation blocks, matrix operation blocks, delay blocks, and complex blocks, which are frequently used in various embedded scenarios. For each block, we have crafted corresponding DLL files for code generation in accordance with the data type, inputs/outputs, and functionalities of the target actor. Besides, for blocks with internal states, we have designed the specific interaction rules to determine if they are state-dependents. These rules are maintained in external files to support cross-architectures.

[1]The implementation, benchmark models, and the results are represented at the repository (https://github.com/YzhDDDing/Knight).

VI. EVALUATION

Experiments Setup: We evaluated the performance of KNIGHT on a benchmark of 8 commonly used loop-semantic Simulink models collected from both academia and industry [32], [13], as shown in Table II. These models contain 243 blocks and 28 subsystems on average, and are employed in real-world embedded scenarios. To investigate the effectiveness of our approach, we compared KNIGHT with three state-of-the-art code generators, the official Simulink Embedded Coder [10], DFSynth [12], and HCG [11]. The comparison experiments were conducted across different compilers and architectures to validate the practicality of KNIGHT. Besides, we collected other important indicators among these code generators to measure the overhead of KNIGHT, including code length and memory usage. Since Simulink Embedded Coder is a built-in tool for Simulink, we use Simulink as the abbreviation in the following experimental content.

TABLE II: Description of benchmark models.

Model	Description	#Subsystem	#Block
CPUTask	AutoSAR CPU task dispatch system	27	215
Diffsion	Thermal diffusion simulation model	6	44
FMTM	Factory Multi-point Temperature Monitor	42	301
HighPass	HighPass filter model	9	63
Hybrid	Hybrid filter model	17	114
LANSwitch	LAN Switch controller	39	409
LEDLC	LED matrix load control	29	232
RAC	Robotic arm controller	57	565

A. Evaluation on x86 Platform

To demonstrate the performance of the generated code by KNIGHT, we conducted the experiment on the benchmark models. The experiment environment was an industrial machine (Win11, AMD Ryzen 7 5800X, 32GB memory). The generated code was compiled by C-Compilers, i.e., GCC (v11.3.0) and Clang (v14.0.6), employing various optimization levels including `-O2`, and `-O3`. Besides, to eliminate statistical errors, each of the generated code is repeatedly executed 5,000,000 times to obtain an average result.

Comparison experiment using `-O0` flag. Developers tend to use `-O0` as the compilation flag for code traceability

in safety-critical embedded scenarios. Consequently, we first conducted the comparison experiment using this flag. Table III shows the experiment results under `-O0` compilation flag. For comparison experiments compiled with GCC, the execution duration of KNIGHT is 21.32× faster than Simulink, 19.40× faster than DFSynth, and 18.16× faster than HCG on average. As for compiling using Clang, the execution duration of KNIGHT is 18.34× faster than Simulink, 17.78× faster than DFSynth, and 16.20× faster than HCG on average. The statistics indicate that KNIGHT achieves significant performance improvements compared to other code generators.

Simulink outperforms DFSynth and HCG on some models, such as `Diffsion` model, but still not as good as KNIGHT. Its optimization technique for expression folding is very powerful, enabling the reuse of block outports and thereby minimizing many intermediate variables. Meanwhile, Simulink generates code for subsystems using an inline approach, while KNIGHT, DFSynth, and HCG generate separate functions for these subsystems. This approach somewhat reduces the overhead of function calls but decreases the reusability and readability of the generated code. The performance of the generated code by DFSynth is relatively limited, mainly because they lack effective optimization techniques for data-intensive models. It is mainly because DFSynth mainly focuses on generating concise code for complex branch blocks inside the model, thus lacking effective optimization techniques for loop-semantic models. As for HCG, it synthesizes appropriate SIMD instructions for computing blocks and achieves performance improvement on part of models, for example, the `LANSwitch` model. However, for other models, HCG does not perform very well, falling behind other code generators. This is because most benchmark models contain decision-related blocks, for example, `IF` subsystem, and HCG is unable to effectively synthesize appropriate SIMD instructions for such blocks. Consequently, the code generated by HCG often exhibits a combination of SIMD and regular instructions. This leads to frequent data exchanges between memory and vector registers, ultimately resulting in unsatisfactory performance.

TABLE III: Comparison of the code execution duration on x86 with `-O0` flag.

Model	Compiler	Simulink	DFSynth	HCG	KNIGHT
CPUTask	GCC	6.28s	26.81s	27.01s	**3.03s**
	Clang	7.95s	31.12s	30.39s	**3.45s**
Diffsion	GCC	6.26s	34.07s	28.41s	**2.98s**
	Clang	5.72s	37.77s	31.33s	**3.24s**
FMTM	GCC	39.38s	11.41s	11.34s	**0.57s**
	Clang	40.54s	11.48s	11.61s	**0.71s**
HighPass	GCC	2.10s	7.41s	5.49s	**0.58s**
	Clang	2.20s	8.18s	5.86s	**0.57s**
Hybrid	GCC	1.07s	3.77s	3.77s	**0.31s**
	Clang	1.14s	4.13s	4.10s	**0.29s**
LANSwitch	GCC	43.79s	34.97s	31.30s	**0.81s**
	Clang	45.16s	36.12s	29.44s	**1.00s**
LEDLC	GCC	21.54s	88.57s	90.16s	**3.66s**
	Clang	22.99s	92.43s	94.24s	**4.37s**
RAC	GCC	39.02s	29.12s	28.39s	**1.29s**
	Clang	41.51s	29.77s	29.43s	**1.53s**

Compared to other code generators, KNIGHT proactively leverages critical information within the model, e.g., block functionalities and connections. In this way, KNIGHT can analyze interactions between blocks to identify state-dependent blocks. This process reveals a substantial amount of redundant calculations, and KNIGHT can relocate them outside the loop semantic for performance improvement. Other code generators, even including state-of-the-art compilers, are unable to achieve this. Therefore, KNIGHT outperforms other code generators. Besides, we found that KNIGHT achieves excellent results on some benchmark models. For example, for the `FMTM` model, KNIGHT is 60.20× faster than Simulink, 20.05× than DFSynth, and 19.93× than HCG, respectively. Through detailed analysis of this model, we found that, compared to other models, it contains more calculation-related blocks rather than decision-related blocks. In other words, redundant calculations have a greater impact on this model, allowing KNIGHT to achieve significant improvement. Furthermore, for all models under `-O0` compilation flag, the code compiled by GCC performs better than the code compiled by Clang.

Comparison experiment using `-O2` flag. Due to stability and reliability, the majority of embedded software is compiled with `-O2` compilation flag. Consequently, we also conducted a comparison experiment using this flag. Table IV shows the experiment results under `-O2` compilation flag. For comparison experiments compiled with GCC, the execution duration of KNIGHT is 12.60× faster than Simulink, 23.81× faster than DFSynth, and 16.68× faster than HCG on average. As for compiling using Clang, the execution duration of KNIGHT is 11.66× faster than Simulink, 11.35× faster than DFSynth, and 12.60× faster than HCG on average.

TABLE IV: Comparison of the code execution duration on x86 with `-O2` flag.

Model	Compiler	Simulink	DFSynth	HCG	KNIGHT
CPUTask	GCC	1.41s	5.18s	4.93s	**0.61s**
	Clang	1.13s	1.44s	1.44s	**0.31s**
Diffsion	GCC	1.97s	11.16s	4.00s	**0.50s**
	Clang	0.84s	2.10s	1.89s	**0.21s**
FMTM	GCC	3.62s	4.00s	3.90s	**0.11s**
	Clang	1.90s	0.53s	0.54s	**0.08s**
HighPass	GCC	0.84s	2.43s	1.07s	**0.15s**
	Clang	0.43s	0.53s	0.89s	**0.04s**
Hybrid	GCC	0.38s	1.17s	1.22s	**0.10s**
	Clang	0.18s	0.27s	0.27s	**0.03s**
LANSwitch	GCC	3.98s	8.88s	6.19s	**0.20s**
	Clang	2.11s	3.03s	2.83s	**0.11s**
LEDLC	GCC	5.55s	11.71s	9.01s	**0.51s**
	Clang	3.67s	3.56s	4.61s	**0.31s**
RAC	GCC	5.20s	6.88s	3.43s	**0.24s**
	Clang	2.31s	1.30s	1.36s	**0.16s**

Compared to other code generators, the statistics show that KNIGHT can still achieve substantial performance improvements. While state-of-the-art compilers employ elaborate optimizations under `-O2` compilation flag, such as function inline and loop align, these optimizations are unable to eliminate the redundant calculations addressed by KNIGHT. Moreover, compared to the aforementioned evaluation under

`-O0` compilation flag, we found that Simulink outperforms DFSynth and HCG in all benchmark models. It indicates that the optimization techniques employed by Simulink are effective under `-O2` compilation flag. Besides, compared with the aforementioned experiments, the code compiled with Clang outperforms the code compiled by Clang. This is probably because Clang implements and employs more optimizations under `-O2` compilation flag compared to GCC.

Comparison experiment using `-O3` flag. To validate the effectiveness of KNIGHT under high-level compiler optimizations, we further conducted the comparison experiment using `-O3` compilation flag. Table V shows the experiment results. For comparison experiments compiled with GCC, the execution duration of KNIGHT is 16.07× faster than Simulink, 9.09× faster than DFSynth, and 11.19× faster than HCG on average. As for compiling using Clang, the execution duration of KNIGHT is 9.63× faster than Simulink, 9.09× faster than DFSynth, and 11.17× faster than HCG on average. The data reveals that under the high-level optimization, KNIGHT still achieves a significant performance improvement. It shows that the redundant operations eliminated by KNIGHT cannot be effectively addressed by the sophisticated optimization techniques implemented by compilers, demonstrating the effectiveness and practicability of our approach.

TABLE V: Comparison of the code execution duration on x86 with `-O3` flag.

Model	Compiler	Simulink	DFSynth	HCG	KNIGHT
CPUTask	GCC	1.11s	1.44s	2.04s	**0.28s**
	Clang	1.11s	1.41s	1.55s	**0.30s**
Diffsion	GCC	0.78s	1.62s	2.52s	**0.21s**
	Clang	0.82s	2.00s	1.89s	**0.30s**
FMTM	GCC	3.72s	0.72s	0.74s	**0.11s**
	Clang	1.84s	0.59s	0.57s	**0.08s**
HighPass	GCC	0.79s	0.55s	0.97s	**0.06s**
	Clang	0.44s	0.53s	0.88s	**0.04s**
Hybrid	GCC	0.36s	0.32s	0.32s	**0.05s**
	Clang	0.18s	0.27s	0.27s	**0.03s**
LANSwitch	GCC	3.98s	1.96s	1.79s	**0.13s**
	Clang	2.15s	2.44s	2.89s	**0.13s**
LEDLC	GCC	1.06s	2.63s	3.41s	**0.24s**
	Clang	1.16s	3.64s	4.67s	**0.29s**
RAC	GCC	5.32s	2.03s	2.35s	**0.17s**
	Clang	2.42s	1.24s	1.34s	**0.17s**

Compared to the aforementioned evaluation under `-O2` compilation flag, we found that DFSynth outperforms Simulink and HCG in benchmark models. It indicates that generating a separate function for each subsystem is more efficient under `-O3` compilation flag. This prompts compilers to optimize each function individually, and releases the analytical burden of the context, compared to the large inline code blocks. Additionally, under `-O3` compilation flag, compilers are highly efficient at minimizing the overhead of function calls. Moreover, compared to the aforementioned experimental results, the efficiency of the code compiled with GCC shows a marked enhancement. By analyzing the specific assembly code, we found that GCC adopts effective and aggressive optimization methods to speed up the execution, such as loop unrolling and SIMD instructions, decreasing the performance gap with Clang. The code generated by KNIGHT similarly benefits from these advancements.

B. Evaluation on Other Platform

To demonstrate the effectiveness of KNIGHT on different architectures, we also conducted repetitive experiments on an embedded machine with an ARM processor (Linux v6.1.21, ARM Cortex A72, 8GB memory), using GCC (v11.3.0) and Clang (v14.0.6) compilers. The overall results on ARM are shown in Figure 4. The red line marks the execution duration of the generated by KNIGHT, while the box represents the performance improvement achieved by KNIGHT compared to the corresponding code generator. For comparison experiments compiled with GCC, the execution duration of KNIGHT is 22.59× faster than Simulink, 20.99× faster than DFSynth, and 17.69× faster than HCG on average, across different optimization levels. As for compiling using Clang, the execution duration of KNIGHT is 20.84× faster than Simulink, 19.03× faster than DFSynth, and 16.25× faster than HCG on average, across different optimization levels. This demonstrates the effectiveness of KNIGHT on the embedded platform.

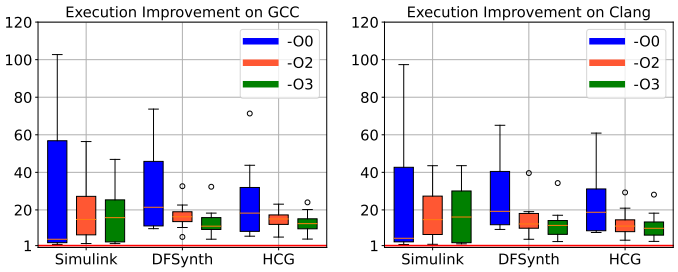


Fig. 4: The comparison experiments on ARM architecture. The red line is the baseline, representing the execution duration of KNIGHT. The box represents the performance improvement achieved by KNIGHT, compared to other code generators.

Compared to experiments on the x86 architecture, we observed that KNIGHT achieves better performance improvement. This is mainly because embedded devices have certain disparities in performance and hardware optimization compared to conventional devices. For example, while the AMD processor used previously supports 512-bit SIMD instructions, the ARM processor in the embedded device only supports 128-bit SIMD instructions. In addition, modern processors support speculative execution, which allows the execution of required operations in advance, whereas embedded processors do not support this technique. Consequently, for embedded devices with limited performance, the main performance limitation is related to the logic of the generated code. KNIGHT benefits from this and thereby achieves higher performance improvement. Furthermore, we analyzed the outliers in Figure 4. The main outliers of DFSynth and HCG are the `LANSwitch` model. As mentioned above (Section VI-A), this model contains more calculation-related blocks rather than decision-related blocks. Therefore, redundant calculations within this model have a greater impact. Consequently, KNIGHT achieves significant performance improvement on this model. In the case of the experiments conducted on Simulink, no outliers were observed.

C. Evaluation on Memory Usage

We further conducted experiments on memory usage among the code generated by different code generators, to fully evaluate our approach. This metric is important, especially on resource-constrained embedded devices.

As shown in Table VI, the code generated by different code generators requires approximately the same amount of memory for execution. Since they use the same number of variables and do not utilize memory allocation functions such as `malloc`, `calloc`, and `new`, the heap size among different compiled executable files remains nearly identical. Stack memory is automatically managed by the compiler and all the code is compiled with the same compiler, thus the stack size is also almost the same. Moreover, due to the relatively small size of the target executable files, other metrics do not have a significant impact on memory usage, such as data segment and text segment. Consequently, the memory usage of the generated code is nearly identical. This demonstrates that KNIGHT improves code performance without incurring substantial additional memory overhead.

TABLE VI: Comparison experiments on memory usage.

Model	Memory Usage (kB)			
	Simulink	DFSynth	HCG	KNIGHT
CPUTask	844	**824**	**824**	**824**
Diffsion	**800**	**800**	**800**	**800**
FMTM	**772**	**772**	776	**772**
HighPass	804	**800**	804	804
Hybrid	800	**756**	760	**756**
LANSwitch	**792**	800	800	800
LEDLC	**788**	792	792	792
RAC	780	780	**772**	**772**

D. Comparison with Polyhedral Model Optimizations

The polyhedral model is a high-level mathematical model within compilers, mainly used for loop optimization and parallelization. This model can efficiently analyze and transform loop nesting structures in code by abstracting loop structures into polyhedra. To determine if polyhedral model optimizations can eliminate redundant calculations, we further conducted the comparison experiments and enabled polyhedral model optimizations under `-O3` flag: `-fgraphite` for GCC and `-mllvm -polly` for Clang. For code generated by Simulink, DFSynth, and HCG, we enabled these specific flags; for the code generated by KNIGHT, we only enabled `-O3` flag. The experiments were carried out on the previously mentioned x86 industrial machine.

Table VII shows the experiment results. For comparison experiments compiled with GCC, the execution duration of KNIGHT is 15.52× faster than Simulink, 8.79× faster than DFSynth, and 11.56× faster than HCG on average. For comparison experiments compiled with Clang, the execution duration of KNIGHT is 9.73× faster than Simulink, 9.97× faster than DFSynth, and 11.82× faster than HCG on average. We found that, compared to the results using `-O3` flag, the generated code for most benchmark models is a little faster by enabling polyhedral model optimizations. However, for some models, the performance of the generated code significantly decreased. For example, consider `CPUTask` model, the code generated by DFSynth and HCG compiled by Clang is 2.03× slower and 1.87× slower. Through performance analysis of the generated code, we found that after using polyhedral model optimizations, control logic constitutes more execution duration, which because the main reason for the performance decline. Besides, experiment results illustrate that although enabling polyhedral model optimizations, KNIGHT still can achieve significant performance improvement.

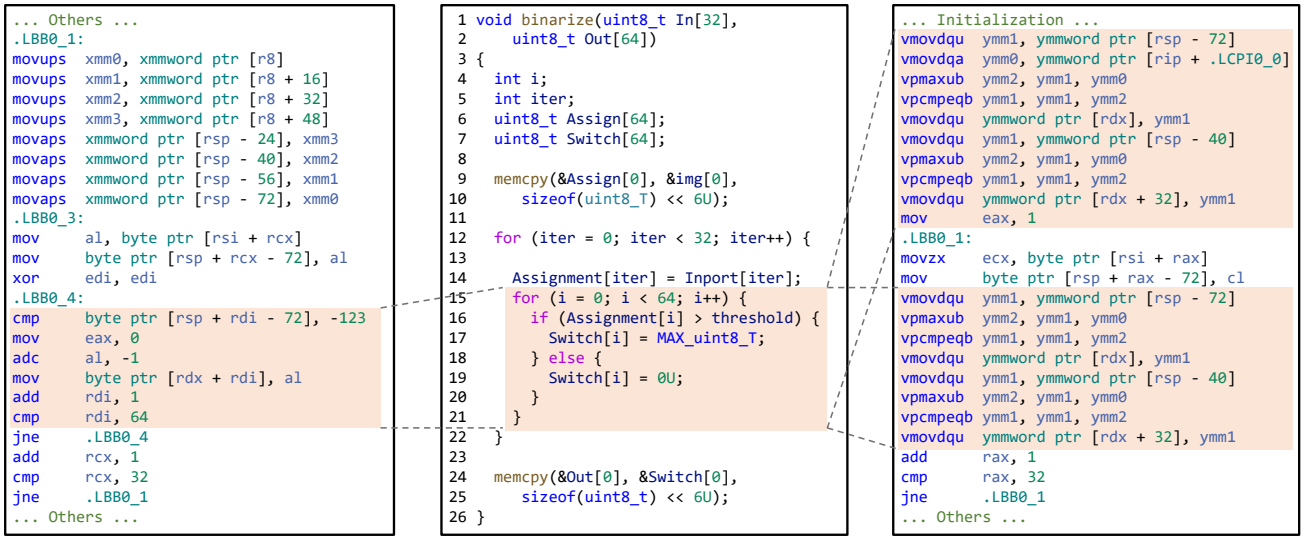
TABLE VII: Comparison of the code execution duration on x86 with polyhedral model optimizations.

Model	Compiler	Simulink	DFSynth	HCG	KNIGHT
CPUTask	GCC	1.06s	1.36s	1.90s	**0.28s**
	Clang	1.10s	2.86s	2.89s	**0.30s**
Diffsion	GCC	0.75s	1.54s	2.40s	**0.21s**
	Clang	0.77s	2.00s	1.88s	**0.30s**
FMTM	GCC	3.72s	0.67s	0.67s	**0.11s**
	Clang	1.75s	0.52s	0.52s	**0.08s**
HighPass	GCC	0.72s	0.52s	0.92s	**0.06s**
	Clang	0.42s	0.54s	0.89s	**0.04s**
Hybrid	GCC	0.33s	0.30s	0.30s	**0.05s**
	Clang	0.17s	0.20s	0.20s	**0.03s**
LANSwitch	GCC	3.67s	1.83s	1.69s	**0.13s**
	Clang	2.00s	2.34s	2.65s	**0.13s**
LEDLC	GCC	0.99s	3.03s	5.03s	**0.24s**
	Clang	1.10s	3.27s	4.76s	**0.29s**
RAC	GCC	5.47s	1.83s	2.19s	**0.17s**
	Clang	2.44s	1.25s	1.09s	**0.17s**

VII. DISCUSSION

Extensibility of KNIGHT. Currently, KNIGHT supports code generation not only for x86 and ARM architectures, but also supports optimized code for other architectures. The fundamental reason for performance improvement comes from avoiding redundant calculations. Therefore, the code employed on other architectures will also benefit from our approach. At present, the benchmark models in experiments are widely used industry models. KNIGHT handles these models effectively and achieves significant performance improvement. However, for more complex models, e.g., those containing over 10000 blocks, identifying redundant operations becomes exceedingly complex and time-consuming during code generation for analysis. A feasible approach is to treat some complex subsystems within loop-semantic subsystems as state-independent blocks and not subject them to further analysis, thus reducing the burden of code generation. This represents a trade-off between tool efficiency and code performance.

KNIGHT supports loop reshaping for Simulink models to enhance the performance of the generated code. The code generation part of KNIGHT can be extended to models constructed by other model-driven design tools. Different model-driven design tools have their representations for constructed models. Therefore, KNIGHT requires to identify the differences between their representations and those of Simulink models, and then parse them into runtime data structures for analysis. As loop reshaping of KNIGHT, this optimization technique can also applied to other model-driven design tools, but it



(a) Compiled with –O1 (b) Original Code (c) Compiled with –O3

Fig. 5: The compiled assembly code. The left snippet is compiled at `-O1` flag. The right snippet is compiled at `-O3` flag. The corresponding assembly code of redundant calculations is highlighted with light orange. This shows that the sophisticated optimization techniques employed by GCC cannot effectively address the redundant calculations in the generated code.

should meet the following conditions. First, loop-semantic subsystems supported by these tools should be open, similar to *for* statement in C. The loop-semantic subsystems need to support execution for blocks with different data dimensions. Second, these tools should support blocks that contain internal states. Once the above two conditions are violated, it becomes difficult for loop reshaping to effectively identify redundant operations. For example, SCADE supports loop iterators for modeling. However, the semantics of these iterators are strict. For instance, *map* iterator in SCADE strictly requires that each iteration accesses the corresponding element, and the data dimensions of blocks within the iterator are consistent. As a result, the redundant calculations found in Simulink models do not occur in SCADE models, preventing KNIGHT from effectively optimizing SCADE models.

Compiler Optimizations. The inherent inefficiency of the generated code makes it difficult for compilers to employ aggressive optimization techniques or achieve satisfactory results. For instance, consider the code generated by Simulink Embedded Coder (Figure 5b). The left snippet is the assembly code compiled by GCC under `-O1` flag, while the right snippet is the assembly code under `-O3` flag with polyhedral model optimizations. For the assembly code under `-O3` flag, GCC utilizes the loop unrolling technique to unroll the inner loop statement (highlighted with light orange in Figure 5b), and splits the inner loop statement into two parts for optimization. Then, it utilizes SIMD instructions, such as `vpmaxub` and `vpcmpeqb`, to speed up the calculation and improve overall performance. However, the optimized assembly code still requires repeating the above instructions 32 times (see `cmp` and `jne` instruction), resulting in executing time-consuming redundant instructions. Compilers have implemented numerous loop-related optimizations, such as `-floop-unroll-and-jam` and `-fmove-loop-invariants`. However, to address the aforementioned issues, compilers must strategically combine these optimization techniques. The complex model semantics pose challenges to compilers in deploying these aggressive optimizations. For instance, the intricate data types and connections make it difficult for compilers to identify the invariant variables within the loop semantic.

In fact, existing loop-related optimizations can be employed in code generation for performance improvement, and KNIGHT specifically targets the elimination of redundant calculations caused by state-dependent blocks. For example, loop fusion can merge multiple loops to avoid repeated loop condition checks. However, the effective application of these optimization techniques necessitates the targeted utilization of model semantics to achieve better performance and ensure the correctness of the generated code. Our future work will focus on exploring strategies to effectively integrate model semantics with these optimizations during code generation.

Code Correctness. We have employed a series of effective approaches to ensure code correctness. First, for each model, we randomly generate numerous test cases to the code generated by KNIGHT, and compare them with results derived from model simulations. Additionally, for benchmark models, we have verified the consistency between the code and simulation results, confirming their reliability. Second, in loop reshaping, KNIGHT strategically relocates the subsequent blocks and irrelevant blocks outside the loop semantic to avoid redundant calculations. For subsequent blocks, if they cannot be relocated, it means either they are related to the state calculation of state-dependent blocks, or they are state-dependent blocks. In such cases, KNIGHT will not identify these blocks as subsequent blocks, which is contrary to the definition (Section IV-C). Similarly, for irrelevant blocks, if they cannot be relocated, it implies that they are state-dependent blocks, which is also contrary to the definition (Section IV-C).

Optimization Approaches. To eliminate redundant calculations, KNIGHT relocates the subsequent and irrelevant blocks outside the loop semantics, while maintaining their connections within the dataflow graph. Another approach for implementing the aforementioned optimizations is model rewriting. Specifically, after identifying optimizable blocks, the model files are rewritten to relocate these blocks, using Simulink Embedded Coder for code generation. However, this method presents several challenges. For instance, model rewriting must preserve model semantics to ensure correctness, and incorrect modifications can result in model files not opening properly. Moreover, implementing KNIGHT during code generation facilitates the integration of additional loop-related optimizations.

Threats to Validity. At present, KNIGHT identifies internal-state blocks within Simulink models and designs corresponding interaction rules to determine whether they are state-dependent. Indeed, Simulink offers specialized blocks for domain-specific modeling, such as powertrain blockset, which may contain interaction rules for identifying state-dependent blocks not yet covered by Knight. However, KNIGHT supports "*commonly-used blocks*" in Simulink, including but not limited to math operation blockset, matrix operation blockset, DSP system blockset, logic blockset, etc, ensuring the practicality and effectiveness of KNIGHT. Enriching interaction rules for these domain-specific blocks is our future work. Moreover, for blocks within the loop-semantic, KNIGHT generates separate *for* statements to represent their functionalities. While this approach leads to increased lines of code and could potentially incur overhead for loop boundary judgments, our empirical evaluation suggests that, with compiler optimizations at `-O2` flag or higher, this overhead does not adversely affect execution duration, due to loop-fusion optimizations, such as `ftree-loop-fusion`.

VIII. Related Work

Recently, many research and commercial tools have made remarkable efforts to improve the performance of the generated code. Specifically, the built-in Simulink Embedded Coder, specialties in generating production-quality code that can be directly deployed to the target hardware platform. It employs various high-level optimization techniques to improve the performance of the generated code, including expression folding, variable reuse, SIMD instruction replacement, etc. In academics, some works have made some efforts to improve code efficiency. DFSynth concentrates on the dataflow analysis, scheduling, and code synthesis steps of the code generation process [12]. It is capable of generating well-structured code for Ptolemy-II and Simulink models. Initially, DFSynth disassembles the dataflow model into blocks embedded within if-else or switch-case statements based on schedule analysis, effectively bridging the semantic gap between the code and the original dataflow model. Subsequently, DFSynth designs tailored templates for each block and synthesizes well-structured, executable C and Java code using sequential code assembly. HGC [11], another approach, emphasizes the parallel acceleration of the code generated. Based on adaptive pre-calculation on input scales, HCG selects optimal implementations for intensive computing blocks on different scenarios. For batch computing blocks, HCG synthesizes appropriate SIMD instructions using iterative dataflow graph mapping.

IX. Conclusion

In this paper, we propose KNIGHT, an efficient code generator for loop-semantic Simulink models with *loop shaping*. For the target Simulink model, KNIGHT identifies blocks with internal states, and implements the specific interaction rules to determine state-dependent blocks. Then, for state-dependent blocks, KNIGHT conducts forward inference to obtain their preceding blocks, which influence the internal state calculations. After that, KNIGHT isolates blocks that are optimizable and irrelevant to internal states, and strategically relocates them outside the loop semantics. We evaluate the effectiveness of KNIGHT on benchmark models across different compilers and architectures. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by KNIGHT is 16.58× faster, 16.89× faster, and 15.38× faster in terms of execution duration on average, without incurring additional overhead of memory usage.

X. Acknowledgment

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), China Postdocoral Science Foundation (BX20230183, 2023M731954) and NSFC Program (No. 92167101, 62021002, 62372263).

References

[1] Simulink and Matlab, *Simulink Documentation*, 2023, https://www.mathworks.com/help/simulink/index.html.

[2] Y. Zhu, H. Hu, G. Xu, and Z. Zhao, "Hardware-in-the-loop simulation of pure electric vehicle control system," in *2009 International Asia Conference on Informatics in Control, Automation and Robotics*. IEEE, 2009, pp. 254–258.

[3] R. A. Faris, A. Ibrahim, M. Abdulwahid, M. Mosleh *et al.*, "Optimization and enhancement of charging control system of electric vehicle using matlab simulink," in *IOP Conference Series: Materials Science and Engineering*, vol. 1105, no. 1. IOP Publishing, 2021, p. 012004.

[4] D. Christhilf and B. Bacon, "Simulink-based simulation architecture for evaluating controls for aerospace vehicles (sarec-asv)," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2006, p. 6726.

[5] D. Hercog and K. Jezernik, "Rapid control prototyping using matlab/simulink and a dsp-based motor controller," *International Journal of Engineering Education*, vol. 21, no. 4, p. 596, 2005.

[6] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of linux kernels with preempt_rt on arm-based embedded devices," *Electronics*, vol. 10, no. 11, p. 1331, 2021.

[7] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Slnet: A redistributable corpus of 3rd-party simulink models," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 237–241.

[8] S. Sankaranarayanan and G. Fainekos, "Simulating insulin infusion pump risks by in-silico modeling of the insulin-glucose regulatory system," in *International Conference on Computational Methods in Systems Biology*. Springer, 2012, pp. 322–341.

[9] Z. Su, D. Wang, Z. Yu, Y. Yang, Y. Jiang, R. Wang, W. Chang, W. Li, A. Cui, and J. Sun, "Phcg: Optimizing simulink code generation for embedded system with simd instructions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 4, pp. 1072–1084, 2022.

[10] Simulink, *Simulink Embedded Coder Documentation*, 2023, https://www.mathworks.com/solutions/embedded-code-generation.html.

[11] Z. Su, Z. Yu, D. Wang, Y. Yang, Y. Jiang, R. Wang, W. Chang, and J. Sun, "Hcg: optimizing embedded code generation of simulink with simd instruction synthesis," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1033–1038.

[12] 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*, vol. 41, no. 1, pp. 49–61, 2021.

[13] Z. Yu, Z. Su, Y. Yang, J. Liang, Y. Jiang, A. Cui, W. Chang, and R. Wang, "Mercury: Instruction pipeline aware code generation for simulink models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4504–4515, 2022.

[14] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, no. 2, pp. 33–40, 2006.

[15] 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.

[16] J.-M. Jézéquel, "Model driven design and aspect weaving," *Software & Systems Modeling*, vol. 7, pp. 209–218, 2008.

[17] Y. Jiang, H. Liu, H. Song, H. Kong, R. Wang, Y. Guan, and L. Sha, "Safety-assured model-driven design of the multifunction vehicle bus controller," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 10, pp. 3320–3333, 2018.

[18] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D. Méndez, A. Moreira *et al.*, "Dealing with non-functional requirements in model-driven development: A survey," *IEEE Transactions on Software Engineering*, 2019.

[19] 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.

[20] 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.

[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] P. Baldwin, S. Kohli, and E. A. Lee, "Modeling of sensor nets in ptolemy ii," in *Proceedings of the 3rd international symposium on Information processing in sensor networks*. ACM, 2004, pp. 359–368.

[23] C. Brooks, E. A. Lee, and S. Tripakis, "Exploring models of computation with ptolemy ii," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2010, pp. 331–332.

[24] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[25] Z. Su, D. Wang, Y. Yang, Z. Yu, W. Chang, W. Li, A. Cui, Y. Jiang, and J. Sun, "Mdd: A unified model-driven design framework for embedded control software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[26] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 1–26, 2013.

[27] G. Zhou, M.-K. Leung, and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," in *International Conference on Embedded Software and Systems*. Springer, 2007, pp. 193–206.

[28] R. Grepl, "Real-time control prototyping in matlab/simulink: Review of tools for research and education in mechatronics," in *2011 IEEE International Conference on Mechatronics*. IEEE, 2011, pp. 881–886.

[29] A. Tewari, *Automatic control of atmospheric and space flight vehicles: design and analysis with MATLAB® and Simulink®*. Springer, 2011.

[30] B. Gatos, I. Pratikakis, and S. J. Perantonis, "Adaptive degraded document image binarization," *Pattern recognition*, vol. 39, no. 3, pp. 317–327, 2006.

[31] G. K. Adam, "Real-time performance analysis of distributed multithreaded applications in a cluster of arm-based embedded devices," *International Journal of High Performance Systems Architecture*, vol. 11, no. 2, pp. 105–116, 2022.

[32] Z. Su, Z. Yu, D. Wang, Y. Yang, R. Wang, W. Chang, A. Cui, and Y. Jiang, "Stcg: state-aware test case generation for simulink models," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

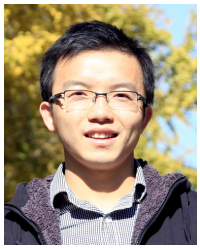
Zehong Yu received the B.S. degree in software engineering from Southeast University in 2021. He is pursuing a M.S.E. degree in software engineering at Tsinghua University, Beijing, China. His research interests are in the areas of model driven development and embedded software engineering.

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.

Zhuo Su received the B.S. degree in software engineering from Northeastern University, Shenyang, China, in 2018, and the Ph.D. degree in software engineering from Tsinghua University, Beijing, China, in 2023. He is currently working as a postdoctoral fellow with the School of Software, Tsinghua University, Beijing, China. His research interests are in the areas of model driven development and embedded software engineering.

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.

Yang Tao 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.



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.